MARKDOWN

```
# 🎯 Project Charter: SHA-256 Hash Function

## What You Are Building

A standalone cryptographic library that implements the SHA-256 (Secure Hash Algorithm 256) exactly as defined in the NIST FIPS 180-4 specification. You will build a stateful engine that consumes arbitrary-length data and produces a unique 256-bit "fingerprint." By the end, you will have a streaming API (init, update, finalize) that can hash files of any size and verify its own correctness against official NIST test vectors.

## Why This Project Exists

Most developers treat cryptographic hashes as "black boxes"—magic functions that turn data into hex strings. Building SHA-256 from scratch shatters this abstraction, exposing how bit-level operations (rotations, shifts, and logical choices) create collision-resistant security. You will learn to translate formal mathematical specifications into executable code, a critical skill for security engineering and systems programming.

## What You Will Be Able to Do When Done

- **Implement Message Padding:** Manually append the '1' bit, zero-fill, and encode 64-bit lengths to align data into 512-bit blocks.

- **Perform Bit-Level Diffusion:** Use bitwise rotation and XOR to expand 16 input words into a 64-word "message schedule."

- **Build the Compression Engine:** Implement the 64-round loop using the Ch (Choice), Maj (Majority), and Sigma logical functions.

- **Manage Cryptographic State:** Design an `init/update/finalize` streaming API that handles partial blocks and memory cleanup.

- **Debug Bitwise Logic:** Use intermediate value validation to track down off-by-one errors in bit-rotations and modular addition.

## Final Deliverable

A C library (or equivalent in your chosen language) consisting of approximately 300–500 lines of code. It includes a `SHA256_CTX` structure for state management and a test suite that successfully validates the hashes for an empty string, the string "abc", and a 56-byte boundary case against NIST standards.

## Is This Project For You?

**You should start this if you:**

- Understand binary, hexadecimal, and how data is stored in bytes.

- Are comfortable with bitwise operators: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), and shifts.

- Want to move past using libraries and start understanding the internals of data integrity.

- Can manage basic memory (pointers and structs in C, or byte-arrays in Python/JS).

**Come back after you've learned:**

- [Binary and Bitwise Basics](https://en.wikipedia.org/wiki/Bitwise_operation)

- [C Programming Fundamentals](https://www.learn-c.org/) (specifically Structs and Typedefs)

## Estimated Effort

| Phase | Time |

|-------|------|

| Message Preprocessing and Padding | ~2-3 hours |

| Message Schedule Generation | ~2-3 hours |

| Compression Function (64 Rounds) | ~4-5 hours |

| Final Hash Output and Validation | ~2-4 hours |

| **Total** | **~10-15 hours** |

## Definition of Done

The project is complete when:
```

```
- `SHA-256("")` produces `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`.
- `SHA-256("abc")` produces `ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`.
- The 56-byte boundary test case passes (verifying that the two-block padding logic is correct).
- A message fed into the `update()` function in 1-byte chunks produces the same hash as the message fed in all at once.
- The code successfully clears its internal buffers after finalization (Secret Erasure).
```

SHA-256 Hash Function

This project implements SHA-256 entirely from the NIST FIPS 180-4 specification, building every component by hand: message preprocessing and padding, message schedule generation with σ0/σ1 functions, the 64-round compression function with Ch/Maj/Σ0/Σ1, and final hash output with streaming API support. The learner will transform a formal cryptographic specification into working code, gaining deep intuition for how a Merkle-Damgård hash function converts arbitrary-length input into a fixed 256-bit digest through iterative compression of 512-bit blocks.

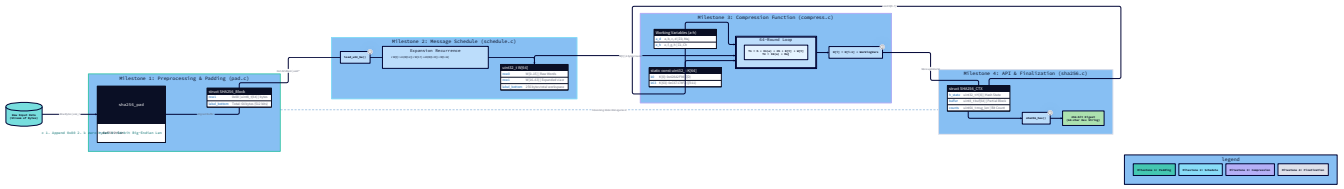
The journey starts at the data boundary — how raw bytes become padded, aligned blocks — then moves inward through the message schedule expansion that amplifies 16 input words into 64 round inputs, through the compression function where eight working variables are churned through 64 rounds of bitwise logic and modular arithmetic, and finally to the output stage where accumulated state becomes a hex digest. Every stage is validated against NIST's own intermediate computation examples, so correctness is never a matter of opinion.

This is a 'Build Your Own' project: we are constructing the hash engine itself, not merely using one. Every diagram and explanation looks INTO the algorithm's internals — bit rotations, word expansions, state transformations — rather than treating SHA-256 as a black box.

Milestone 1: Message Preprocessing and Padding

Where We Are

Before SHA-256 can crunch a single bit through its compression function, it has to answer a deceptively simple question: *how do you feed arbitrary-length input to a machine that only speaks in 512-bit chunks?* The answer is padding — and not the naive "add some zeros until it fits" kind. SHA-256's padding is a precisely engineered cryptographic contract, and understanding every byte of it is the foundation everything else rests on.



In the full pipeline diagram above, you're working on the very first stage: the raw message bytes enter on the left, and what emerges from your work today is an array of aligned 64-byte blocks ready for the compression engine. Get this wrong, and no amount of perfect compression-function code will save you — the output will be a valid-looking 64-character hex string that is simply the wrong hash for every input you feed it. Silently wrong. That's the worst kind of bug.

The Misconception Worth Shattering

Here's what most developers assume when they first see SHA-256 padding: *"Oh, it's just zero-padding to round up to the block size — like padding a network packet or null-terminating a string. The '1' bit and*

length field are bookkeeping. I'll knock this out in ten minutes." **That mental model will send you chasing a subtle, silent bug. Let's break open the real story.**

Why Padding Has to Be Cryptographically Precise

Imagine two messages:

```
Message A: "abc\x00\x00\x00"   (6 bytes)
Message B: "abc"               (3 bytes)
```

If you pad both messages by simply zero-filling to 64 bytes, they produce **identical padded blocks**. A hash function operating on those blocks produces **identical output**. Two different messages, same hash — a trivial collision induced by your padding scheme. This is called **padding ambiguity**, and it breaks the fundamental property a hash function must have: every distinct message must produce a distinct digest (collision resistance starts at the input boundary, not inside the compression function). The **'1' bit** is the domain separator that makes this impossible. After your message ends, you always append a `0x80` byte (which is `10000000` in binary — exactly a '1' bit followed by seven '0' bits). Now "abc" padded looks like:

```
61 62 63 80 00 00 00 00 ...
```

And "abc\x00\x00\x00" padded looks like:

```
61 62 63 00 00 00 80 00 ...
```

The '1' bit marks where the message ends. No matter how many trailing zeros the message itself contains, the separator is always unambiguous. The **64-bit length field** at the end reinforces this with a second commitment: it says exactly how many bits the original message contained. Even in a hypothetical world where two messages somehow reached the same padded body (which the '1' bit already prevents), they'd differ in their length fields unless they were genuinely the same message.

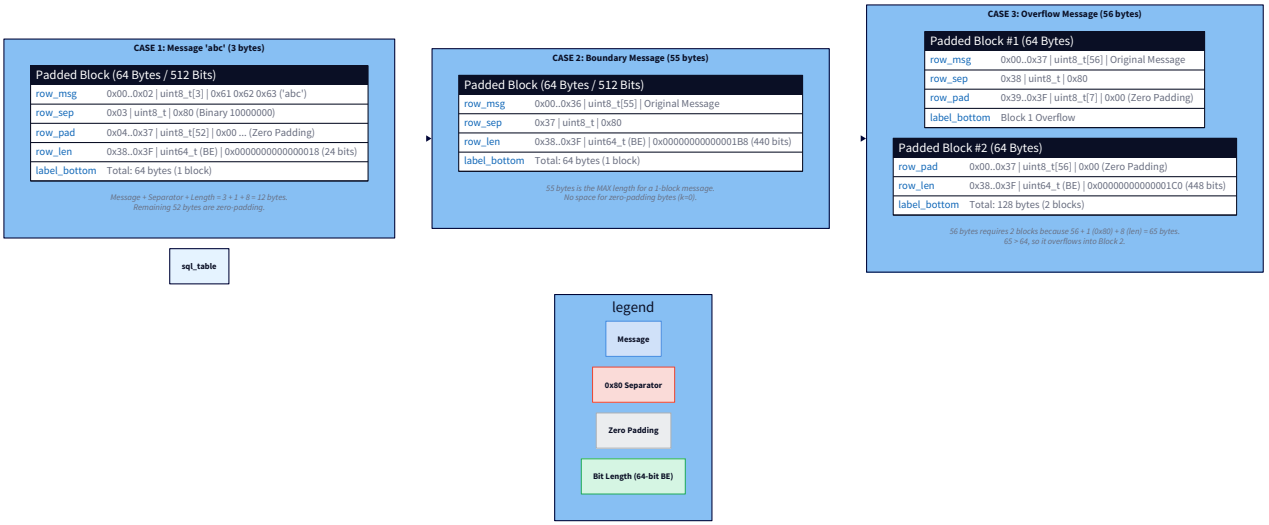
> 🔑 **The insight**: SHA-256 padding is not "round up to block size." It is a **self-describing, unambiguous encoding** of message length into the block structure. The design is closer to a protocol header than to a zero-padding convenience function.

The Merkle-Damgård Foundation

> 🔑 **Foundation: The Merkle-Damgård construction: how SHA-256 processes multiple blocks by iteratively feeding the output of one compression into the next**
>
> 1. **What it IS**:
>
> The Merkle-Damgård construction is a blueprint for building a cryptographic hash function that can handle messages of any size. Since mathematical "compression functions" usually require a fixed input size (like exactly 512 bits), Merkle-Damgård provides a way to break a long message into equal-sized blocks and process them one by one. The output of the first block's compression is fed as an input into the next block's compression, creating a chain that eventually produces the final hash.

2. **WHY you need it right now**: SHA-256 is a "Merkle-Damgård" hash. When you implement it, you aren't just running one massive calculation on your data. Instead, you are maintaining a "state" (the 256-bit intermediate hash). You must pad your message so it fits perfectly into 512-bit blocks, then iteratively update that state block-by-block. Understanding this construction explains why SHA-256 is susceptible to "length extension attacks" and why the order of blocks is vital to the final result.
3. **Key Insight**: **The Relay Race Model.** Imagine a relay race where each runner (a message block) doesn't just pass a baton, but adds their own unique DNA to it. The "baton" is the internal state of the hash. By the time the last runner finishes, the baton contains a summarized history of every single runner that touched it. If you change even one runner in the middle, the final baton will look completely different.

The key thing you need right now: SHA-256 is built on the **Merkle-Damgård construction**, which means it processes your message one 512-bit block at a time, threading a running "hash state" from block to block. The final hash state becomes your digest. This architecture has a direct consequence for padding: every block must be exactly 512 bits. Not approximately 512 bits — exactly. The compression function is a fixed-size machine. Your padding function's job is to manufacture that guarantee.



The FIPS 180-4 Padding Rules, Precisely

The NIST FIPS 180-4 specification (Section 5.1.1) states the padding rules in three steps. Here they are translated from bureaucratic specification language into concrete terms: Step 1: Append the '1' bit. Immediately after the last message byte, append the byte `0x80`. This is the '1' bit followed by seven '0' bits, giving you byte-aligned handling for free. (If you were operating at the bit level, you'd append exactly one '1' bit — but since SHA-256 is always used with byte-aligned input in practice, appending `0x80` is correct.) Step 2: Append zero bytes until the message length is congruent to 448 mod 512 bits. In byte terms: append `0x00` bytes until the total padded length (message + `0x80` + zeros) is congruent to 56 mod 64 bytes. This leaves exactly 8 bytes (64 bits) at the end of the block for the length field. Step 3: Append the 64-bit big-endian message length. The original message length in bits (not bytes) is written as a big-endian 64-bit integer into those final 8 bytes. The result: a padded message whose total length is a multiple of 64 bytes (512 bits). ✓

Big-Endian: What It Means and Why SHA-256 Mandates It

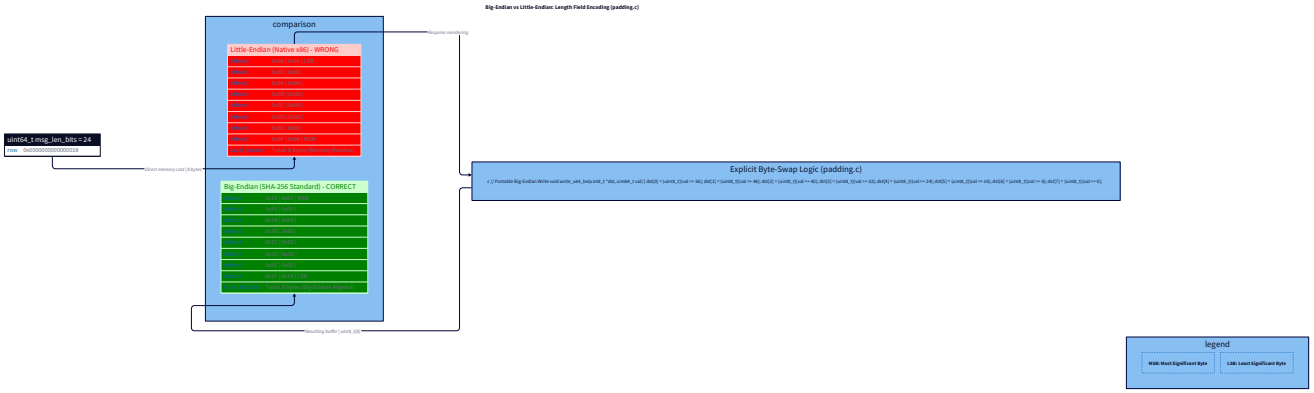
> 🔑 **Foundation: Big-endian vs. little-endian byte ordering in multi-byte integers**
>
> 1. **What it IS**:
>
> Endianness refers to the order in which bytes are stored in memory for multi-byte data types (like a 32-bit integer). In **Big-endian**, the "big end" (the most significant byte) comes first. If you have the hex value `0x12345678`, a big-endian system stores it in memory as `12 34 56 78`. This matches how we write numbers on paper from left to right. Contrast this with **Little-endian** (used by Intel/AMD processors), which would store it as `78 56 34 12`.

2. **WHY you need it right now**: The SHA-256 specification (FIPS 180-4) is explicitly defined using big-endian byte ordering. All constants and all message processing must be treated as big-endian. Because most modern computers use little-endian hardware, you cannot simply cast a byte array to an integer array and expect the math to work. You must manually "swap" the byte order when loading data from the message buffer into the 32-bit words that the SHA-256 algorithm operates on.
3. **Key Insight**: **Big-endian is "Network/Protocol Order."** Think of it as the "Human Standard." While computers often prefer Little-endian for hardware efficiency, humans and cryptographic specifications almost always prefer Big-endian because it is easier to read and matches the standard way we represent numerical significance.

Here's the short version you need right now: When you write a number that spans multiple bytes — like the 64-bit length field — you have a choice about which byte goes first in memory. **Big-endian** puts the most-significant byte (the "big" end) first. **Little-endian** puts the least-significant byte first. Consider the number `0x0000000000000018` (which is 24 in decimal — the bit length of "abc"):

```
Big-endian    (SHA-256):  00 00 00 00 00 00 00 18
Little-endian (x86 CPU):  18 00 00 00 00 00 00 00
```

SHA-256 mandates **big-endian throughout** — for the length field, for parsing message words, for the final hash output. On x86 processors (which are little-endian), your code must explicitly byte-swap. This is not an optimization detail; it is a correctness requirement. Get it wrong and your hash will differ from every reference implementation, every test vector, and every cryptographic library in existence.



The function you'll write to encode the length field big-endian in C:

```c
// Write a 64-bit value as 8 big-endian bytes into dst

static void write_u64_be(uint8_t *dst, uint64_t value) {

    dst[0] = (uint8_t)(value >> 56);

    dst[1] = (uint8_t)(value >> 48);

    dst[2] = (uint8_t)(value >> 40);

    dst[3] = (uint8_t)(value >> 32);

    dst[4] = (uint8_t)(value >> 24);

    dst[5] = (uint8_t)(value >> 16);

    dst[6] = (uint8_t)(value >>  8);

    dst[7] = (uint8_t)(value >>  0);

}
```

Each shift extracts one byte, starting from the most-significant. The cast to `uint8_t` keeps only the lowest 8 bits after shifting. This is explicit, portable, and correct on any platform regardless of CPU endianness.

The Critical Boundary: 55 Bytes vs. 56 Bytes

This is the #1 beginner bug, and it costs hours of debugging because the result doesn't crash — it just produces the wrong hash. A single 512-bit (64-byte) block has this structure:

```
[ message bytes | 0x80 | zero padding | 8-byte length ]
                                          ^^^^^^^^
                                      must fit here
```

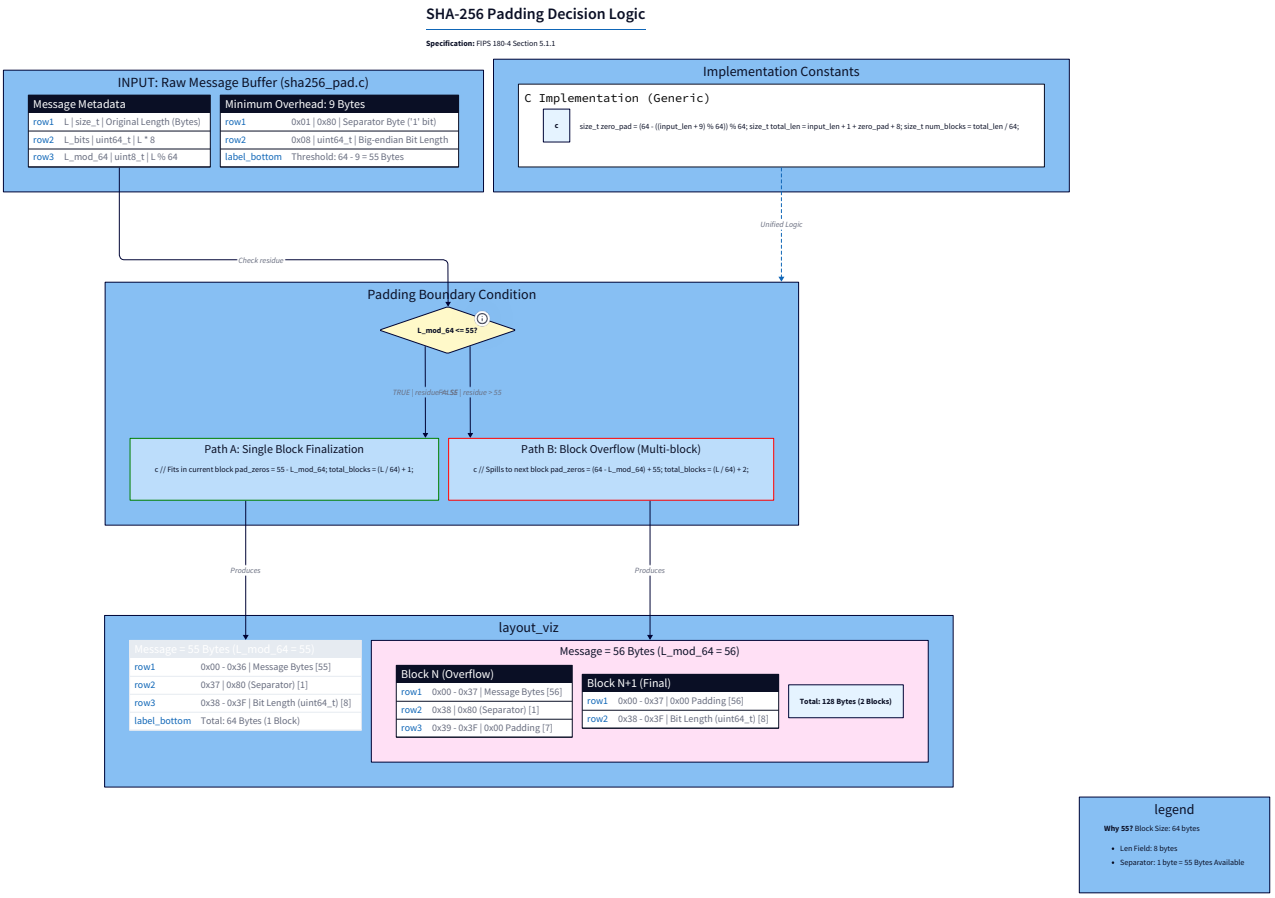
For everything to fit in one block, you need:

- The message itself
- 1 byte for `0x80`

- At least 0 bytes of zero padding (maybe none needed)
- Exactly 8 bytes for the length So the maximum message length that fits in a single block is:

```
64 - 1 (for 0x80) - 8 (for length) = 55 bytes
```

A 55-byte message fits in one block. A 56-byte message requires two blocks.



Let's trace both cases: **55-byte message → 1 block:**

```
Bytes 0–54:   message (55 bytes)
Byte  55:     0x80
Bytes 56–63:  8-byte big-endian length = 0x00000000000001B8 (440 in decimal = 55 × 8)
Total:        64 bytes = 1 block ✓
```

56-byte message → 2 blocks:

```
Block 1:
  Bytes 0–55:   message (56 bytes)
  Byte  56:     0x80
  Bytes 57–63:  seven 0x00 bytes (zero padding, partially fills this block)
Block 2:
  Bytes 64–119: 56 zero bytes (more zero padding)
  Bytes 120–127: 8-byte big-endian length = 0x00000000000001C0 (448 = 56 × 8)
Total:          128 bytes = 2 blocks ✓
```

The confusion arises because developers think "one block is 64 bytes, my message is 56 bytes, so there are 8 bytes left — perfect for the length field." They forget the `0x80` byte takes one of those 8 bytes, leaving

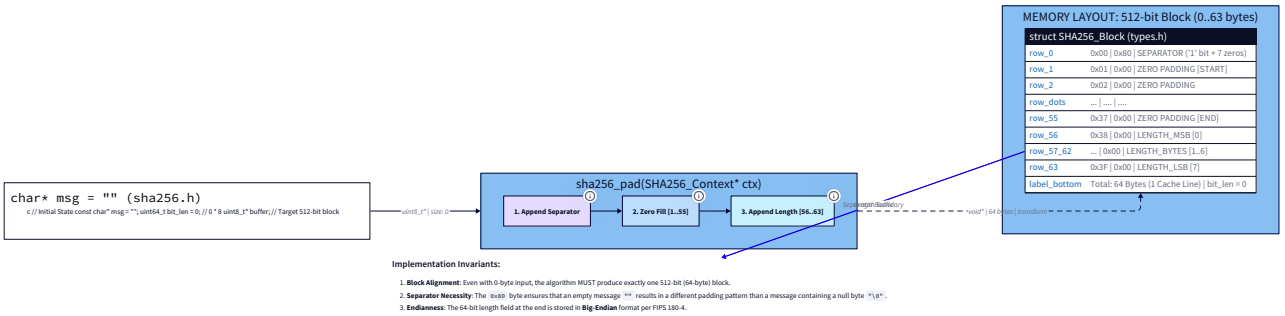
only 7 bytes for the length, which requires 8. That's 1 byte short — overflow into a second block. The formula: a message of `len` bytes fits in a single padded block if and only if `len <= 55`.

Walking Through the Empty Input Case

The empty input case (`len = 0`) is both the simplest and the most useful for verification, because the NIST test vectors give you the expected hash for it:

```
SHA-256("") = e3b0c44298fc1c149afbf4c8996fb924
              27ae41e4649b934ca495991b7852b855
```

Let's trace exactly what your padding function produces for empty input:




```
Byte   0:     0x80    ← the '1' bit (message is empty, so padding starts immediately)
Bytes  1–55:  0x00    ← 55 zero bytes of padding
Bytes 56–63:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00  ← 64-bit length = 0 (zero bits of message)
Total: 64 bytes = 1 block ✓
```

The `0x80` goes in byte 0 because the message occupies zero bytes — padding starts at position 0. The length field is all zeros because the original message was 0 bits long. This produces exactly one block.

The Padding Algorithm in C

Now let's build the actual function. Your function receives a pointer to the input bytes and the input length, and returns a dynamically allocated array of padded blocks. First, the data structures you'll need:

```c
#include <stdint.h>

#include <stddef.h>

#include <stdlib.h>

#include <string.h>

#define SHA256_BLOCK_SIZE 64   /* 512 bits in bytes */

/* A single 512-bit block */

typedef struct {

    uint8_t bytes[SHA256_BLOCK_SIZE];

} SHA256_Block;
```

Now the padding function:

```c
/*
 * sha256_pad - Pad a message according to FIPS 180-4 Section 5.1.1
 *
 * input:      pointer to the raw message bytes
 * input_len:  length of the message in bytes
 * num_blocks: OUTPUT — set to the number of 512-bit blocks produced
 *
 * Returns a heap-allocated array of SHA256_Block. Caller must free().
 * Returns NULL on allocation failure.
 */
SHA256_Block *sha256_pad(const uint8_t *input, size_t input_len,
                         size_t *num_blocks) {
    /*
     * Calculate the total padded length in bytes.
     *
     * We need:
     *   input_len bytes    (the message)
     *   + 1 byte           (the 0x80 separator)
     *   + k bytes          (zero padding, k >= 0)
     *   + 8 bytes          (the 64-bit big-endian length field)
     *
     * The total must be a multiple of 64.
     *
     * input_len + 1 + k + 8 ≡ 0 (mod 64)
     * => (input_len + 9 + k) ≡ 0 (mod 64)
     * => k ≡ -(input_len + 9) (mod 64)
     * => k = (64 - ((input_len + 9) % 64)) % 64
     *
     * Note: the outer % 64 handles the case where (input_len + 9) is
     * already a multiple of 64 (k would be 64 without it, but should be 0).
     */
    size_t zero_pad = (SHA256_BLOCK_SIZE - ((input_len + 9) % SHA256_BLOCK_SIZE))
                      % SHA256_BLOCK_SIZE;
    size_t total_len = input_len + 1 + zero_pad + 8;
    /* total_len must be a multiple of 64 — verify with assert in debug builds */
    /* assert(total_len % SHA256_BLOCK_SIZE == 0); */
    *num_blocks = total_len / SHA256_BLOCK_SIZE;
    /* Allocate the padded buffer (zero-initialized) */
    uint8_t *buf = (uint8_t *)calloc(total_len, 1);
    if (!buf) {
```

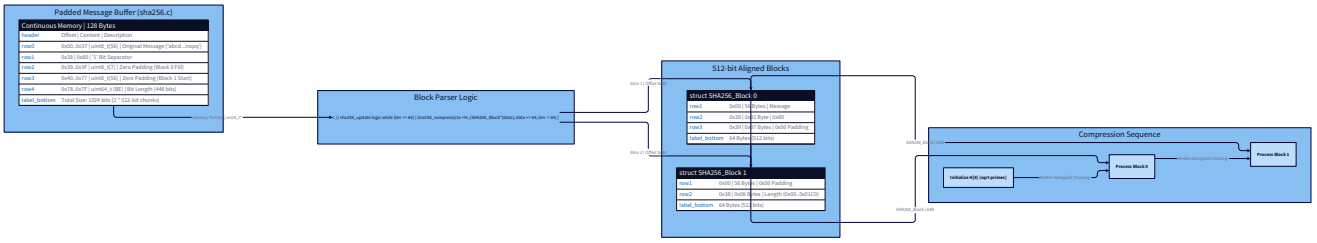
```
        *num_blocks = 0;
        return NULL;
    }
    /* Step 1: Copy the original message */
    if (input_len > 0) {
        memcpy(buf, input, input_len);
    }
    /* Step 2: Append the '1' bit as byte 0x80 */
    buf[input_len] = 0x80;
    /* Step 3: Zero padding is already in place (calloc zero-initializes) */
    /* Step 4: Write 64-bit big-endian message length IN BITS */
    uint64_t bit_length = (uint64_t)input_len * 8;
    uint8_t *len_field = buf + total_len - 8;
    len_field[0] = (uint8_t)(bit_length >> 56);
    len_field[1] = (uint8_t)(bit_length >> 48);
    len_field[2] = (uint8_t)(bit_length >> 40);
    len_field[3] = (uint8_t)(bit_length >> 32);
    len_field[4] = (uint8_t)(bit_length >> 24);
    len_field[5] = (uint8_t)(bit_length >> 16);
    len_field[6] = (uint8_t)(bit_length >>  8);
    len_field[7] = (uint8_t)(bit_length >>  0);
    /* Cast the buffer to our block array type — sizes match exactly */
    return (SHA256_Block *)buf;
}
```

Notice several deliberate choices in this code:

1. **`calloc` instead of `malloc`**: `calloc` zero-initializes the allocation. This means the zero-padding bytes between `0x80` and the length field are automatically correct — you don't need a separate `memset`. This is not just convenience; it prevents bugs where you forget to zero the padding region explicitly.
2. **The modular arithmetic for `zero_pad`**: The formula `(64 - ((input_len + 9) % 64)) % 64` correctly handles both the normal case and the case where `input_len + 9` is already a multiple of 64 (the outer `% 64` makes the result 0 in that case, rather than 64).
3. **`bit_length` as `uint64_t`**: The length field stores the bit count. For a 1 GB message, the byte count fits in a 32-bit integer but the bit count (×8) requires a 64-bit value. Cast before multiplying.
4. **The length field written explicitly byte-by-byte**: No `memcpy` from a uint64_t local variable — that would be endianness-dependent. The explicit shifts are correct on every platform.

Parsing into Blocks

The `SHA256_Block *` returned by `sha256_pad` is already the block array — the cast in the last line of the function does the work. But let's make the block-parsing step explicit to build understanding:




```c
/*
 * Print a visual dump of a padded block (useful for debugging)
 * Shows the hex content of each byte
 */
void dump_block(const SHA256_Block *block, size_t block_index) {
    printf("Block %zu:\n", block_index);
    for (int i = 0; i < SHA256_BLOCK_SIZE; i++) {
        printf("%02x ", block->bytes[i]);
        if ((i + 1) % 16 == 0) printf("\n");
    }
    printf("\n");
}
/*
 * Example of how the compression stage will iterate:
 * (This is the calling convention — full compression comes in Milestone 3)
 */
void process_all_blocks(const uint8_t *message, size_t message_len) {
    size_t num_blocks;
    SHA256_Block *blocks = sha256_pad(message, message_len, &num_blocks);
    if (!blocks) {
        fprintf(stderr, "Allocation failed\n");
        return;
    }
    for (size_t i = 0; i < num_blocks; i++) {
        /* Each blocks[i] is a SHA256_Block — exactly 64 bytes */
        dump_block(&blocks[i], i);
        /* In Milestone 3, you'll call: sha256_compress(state, &blocks[i]); */
    }
    free(blocks);   /* Always free after use */
}
```

The block structure cleanly encapsulates the 64 bytes. Downstream code (the message schedule generator in Milestone 2) will receive a `SHA256_Block *` and extract 32-bit words from it — we'll get to exactly

how that works then.

Testing Your Padding Function

This is the moment to write tests before touching anything else. The FIPS 180-4 appendix gives you ground truth for intermediate values, and your tests should check the padded bytes directly — not just the final hash — so you can isolate bugs to this layer.

```c
#include <stdio.h>

#include <string.h>

#include <assert.h>

/* Helper: check that two byte arrays are equal, print diagnostic on failure */

static void assert_bytes_equal(const uint8_t *got, const uint8_t *expected,

                               size_t len, const char *test_name) {

    for (size_t i = 0; i < len; i++) {

        if (got[i] != expected[i]) {

            printf("FAIL [%s] at byte %zu: got 0x%02x, expected 0x%02x\n",

                   test_name, i, got[i], expected[i]);

            return;

        }

    }

    printf("PASS [%s]\n", test_name);

}

void test_empty_input(void) {

    size_t num_blocks;

    SHA256_Block *blocks = sha256_pad(NULL, 0, &num_blocks);

    assert(blocks != NULL);

    assert(num_blocks == 1);  /* Empty input → exactly 1 block */

    /* Byte 0 must be 0x80 */

    assert(blocks[0].bytes[0] == 0x80);

    /* Bytes 1–55 must be 0x00 */

    for (int i = 1; i <= 55; i++) {

        assert(blocks[0].bytes[i] == 0x00);

    }

    /* Bytes 56–63 must be 0x00 (length = 0 bits) */

    for (int i = 56; i <= 63; i++) {

        assert(blocks[0].bytes[i] == 0x00);

    }

    printf("PASS [test_empty_input]\n");

    free(blocks);

}

void test_abc(void) {

    /* "abc" = 0x61 0x62 0x63, 24 bits */

    const uint8_t msg[] = {0x61, 0x62, 0x63};

    size_t num_blocks;

    SHA256_Block *blocks = sha256_pad(msg, 3, &num_blocks);

    assert(blocks != NULL);

    assert(num_blocks == 1);  /* 3 bytes → 1 block */
```

```c
    /* Check message bytes */
    assert(blocks[0].bytes[0] == 0x61);
    assert(blocks[0].bytes[1] == 0x62);
    assert(blocks[0].bytes[2] == 0x63);
    /* Byte 3 must be 0x80 */
    assert(blocks[0].bytes[3] == 0x80);
    /* Bytes 4–55 must be 0x00 */
    for (int i = 4; i <= 55; i++) {
        assert(blocks[0].bytes[i] == 0x00);
    }
    /*
     * Bytes 56–63: 64-bit big-endian length = 24 (= 3 bytes × 8 bits)
     * = 0x0000000000000018
     */
    assert(blocks[0].bytes[56] == 0x00);
    assert(blocks[0].bytes[57] == 0x00);
    assert(blocks[0].bytes[58] == 0x00);
    assert(blocks[0].bytes[59] == 0x00);
    assert(blocks[0].bytes[60] == 0x00);
    assert(blocks[0].bytes[61] == 0x00);
    assert(blocks[0].bytes[62] == 0x00);
    assert(blocks[0].bytes[63] == 0x18);  /* 24 decimal = 0x18 */
    printf("PASS [test_abc]\n");
    free(blocks);
}
void test_55_byte_boundary(void) {
    /* 55 bytes → should produce exactly 1 block (the maximum for 1-block messages) */
    uint8_t msg[55];
    memset(msg, 0xAA, 55);  /* Fill with 0xAA so we can see the separator clearly */
    size_t num_blocks;
    SHA256_Block *blocks = sha256_pad(msg, 55, &num_blocks);
    assert(blocks != NULL);
    assert(num_blocks == 1);
    /* byte 55 = 0x80 (immediately after message) */
    assert(blocks[0].bytes[55] == 0x80);
    /* No zero padding bytes between 0x80 and length (55 + 1 + 8 = 64) */
    /* Length = 55 × 8 = 440 bits = 0x00000000000001B8 */
    assert(blocks[0].bytes[56] == 0x00);
    assert(blocks[0].bytes[57] == 0x00);
    assert(blocks[0].bytes[58] == 0x00);
```

```c
    assert(blocks[0].bytes[59] == 0x00);
    assert(blocks[0].bytes[60] == 0x00);
    assert(blocks[0].bytes[61] == 0x00);
    assert(blocks[0].bytes[62] == 0x01);  /* 0x01B8 >> 8 = 0x01 */
    assert(blocks[0].bytes[63] == 0xB8);  /* 440 & 0xFF = 0xB8 */
    printf("PASS [test_55_byte_boundary]\n");
    free(blocks);
}
void test_56_byte_boundary(void) {
    /* 56 bytes → must produce 2 blocks (the critical boundary case) */
    uint8_t msg[56];
    memset(msg, 0xBB, 56);
    size_t num_blocks;
    SHA256_Block *blocks = sha256_pad(msg, 56, &num_blocks);
    assert(blocks != NULL);
    assert(num_blocks == 2);  /* ← This is the key assertion */
    /* In block 1, byte 56 should be 0x80 */
    assert(blocks[0].bytes[56] == 0x80);
    /* Bytes 57–63 in block 1 should be 0x00 (zero padding continues) */
    for (int i = 57; i <= 63; i++) {
        assert(blocks[0].bytes[i] == 0x00);
    }
    /* All of block 2 bytes 0–55 are zero padding */
    for (int i = 0; i <= 55; i++) {
        assert(blocks[1].bytes[i] == 0x00);
    }
    /* Length field in block 2, bytes 56–63 = 56 × 8 = 448 = 0x00000000000001C0 */
    assert(blocks[1].bytes[62] == 0x01);
    assert(blocks[1].bytes[63] == 0xC0);  /* 448 & 0xFF = 0xC0 */
    printf("PASS [test_56_byte_boundary]\n");
    free(blocks);
}
int main(void) {
    test_empty_input();
    test_abc();
    test_55_byte_boundary();
    test_56_byte_boundary();
    printf("\nAll padding tests passed.\n");
    return 0;
```

```
}
```

Compile and run with:

```
gcc -Wall -Wextra -o test_padding test_padding.c && ./test_padding
```
BASH

All four tests must pass before you proceed to Milestone 2. If `test_56_byte_boundary` fails with `num_blocks == 1`, you have the classic boundary bug — go back to your `zero_pad` formula.

What the Adversary Sees Here

From a security perspective, the padding scheme is not just a formatting convenience — it closes a real attack surface. If the length field were omitted, an attacker could construct two different messages that pad to identical block sequences. Worse, without the length commitment, SHA-256's Merkle-Damgård structure would be directly vulnerable to length extension attacks. Here's the shape of that attack: given `SHA-256(secret || message)` and the length of `secret || message` (but not `secret` itself), an attacker can compute `SHA-256(secret || message || padding || extension)` for any `extension` of their choice — without knowing `secret`. The attack works because the attacker knows the exact internal state of SHA-256 after processing the padded original message, and can continue feeding blocks from there. The length field in the padding doesn't prevent length extension (that requires HMAC or SHA-3). What it does do is make the padding scheme injective — no two distinct messages produce the same padded bit string, so padding itself doesn't create collisions. The broader length-extension vulnerability in Merkle-Damgård is why protocols that need a MAC use HMAC (which wraps SHA-256 in a specific double-hash construction) rather than raw SHA-256.

Summary: What Your Padding Function Guarantees

After completing this milestone, your `sha256_pad` function enforces these invariants:

1. **Total padded length is always a multiple of 512 bits (64 bytes).**
2. **Byte immediately following the message is always `0x80`.**
3. **Zero-padding fills the gap between `0x80` and the length field.**
4. **Last 8 bytes of the last block encode the original message length in bits, big-endian.**
5. **A message of ≤ 55 bytes produces exactly 1 block; ≥ 56 bytes produces ≥ 2 blocks.**
6. **Empty input produces exactly 1 block with `0x80` at byte 0 and a zero length field.** These are not negotiable. The SHA-256 specification has no "close enough" — every downstream computation (message schedule generation, compression) depends on these bytes being exactly correct.

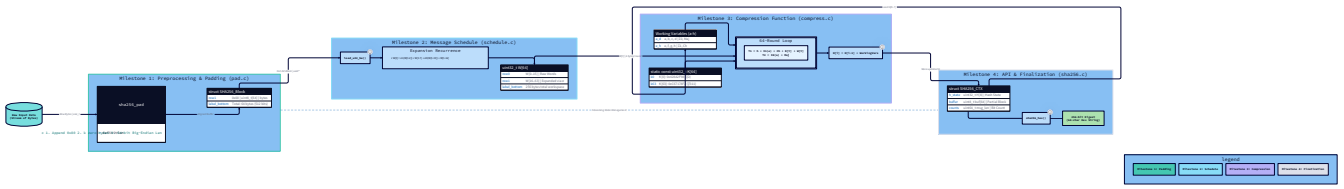
Knowledge Cascade: What This Unlocks

Understanding SHA-256 padding precisely connects you to a web of concepts that appear throughout cryptography and systems design: 1. Block Cipher Padding (PKCS#7, ISO 7816-4) The same "unambiguous padding" principle appears in AES-CBC mode. PKCS#7 padding appends `N` bytes each with value `N` (so 3 bytes of padding looks like `03 03 03`). The reason for this is identical to SHA-256's '1' bit: you need to be able to remove padding unambiguously after decryption. When this padding scheme is implemented incorrectly or the error messages are too informative, it enables padding oracle attacks — one of the most devastating and elegant attacks in practical cryptography, where an attacker decrypts ciphertext one byte at a time by observing whether the server accepts or rejects padding. The principle

you learned here — "padding must be self-describing and unambiguous" — is what PKCS#7 violated in dozens of real-world implementations. 2. Length Extension Attacks and HMAC As noted above, the Merkle-Damgård structure SHA-256 uses means that knowing a SHA-256 hash gives an attacker the internal state *after* processing the padded message. They can append new blocks and compute a valid hash. This is why `SHA-256(key || message)` is broken as a MAC (message authentication code), and why HMAC uses `SHA-256(key || SHA-256(key || message))` with specific padding — the inner and outer hashes prevent state reconstruction. Your padding work directly informs understanding of why HMAC is designed the way it is. 3. TLV Encoding in Network Protocols TLS, Protocol Buffers, DNS, and ASN.1 all use Type-Length-Value (TLV) encoding: every data field is prefixed with its type tag and length. The motivation is identical — a receiver needs to know where one field ends and the next begins without ambiguity. SHA-256's length field is essentially a value-level TLV suffix: the "length" field tells the decoder (the compression function) exactly how many meaningful bits preceded the padding. When you next read a TLS record or a protobuf message on the wire, you'll recognize the same design principle. 4. Endianness as a Real Bug Source The big-endian requirement in SHA-256 is where real-world implementation bugs live. The infamous Heartbleed vulnerability (CVE-2014-0160) in OpenSSL was caused by a missing bounds check in a length field, not by endianness — but endianness-related bugs in cryptographic code have caused real vulnerabilities, particularly in implementations that use `memcpy` from a native integer directly into a byte buffer, getting the byte order wrong. Your explicit byte-by-byte encoding approach (shifting and casting) is the portable, correct pattern used in production cryptographic libraries.

Milestone 2: Message Schedule Generation

Where We Are



In Milestone 1, you built the front gate: your `sha256_pad` function takes arbitrary-length input and manufactures precise, unambiguous 512-bit blocks. Those blocks now sit in memory, perfectly formatted, waiting to be consumed. In this milestone, you're building the first stage of the engine itself. For every 512-bit block your padding function produces, the message schedule generator does two things:

1. **Parses** the 64-byte block into 16 big-endian 32-bit words: `W[0]` through `W[15]`.
2. **Expands** those 16 words into 64 words: `W[16]` through `W[63]`, using a recurrence relation driven by two bitwise mixing functions called σ0 ("sigma-zero") and σ1 ("sigma-one"). The compression function in Milestone 3 will consume one word per round, for 64 rounds. Your schedule generator is what makes that possible — it's the supply chain that feeds the compression engine.

The Misconception Worth Shattering

Here's what most developers assume when they first read the SHA-256 spec and see `W[0]..W[63]`:

> *"W[0] through W[15] are the real input — the actual message bits. W[16] through W[63] are just... padding? A way to fill the remaining 48 round slots? Maybe some kind of key-stretching? I'll knock this out with a loop."* That mental model will cause you to write a schedule generator that compiles, runs, and produces complete-looking output — for the completely wrong hash. Here's what's actually happening: **The message schedule is the diffusion engine of SHA-256.** Consider what would happen if you *skipped* the expansion entirely and just fed `W[0]..W[15]` repeatedly through 64 rounds of compression (cycling back to `W[0]` after `W[15]`). Each of the 16 original words would appear 4 times in 64 rounds. A single-bit change in `W[0]` would affect exactly the 4 rounds where `W[0]` appears. The other 60 rounds? Completely unaffected by that bit. That's catastrophically weak. An attacker can construct two different messages that differ in one carefully chosen bit, predict exactly which 4 rounds are affected, and potentially engineer a collision. With the expansion, the picture is completely different. Every word `W[t]` for `t ≥ 16` depends on `W[t-2]`, `W[t-7]`, `W[t-15]`, and `W[t-16]`. A one-bit change in `W[0]` ripples into `W[16]` (via `W[t-16]` term), then into `W[17]`, `W[18]`, and so on — spreading across the entire 64-word schedule. By the time you reach `W[63]`, every word has been touched by the original perturbation. **This is the avalanche effect at the schedule level: flip one input bit, scramble all 64 round inputs.** The σ functions aren't bookkeeping. They're the mechanism that makes this cascade happen irreversibly.

The Real Distinction: ROTR vs. SHR

Before writing a single line of the schedule generator, you need to understand the most dangerous confusion in this milestone. It's so common and so subtle that it deserves its own section. {{DIAGRAM:diag-m2-rotr-vs-shr}} Both operations "move bits to the right." They look almost identical. The difference is what happens to the bits that fall off the right edge. **Right-Shift (SHR)**: Bits that fall off the right are **destroyed**. Zero bits are fed in from the left. This is what C's `>>` operator does on unsigned integers.

```c
uint32_t x = 0b10110001000000000000000000000001;
//           = 0xB1000001
uint32_t result = x >> 3;
//           = 0b00010110001000000000000000000000
//           = 0x16200000
// The three rightmost bits (001) are GONE. Three zero bits entered from the left.
```

Right-Rotate (ROTR): Bits that fall off the right **wrap around** to the left. No information is lost.

```c
// ROTR(x, 3):
uint32_t x = 0b10110001000000000000000000000001;
//           = 0xB1000001
// The three bits that fall off the right: 001
// They re-enter from the left: 001...
uint32_t result = (x >> 3) | (x << (32 - 3));
//           = 0b00110110001000000000000000000000 | (001 << 29)
//           = 0b00110110001000000000000000000000
//           | 0b00100000000000000000000000000000
//           = 0b00110110001000000000000000000000
// Wait, let me trace this correctly:
// x >> 3  = 0b00010110001000000000000000000000
// x << 29 = 0b00100000000000000000000000000000
//                                    ^^^
//               The three LSBs of x (001) moved to bits 31,30,29
// result  = 0b00110110001000000000000000000000
```

Let me make this concrete with small numbers so you can trace it by hand:

```
x = 0x000000B0  (in binary: 00000000 00000000 00000000 10110000)
SHR(x, 4):
  Shift right 4 places, zero fill on left:
  = 00000000 00000000 00000000 00001011
  = 0x0000000B
  The bits 0000 that were the rightmost 4 bits of x? Gone.
ROTR(x, 4):
  The rightmost 4 bits of x (0000) wrap to the leftmost 4 bits:
  = 00000000 00000000 00000000 00001011  (the SHR part)
  | 00000000 00000000 00000000 00000000  (the wrapped bits — they were 0000)
  = 0x0000000B  (same here because the wrapped bits were zero)
Different example:
x = 0x000000B7  (binary: 00000000 00000000 00000000 10110111)
SHR(x, 4):
  = 00000000 00000000 00000000 00001011
  = 0x0000000B
  The four rightmost bits (0111) are destroyed.
ROTR(x, 4):
  = 00000000 00000000 00000000 00001011  (SHR part)
  | 01110000 00000000 00000000 00000000  (0111 wrapped to bits 31-28)
  = 01110000 00000000 00000000 00001011
  = 0x7000000B
```

The key insight: ROTR is an invertible operation — you can always undo it (rotate back). SHR is irreversible — once bits fall off the right, they're gone. The σ functions intentionally combine both: ROTR provides the invertible diffusion (bits spread but survive), while SHR introduces selective information destruction (making the overall function irreversible and harder to invert). This combination is what gives the message schedule its one-way, diffusing character. If you use `>>` (SHR) where the spec says ROTR, you destroy information that should be preserved. The result *looks* like a valid 32-bit number. The final hash will be a perfectly formatted 64-character hexadecimal string — just the completely wrong one. No crash, no warning, no indication something is wrong except that the test vectors fail.

Implementing ROTR in C

C has no built-in rotate instruction (even though every CPU has one). The idiom is:

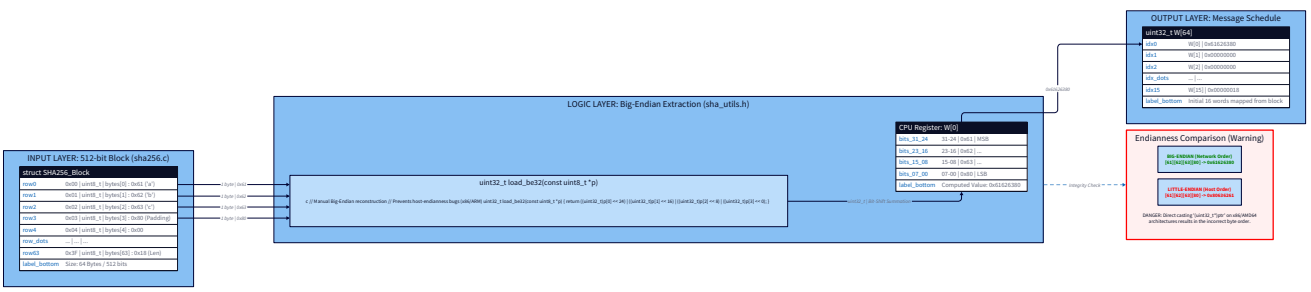
```c
/* Right-rotate a 32-bit unsigned integer by n positions */

static inline uint32_t rotr32(uint32_t x, unsigned int n) {

    return (x >> n) | (x << (32 - n));

}
```

This is the pattern used universally in cryptographic C code — in OpenSSL, in BoringSSL, in libressl. Modern compilers (GCC, Clang) recognize this pattern and emit a single `ROR` instruction on x86 and `ROR` on ARM. You're not writing a slow software loop; you're giving the compiler the information it needs to use the hardware rotate instruction.

> ⚠️ **The `n = 0` edge case**: If `n` is 0 or 32, the expression `x << (32 - n)` becomes `x << 32`, which is **undefined behavior** in C (shifting a 32-bit value by its own width). In practice this never happens in SHA-256 (the rotation constants are 7, 17, 18, 19 — never 0 or 32), but it's worth knowing. If you ever generalize this to variable rotations, guard against it.

Parsing a Block into 32-bit Words

Your `sha256_pad` function returns an array of `SHA256_Block` structs, each containing 64 bytes. Before the message schedule can begin, those 64 bytes must be interpreted as 16 big-endian 32-bit unsigned integers.



Consider the first four bytes of a block: `0x61`, `0x62`, `0x63`, `0x80` (the start of the padded "abc" message). In big-endian, these four bytes form the 32-bit word:

```
byte[0]  byte[1]  byte[2]  byte[3]
  0x61     0x62     0x63     0x80
Big-endian 32-bit word:
  = (0x61 << 24) | (0x62 << 16) | (0x63 << 8) | 0x80
  = 0x61626380
```

The byte with the **largest positional weight** (the "big" end) comes first in memory. This is the big-endian convention SHA-256 mandates everywhere. You cannot just cast a `uint8_t *` to a `uint32_t *` and read it — that would give you little-endian ordering on x86 hardware, silently producing wrong values. You must reconstruct each word byte-by-byte with explicit shifts:

```c
/*
 * load_u32_be - Load a big-endian 32-bit word from 4 consecutive bytes
 *
 * p: pointer to the first (most significant) byte
 * Returns the 32-bit unsigned integer in native (host) byte order
 */
static inline uint32_t load_u32_be(const uint8_t *p) {
    return ((uint32_t)p[0] << 24)
         | ((uint32_t)p[1] << 16)
         | ((uint32_t)p[2] <<  8)
         | ((uint32_t)p[3] <<  0);
}
```

The cast to `(uint32_t)` before shifting is critical. Without it, on a system where `uint8_t` is promoted to `int` (32-bit signed), shifting by 24 positions with a value like `0x80` would shift a negative `int` — undefined behavior in C. The cast ensures the value is treated as unsigned before shifting. With this helper, parsing a block into its 16 initial words is clean:

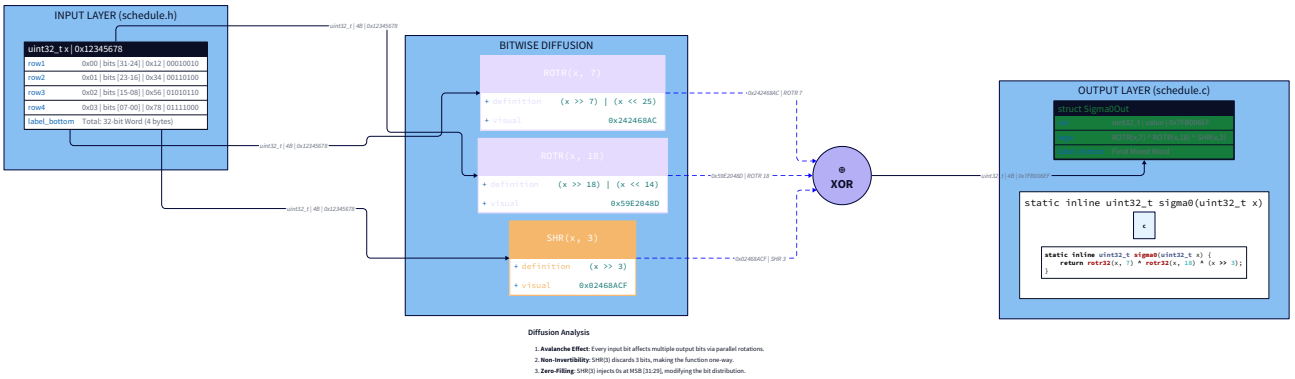
```c
/*
 * sha256_schedule_init - Parse a 512-bit block into W[0]..W[15]
 *
 * block: the 64-byte block to parse
 * W:     array of at least 64 uint32_t values; W[0]..W[15] are set here
 */
static void sha256_schedule_init(const SHA256_Block *block, uint32_t W[64]) {
    for (int t = 0; t < 16; t++) {
        W[t] = load_u32_be(block->bytes + t * 4);
    }
}
```

Sixteen words, sixteen iterations. Each word occupies 4 consecutive bytes at offset `t * 4`. This is unambiguous: `W[0]` comes from bytes 0–3, `W[1]` from bytes 4–7, ..., `W[15]` from bytes 60–63.

The σ Functions: Anatomy of Diffusion

The expansion from 16 to 64 words is driven by two lowercase σ functions. They look like this in the FIPS 180-4 specification:

```
σ0(x) = ROTR(x, 7) XOR ROTR(x, 18) XOR SHR(x, 3)
σ1(x) = ROTR(x, 17) XOR ROTR(x, 19) XOR SHR(x, 10)
```



Let's understand what each piece contributes before implementing anything. **ROTR(x, n)**: Takes the 32-bit word and rotates it — bits wrap around. Using *two* different rotation amounts (7 and 18 for σ0, or 17 and 19 for σ1) and XORing them together means no single bit in the output is determined by just one bit of the input. Each output bit is the XOR of two input bits from different positions. This is linear diffusion. **SHR(x, n)**: Shifts the word, destroying the bits that fall off. The XOR with SHR(x,3) means the three bits that "fall off" in the SHR are not compensated by wrap-around — they genuinely vanish from the computation. This introduces **non-invertibility**: given σ0(x), you cannot uniquely determine x. The SHR terms are what make these functions one-way in character (though they're linear — true non-linearity comes from the Ch and Maj functions in the compression function). **Why these specific constants?** The constants (7, 18, 3) for σ0 and (17, 19, 10) for σ1 were chosen by the SHA-256 designers to maximize the diffusion properties of the full message schedule. The rotation amounts are chosen so that the combined effect of W[t-2], W[t-7], W[t-15], W[t-16] (each processed by σ functions) achieves full diffusion in the minimum number of expansion steps. You don't need to derive them — just know they're not arbitrary, and substituting different constants would weaken the hash.

> ⚠️ **The #1 constant-confusion bug**: σ0 uses **(7, 18, 3)** and σ1 uses **(17, 19, 10)**. These are easy to swap because they look similar. The FIPS 180-4 spec lists them clearly, but under time pressure, developers sometimes write σ0 with the σ1 constants or vice versa. Your known-answer tests will catch this — which is exactly why you write them before trusting your implementation.

Implementing σ0 and σ1 in C

```c
/*
 * sigma0 - SHA-256 lowercase sigma-0 (used in message schedule)
 * σ0(x) = ROTR(x, 7) XOR ROTR(x, 18) XOR SHR(x, 3)
 */
static inline uint32_t sigma0(uint32_t x) {
    return rotr32(x, 7) ^ rotr32(x, 18) ^ (x >> 3);
}
/*
 * sigma1 - SHA-256 lowercase sigma-1 (used in message schedule)
 * σ1(x) = ROTR(x, 17) XOR ROTR(x, 19) XOR SHR(x, 10)
 */
static inline uint32_t sigma1(uint32_t x) {
    return rotr32(x, 17) ^ rotr32(x, 19) ^ (x >> 10);
}
```

These are `static inline` because they're simple enough to inline and used in a hot loop (64 iterations per block). The `^` operator is C's bitwise XOR. The `>>` is the right-shift (SHR) — intentionally not `rotr32` for this operand.

> **Naming note**: These are the **lowercase** σ functions, used only in the message schedule. The compression function (Milestone 3) uses **uppercase** Σ (Sigma) functions, which have different rotation constants (2, 13, 22 for Σ0 and 6, 11, 25 for Σ1). They look visually similar in code if you name them carelessly. Use `sigma0` / `sigma1` (lowercase) for schedule functions and `Sigma0` / `Sigma1` (uppercase) for compression functions to prevent confusion that costs hours of debugging.

Validating σ Functions with Known-Answer Tests

Before expanding the full schedule, validate your σ functions against the NIST SHA-256 example computation from FIPS 180-4 Appendix B.1 (the "abc" test vector). The NIST document provides intermediate values for the first few words of the message schedule for "abc". For the "abc" test vector, after parsing the padded block into W[0]..W[15]:

```
W[ 0] = 0x61626380   (bytes 'a' 'b' 'c' 0x80)
W[ 1] = 0x00000000
W[ 2] = 0x00000000
W[ 3] = 0x00000000
W[ 4] = 0x00000000
W[ 5] = 0x00000000
W[ 6] = 0x00000000
W[ 7] = 0x00000000
W[ 8] = 0x00000000
W[ 9] = 0x00000000
W[10] = 0x00000000
W[11] = 0x00000000
W[12] = 0x00000000
W[13] = 0x00000000
W[14] = 0x00000000
W[15] = 0x00000018   (24 in decimal — the bit length of "abc")
```

Let's compute W[16] manually to verify your σ implementation:

```
W[16] = σ1(W[14]) + W[9] + σ0(W[1]) + W[0]
      = σ1(0x00000000) + 0x00000000 + σ0(0x00000000) + 0x61626380
σ1(0x00000000):
  ROTR(0, 17) = 0
  ROTR(0, 19) = 0
  SHR (0, 10) = 0
  σ1(0) = 0 XOR 0 XOR 0 = 0x00000000
σ0(0x00000000):
  ROTR(0, 7)  = 0
  ROTR(0, 18) = 0
  SHR (0, 3)  = 0
  σ0(0) = 0 XOR 0 XOR 0 = 0x00000000
W[16] = 0x00000000 + 0x00000000 + 0x00000000 + 0x61626380
      = 0x61626380
```

The NIST appendix confirms W[16] = 0x61626380. ✓ Let's also compute W[19] to test σ0 and σ1 on a non-zero input:

```
W[19] = σ1(W[17]) + W[12] + σ0(W[4]) + W[3]
W[17] = σ1(W[15]) + W[10] + σ0(W[2]) + W[1]
      = σ1(0x00000018) + 0 + σ0(0) + 0
σ1(0x00000018):
  x    = 0x00000018 = 0b00000000_00000000_00000000_00011000
  ROTR(x, 17):  rotate right 17 bits
    x >> 17    = 0b00000000_00000000_00000000_00000000  (0x00000000, the bits are too small)
    Actually: 0x00000018 = 24. 24 >> 17 = 0 (since 24 < 2^17 = 131072)
    x << 15    = 0x00000018 << 15 = 0x000C0000
    ROTR(x,17) = 0x00000000 | 0x000C0000 = 0x000C0000
  ROTR(x, 19):
    x >> 19    = 0 (same reasoning)
    x << 13    = 0x00000018 << 13 = 0x00030000
    ROTR(x,19) = 0x00030000
  SHR(x, 10):
    0x00000018 >> 10 = 0 (24 >> 10 = 0)
  σ1(0x00000018) = 0x000C0000 XOR 0x00030000 XOR 0x00000000
                 = 0x000F0000
W[17] = 0x000F0000 + 0 + 0 + 0 = 0x000F0000
```

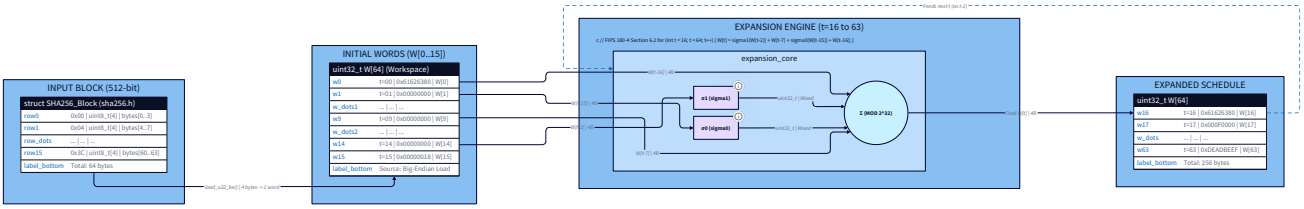
The NIST appendix confirms W[17] = 0x000F0000. ✓ Write a C test for this before moving on:

```c
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
/* Assumes rotr32, sigma0, sigma1 are defined as above */
void test_sigma_functions(void) {
    /* Test σ1 on 0x00000018 (from the NIST "abc" example) */
    uint32_t result = sigma1(0x00000018);
    if (result != 0x000F0000) {
        printf("FAIL [sigma1(0x00000018)]: got 0x%08X, expected 0x000F0000\n", result);
        return;
    }
    printf("PASS [sigma1(0x00000018) = 0x000F0000]\n");
    /* Test σ0 on 0x00000000 (trivial case) */
    assert(sigma0(0x00000000) == 0x00000000);
    printf("PASS [sigma0(0x00000000) = 0x00000000]\n");
    /* Test σ1 on 0x00000000 (trivial case) */
    assert(sigma1(0x00000000) == 0x00000000);
    printf("PASS [sigma1(0x00000000) = 0x00000000]\n");
    /* Test rotr32 with a known value */
    /* ROTR(0xB1000001, 4) should move the bottom 4 bits (0001) to bits 31-28 */
    uint32_t r = rotr32(0xB1000001, 4);
    /* 0xB1000001 >> 4 = 0x0B100000 */
    /* 0xB1000001 << 28 = 0x10000000 */
    /* result = 0x0B100000 | 0x10000000 = 0x1B100000 */
    if (r != 0x1B100000) {
        printf("FAIL [rotr32(0xB1000001, 4)]: got 0x%08X, expected 0x1B100000\n", r);
        return;
    }
    printf("PASS [rotr32(0xB1000001, 4) = 0x1B100000]\n");
}
```

The Recurrence Relation: Expanding to 64 Words

With your σ functions validated, you can implement the expansion. The recurrence relation from FIPS 180-4 Section 6.2:

```
For t = 16 to 63:
    W[t] = σ1(W[t-2]) + W[t-7] + σ0(W[t-15]) + W[t-16]
    (addition is mod 2^32)
```



Let's unpack the subscript choices. Why `t-2`, `t-7`, `t-15`, and `t-16` — not `t-1`, `t-4`, `t-8`, `t-16`? The SHA-256 designers chose these offsets to maximize the rate at which each input word's influence spreads to all future words. The combination of σ1 applied to the near word (`t-2`) and σ0 applied to the far word (`t-15`) ensures that both recent history and older history continuously shape each new word. The exact values came from systematic analysis of diffusion properties. The addition is **modular 2^32** — the mathematical term for "let it overflow and wrap around." In C, this is automatic with `uint32_t`:

```c
uint32_t a = 0xFFFFFFFF;

uint32_t b = 0x00000001;

uint32_t c = a + b;  // c = 0x00000000 — overflow is automatic, no masking needed
```

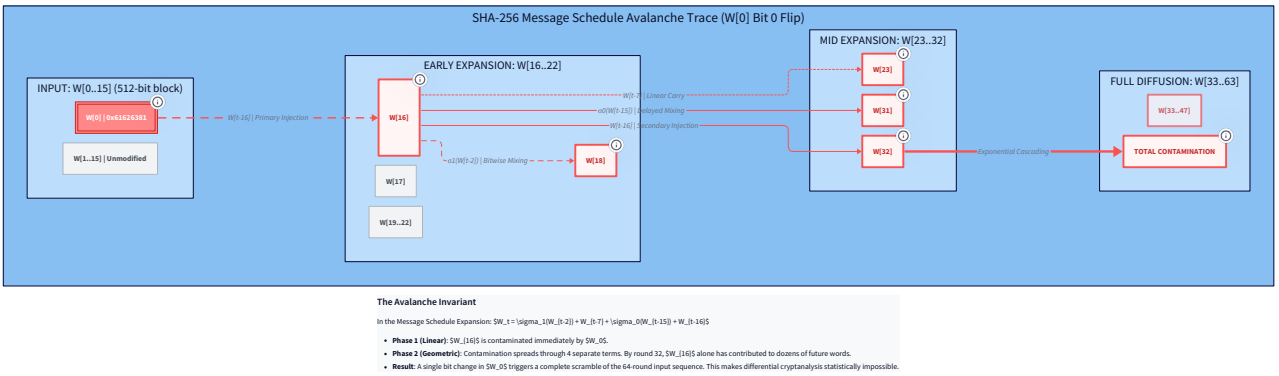
This is one of C's direct advantages for implementing SHA-256: `uint32_t` arithmetic wraps exactly as the spec requires, with no extra work. In Python or JavaScript, you'd need `& 0xFFFFFFFF` after every addition — in C, `uint32_t` does it for free. Here's the complete schedule expansion function:

```c
/*
 * sha256_schedule_expand - Expand W[0]..W[15] to W[0]..W[63]
 *
 * W: array of 64 uint32_t values; W[0]..W[15] must be pre-populated
 *    by sha256_schedule_init. This function sets W[16]..W[63].
 */
static void sha256_schedule_expand(uint32_t W[64]) {
    for (int t = 16; t < 64; t++) {
        W[t] = sigma1(W[t - 2])
             + W[t - 7]
             + sigma0(W[t - 15])
             + W[t - 16];
        /*
         * In C with uint32_t, the addition wraps automatically at 2^32.
         * No masking (& 0xFFFFFFFF) needed here — this is C, not Python.
         */
    }
}
```

And the complete schedule generation — init plus expand — wrapped in a single function:

```c
/*
 * sha256_message_schedule - Generate the full 64-word message schedule
 * from a single 512-bit block.
 *
 * block: the 64-byte input block
 * W:     output array of 64 uint32_t — the complete message schedule
 */
void sha256_message_schedule(const SHA256_Block *block, uint32_t W[64]) {
    sha256_schedule_init(block, W);
    sha256_schedule_expand(W);
}
```

Tracing the Complete "abc" Schedule



Let's verify the first few words of the "abc" schedule against the NIST appendix. You should validate these values by running your code, not just trusting this document:

```
W[ 0] = 0x61626380  W[ 1] = 0x00000000  W[ 2] = 0x00000000  W[ 3] = 0x00000000
W[ 4] = 0x00000000  W[ 5] = 0x00000000  W[ 6] = 0x00000000  W[ 7] = 0x00000000
W[ 8] = 0x00000000  W[ 9] = 0x00000000  W[10] = 0x00000000  W[11] = 0x00000000
W[12] = 0x00000000  W[13] = 0x00000000  W[14] = 0x00000000  W[15] = 0x00000018
W[16] = 0x61626380  W[17] = 0x000F0000  W[18] = 0x61706380  W[19] = 0x610E0000
W[20] = 0x00000000  W[21] = 0x00000000  W[22] = 0x00000006  W[23] = 0x00000000
W[24] = 0x00000000  W[25] = 0x00000000  W[26] = 0x00000000  W[27] = 0x00000000
W[28] = 0x00000000  W[29] = 0x00000000  W[30] = 0x00000000  W[31] = 0x00000000
```

Notice how W[16] through W[19] all contain non-zero values derived from `W[0] = 0x61626380`, while W[20] through W[31] are still zero. This is the diffusion wave beginning to propagate. By W[48], virtually every word will contain non-zero values derived from both `W[0]` and `W[15]`.

Testing the Full Schedule

Write a test that generates the complete schedule for "abc" and compares W[16] and W[17] to the NIST-provided values. These are the two values you can derive by hand and are the most useful regression anchors:

```c
#include <stdio.h>

#include <string.h>

#include <stdint.h>

/* Assumes sha256_pad and sha256_message_schedule are available */

void test_schedule_abc(void) {

    const uint8_t abc[] = {0x61, 0x62, 0x63};

    size_t num_blocks;

    SHA256_Block *blocks = sha256_pad(abc, 3, &num_blocks);

    if (!blocks) {

        printf("FAIL [test_schedule_abc]: allocation failed\n");

        return;

    }

    if (num_blocks != 1) {

        printf("FAIL [test_schedule_abc]: expected 1 block, got %zu\n", num_blocks);

        free(blocks);

        return;

    }

    uint32_t W[64];

    sha256_message_schedule(&blocks[0], W);

    /* Verify W[0]..W[15] parsed correctly */

    if (W[0] != 0x61626380) {

        printf("FAIL [W[0]]: got 0x%08X, expected 0x61626380\n", W[0]);

        free(blocks);

        return;

    }

    if (W[15] != 0x00000018) {

        printf("FAIL [W[15]]: got 0x%08X, expected 0x00000018\n", W[15]);

        free(blocks);

        return;

    }

    /* Verify W[16] and W[17] from NIST Appendix B.1 */

    if (W[16] != 0x61626380) {

        printf("FAIL [W[16]]: got 0x%08X, expected 0x61626380\n", W[16]);

        free(blocks);

        return;

    }

    if (W[17] != 0x000F0000) {

        printf("FAIL [W[17]]: got 0x%08X, expected 0x000F0000\n", W[17]);

        free(blocks);

        return;
```

```
    }

    printf("PASS [test_schedule_abc]: W[0], W[15], W[16], W[17] correct\n");

    /* Print the full schedule for visual inspection */

    printf("\nFull message schedule for \"abc\":\n");

    for (int t = 0; t < 64; t++) {

        printf("W[%2d] = 0x%08X\n", t, W[t]);

    }

    free(blocks);

}

void test_schedule_empty(void) {

    size_t num_blocks;

    SHA256_Block *blocks = sha256_pad(NULL, 0, &num_blocks);

    if (!blocks || num_blocks != 1) {

        printf("FAIL [test_schedule_empty]: padding failed\n");

        return;

    }

    uint32_t W[64];

    sha256_message_schedule(&blocks[0], W);

    /* For empty input, W[0] = 0x80000000 (0x80 byte in MSB position) */

    if (W[0] != 0x80000000) {

        printf("FAIL [empty W[0]]: got 0x%08X, expected 0x80000000\n", W[0]);

        free(blocks);

        return;

    }

    /* W[1]..W[14] should all be 0x00000000 */

    for (int i = 1; i <= 14; i++) {

        if (W[i] != 0x00000000) {

            printf("FAIL [empty W[%d]]: got 0x%08X, expected 0x00000000\n", i, W[i]);

            free(blocks);

            return;

        }

    }

    /* W[15] = 0x00000000 (length = 0 bits) */

    if (W[15] != 0x00000000) {

        printf("FAIL [empty W[15]]: got 0x%08X, expected 0x00000000\n", W[15]);

        free(blocks);

        return;

    }

    printf("PASS [test_schedule_empty]: W[0] = 0x80000000, W[1..15] = 0\n");

    free(blocks);
```

```
}

int main(void) {

    test_sigma_functions();

    test_schedule_abc();

    test_schedule_empty();

    printf("\nAll message schedule tests passed.\n");

    return 0;

}
```

Compile and run:

```
gcc -Wall -Wextra -o test_schedule test_schedule.c && ./test_schedule
```
BASH

If `W[16] = 0x61626380` and `W[17] = 0x000F0000` pass, your schedule generator is correct for the "abc" case. The full SHA-256 hash output test in Milestone 4 will confirm the remaining 46 words are correct.

The Adversary Perspective: Why This Schedule Design Resists Attack

Let's think like an attacker for a moment. Your goal is to find two different messages M and M' such that SHA-256(M) = SHA-256(M') — a collision. Where would you focus your attack on the message schedule? Attack surface 1: Can you engineer messages with identical schedules? Two messages that produce the same `W[0]..W[63]` would produce the same hash (assuming the same initial state). The schedule expansion is a linear function — it XORs and adds words. Linear functions are more amenable to algebraic manipulation than nonlinear ones. However, the σ functions' combination of rotations at different offsets creates a diffusion matrix that, over 64 words, makes it computationally infeasible to find two distinct `W[0]..W[15]` inputs that produce identical `W[0]..W[63]` outputs. This is not a formal proof of collision resistance — SHA-256's full collision resistance relies on the compression function's nonlinear Ch and Maj operations — but the schedule itself is deliberately hard to invert. Attack surface 2: Can you make the schedule "boring" — constant or low-entropy? If an attacker could craft a message where many W values are identical or zero, the compression function would see low-entropy inputs for many rounds, potentially weakening the mixing. The NIST designers chose the σ constants to ensure that even a message block consisting entirely of zeros (except for the mandatory padding) generates a schedule with rapidly increasing entropy. Look at the "abc" schedule above: by W[19], there are already four distinct non-zero values spread across the 32-bit word space. Attack surface 3: Timing side channels? The message schedule has no branches and no data-dependent memory accesses — it's a pure arithmetic pipeline. There are no timing side channels in the schedule itself. The XOR, rotation, and addition operations take constant time regardless of input values on every modern CPU. This is in contrast to table-lookup-based operations (like AES S-boxes in some implementations) where cache timing can leak key bits. SHA-256's bitwise design is inherently constant-time.

Putting It Together: The Complete schedule.c

Here is the complete, self-contained schedule implementation you should have at the end of this milestone:

```c
/* schedule.c — SHA-256 message schedule generation */

#include <stdint.h>

#include <stddef.h>

#include <string.h>

/* SHA256_Block type — must match the definition from your padding implementation */

#define SHA256_BLOCK_SIZE 64

typedef struct {

    uint8_t bytes[SHA256_BLOCK_SIZE];

} SHA256_Block;

/* ──────────────────────────────────────────────── */

 * Primitive: right-rotate a 32-bit value by n positions

 * ──────────────────────────────────────────────── */

static inline uint32_t rotr32(uint32_t x, unsigned int n) {

    return (x >> n) | (x << (32u - n));

}

/* ──────────────────────────────────────────────── */

 * σ0: message schedule mixing function

 * σ0(x) = ROTR(x,7) XOR ROTR(x,18) XOR SHR(x,3)

 * ──────────────────────────────────────────────── */

static inline uint32_t sigma0(uint32_t x) {

    return rotr32(x, 7) ^ rotr32(x, 18) ^ (x >> 3);

}

/* ──────────────────────────────────────────────── */

 * σ1: message schedule mixing function

 * σ1(x) = ROTR(x,17) XOR ROTR(x,19) XOR SHR(x,10)

 * ──────────────────────────────────────────────── */

static inline uint32_t sigma1(uint32_t x) {

    return rotr32(x, 17) ^ rotr32(x, 19) ^ (x >> 10);

}

/* ──────────────────────────────────────────────── */

 * load_u32_be: load a big-endian 32-bit word from 4 bytes

 * ──────────────────────────────────────────────── */

static inline uint32_t load_u32_be(const uint8_t *p) {

    return ((uint32_t)p[0] << 24)

         | ((uint32_t)p[1] << 16)

         | ((uint32_t)p[2] <<  8)

         | ((uint32_t)p[3] <<  0);

}

/* ──────────────────────────────────────────────── */

 * sha256_message_schedule: generate W[0]..W[63] from a 512-bit block
```

```
 *
 * block: pointer to the 64-byte input block
 * W:     caller-allocated array of 64 uint32_t (output)
 * ──────────────────────────────────────────────── */
void sha256_message_schedule(const SHA256_Block *block, uint32_t W[64]) {
    /* Step 1: Parse block into W[0]..W[15] in big-endian order */
    for (int t = 0; t < 16; t++) {
        W[t] = load_u32_be(block->bytes + t * 4);
    }
    /* Step 2: Expand W[16]..W[63] using the recurrence relation */
    for (int t = 16; t < 64; t++) {
        W[t] = sigma1(W[t - 2])   /* σ1 applied to the near-past word  */
             + W[t - 7]           /* an older word, unmodified          */
             + sigma0(W[t - 15])  /* σ0 applied to a distant word       */
             + W[t - 16];         /* the oldest word in the window      */
        /* uint32_t wraps automatically at 2^32 — no masking needed in C */
    }
}
```

Three-Level View: The Message Schedule from Top to Bottom

To solidify your understanding, let's look at the message schedule from three different altitudes: Level 1 — Algorithm (What the spec says): Parse a 512-bit block into 16 big-endian words. Apply a recurrence to extend to 64 words using σ0 and σ1 mixing functions. Feed the result to the compression function one word per round. Level 2 — Mathematical Structure (Why it works): The schedule is a length-64 linear recurrence over GF(2^32) (the field of integers mod 2^32 with bitwise XOR as "addition" for the mixing terms, plus modular addition across the full recurrence). The specific offsets (2, 7, 15, 16) and σ constants (7, 18, 3, 17, 19, 10) were chosen to achieve full rank — meaning the 64×16 "expansion matrix" has no null vectors, ensuring distinct 16-word inputs always produce distinct 64-word schedules. Level 3 — Hardware (What the CPU actually does): The inner loop body compiles to approximately 20 instructions per iteration: two ROR instructions (for each ROTR), two SHR instructions, three XOR instructions, three ADD instructions. On a modern out-of-order CPU with instruction-level parallelism, several iterations execute simultaneously. The schedule generation for a single 512-bit block takes roughly 100–200 CPU cycles — around 40–80 nanoseconds on a modern processor. This is why software SHA-256 without hardware

acceleration (Intel SHA-NI extensions) can hash around 400–600 MB/s on a single core: the schedule loop is tight and predictable.

Knowledge Cascade: What the Message Schedule Unlocks

Understanding how SHA-256's message schedule works connects you to a web of related ideas across cryptography and computer science: 1. The Avalanche Effect — Why 1 Bit Flips ~50% of Output Bits The SHA-256 message schedule is the primary reason for SHA-256's avalanche property. If you change one bit in the input — say, flip bit 17 of W[0] — that change propagates into W[16] (via the W[t-16] term), then into W[17], W[18], and so on. By W[32], that single flipped bit has influenced roughly half the bits of the schedule words through a combination of XOR mixing (linear) and modular addition (introduces carry propagation, which is nonlinear at the bit level). By W[63], essentially every bit of every word has been touched. The compression function then amplifies this further through its nonlinear Ch and Maj functions. The result: flipping one bit of the input message flips on average ~128 of the 256 output bits — exactly as you'd expect from a random function. This is what makes SHA-256 usable as a random oracle in protocol design. 2. AES Key Schedule — Symmetric Cipher's Answer to the Same Problem AES (Advanced Encryption Standard) faces the same challenge SHA-256's message schedule solves: how do you turn a single short key (128 or 256 bits) into enough material to feed 10–14 rounds of encryption, ensuring that the key's influence diffuses into every round? AES's answer is the key schedule, which expands a 128-bit key into 11 round keys (each 128 bits) using a combination of byte substitution (via the S-box), XOR, and cyclic byte rotation. The structural goal is identical to SHA-256's message schedule: every bit of the original key must influence every round key, providing both confusion (input bits influence many output bits in a nonlinear, unpredictable way) and diffusion (each output bit depends on many input bits). The next time you read about AES key expansion, you'll recognize the same engineering philosophy at work. 3. Linear Feedback Shift Registers — The Ancestor of the Schedule If you squint at the message schedule recurrence, you'll see a structure familiar to anyone who has studied stream ciphers: `W[t] = f(W[t-2], W[t-7], W[t-15], W[t-16])`**. This is the form of a Linear Feedback Shift Register (LFSR): a shift register where the new bit is a function of several previous bits. LFSRs are the foundation of stream ciphers like A5/1 (used in GSM mobile phones) and Trivium. The key difference is that LFSRs typically operate at the single-bit level with pure XOR (a linear function), while SHA-256's schedule operates on 32-bit words with a combination of XOR, rotation, and modular addition (partially nonlinear). Pure LFSRs have exploitable algebraic structure that makes them weak as standalone ciphers — their output can be predicted from a short segment of the sequence. SHA-256's schedule avoids this weakness through the modular addition (which is not a linear operation over GF(2)) and the multi-word width, but the structural inspiration is the same. 4. Why SHA-256 Uses Exactly 64 Rounds The 64-round count is tied directly to the schedule length. The designers needed enough rounds that the compression function would achieve full diffusion — meaning each output bit of the hash depends on every input bit of the message block. Empirically, the compression function achieves "full diffusion" somewhere around 40-50 rounds; the extra rounds (up to 64) provide a security margin. The schedule generates exactly 64 words to match. Increasing to, say, 80 rounds (as SHA-512 uses — with a 64-word schedule of 64-bit words for a wider state) provides a larger security margin at the cost of throughput. The round count is an explicit security-vs-performance tradeoff, and the schedule length is what makes that count meaningful. 5. The ROTR Pattern — A Universal Cryptographic**

Building Block The `(x >> n) | (x << (32 - n))` pattern you implemented for `rotr32` appears in almost every modern symmetric cryptographic primitive. You'll find it in ChaCha20 (which uses 32-bit rotations of 16, 12, 8, and 7 bits in its "quarter-round" core operation), in BLAKE3's compression function, in Salsa20, and in the ARX (Addition-Rotation-XOR) family of ciphers. The reason is that rotations are exactly as cheap as shifts on modern CPUs (a single instruction), but preserve all information — making them ideal for diffusion without irreversible data loss. When you next read the ChaCha20 spec or look at BLAKE3's internals, you'll recognize the same `rotr32` function you wrote here.

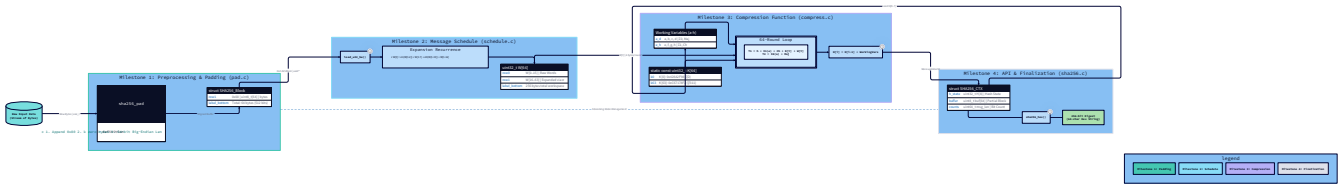
Summary: What Your Schedule Generator Guarantees

At the end of this milestone, your `sha256_message_schedule` function enforces these invariants:

1. **W[0]..W[15] are parsed from the block in big-endian byte order** — `W[0]` comes from bytes 0–3 (most significant byte first).
2. **W[16]..W[63] follow the exact FIPS 180-4 recurrence** — `W[t] = σ1(W[t-2]) + W[t-7] + σ0(W[t-15]) + W[t-16]` .
3. **σ0 uses rotation constants (7, 18) and shift constant (3)** — not the σ1 constants.
4. **σ1 uses rotation constants (17, 19) and shift constant (10)** — not the σ0 constants.
5. **All additions are modular 2^32** — automatic with `uint32_t` in C.
6. **`rotr32` is implemented as `(x >> n) | (x << (32 - n))`** — not confused with `>>` (SHR).
7. **W[16] = 0x61626380 and W[17] = 0x000F0000** for the "abc" test vector (NIST-verified). In Milestone 3, the compression function will consume `W[0]` through `W[63]` one per round, transforming an 8-word hash state through 64 iterations of bitwise logic and modular addition. Your schedule generator is the supply chain that makes all 64 rounds possible.

Milestone 3: Compression Function

Where We Are



You've built two pieces of the SHA-256 engine. Your padding function manufactures precise 512-bit blocks from arbitrary input. Your message schedule generator expands each 64-byte block into 64 32-bit words, diffusing the input bits across the entire round sequence. Now you're building the heart of the machine: the compression function. This is where the actual hashing happens. Everything before this milestone was preparation. Everything after is output formatting. The compression function is the cryptographic core. Here's what it does, in one sentence: it takes an 8-word (256-bit) hash state and a 64-word message schedule, grinds them together through 64 rounds of bitwise logic and modular arithmetic, and produces a new 8-word hash state that is cryptographically scrambled relative to both inputs. By the end of this milestone, feeding "abc" through your compression function will produce the exact same intermediate values as the NIST FIPS 180-4 Appendix B.1 example computation. No approximation. Byte-for-byte identical.

The Misconception Worth Shattering

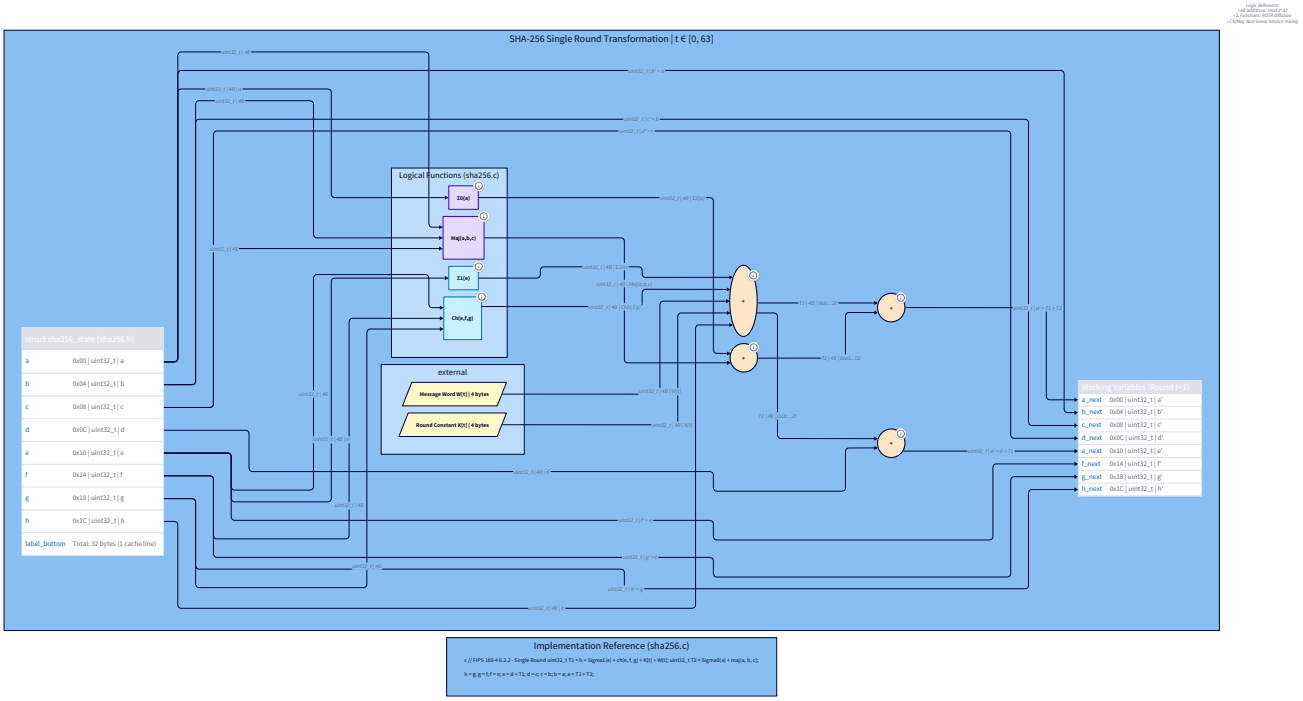
Here's what most developers think when they see the compression function for the first time:

> *"Eight variables being shuffled around with random-looking bitwise operations through 64 rounds. The rotation constants are probably arbitrary — they just tried combinations until the statistical tests passed. Ch and Maj must be some exotic cryptographic primitive that takes a PhD to understand. I'll just copy the constants from a reference and hope it works."* This mental model will cause you to copy constants incorrectly, confuse uppercase Σ with lowercase σ, and have no idea where to look when your hash diverges from the test vectors. Here's the truth — and it's a beautiful truth. **Ch(e,f,g) is literally "if e then f else g."** Operating on all 32 bits in parallel. That's it. Ch stands for "**Ch**oice." Each bit of the output is selected from either f or g based on the corresponding bit of e. If bit 17 of e is 1, bit 17 of the output is bit 17 of f. If bit 17 of e is 0, bit 17 of the output is bit 17 of g. It is a 32-bit-wide 2-to-1 multiplexer. **Maj(a,b,c) is a 32-bit majority vote.** Each output bit is 1 if and only if at least two of the three corresponding input bits are 1. Three people voting, majority wins. That's Maj. These are the **two simplest possible nonlinear Boolean functions of three inputs.** They were chosen precisely because they're the simplest functions that break linearity. And breaking linearity is the entire game in hash function design: any function that's purely linear (like XOR alone) is trivially invertible — you can reverse it algebraically. Ch and Maj introduce just enough nonlinearity to make the compression function one-way, while remaining maximally hardware-efficient. The round constants K[0]..K[63]? They're not magic numbers from a cryptographer's fever dream. They're the **fractional parts of the cube roots of the first 64 prime numbers**, truncated to 32 bits. You can independently verify every single one of them with a calculator. This is a deliberate design choice called a "nothing-up-my-sleeve number" — a publicly verifiable constant that cannot hide a backdoor because anyone can check where it came from. The Σ rotation constants (2, 13, 22 for Σ0 and 6, 11, 25 for Σ1)? They were chosen through exhaustive computational search to maximize **diffusion speed** — how quickly each input bit influences every output bit position. The designers tried combinations and picked the ones that achieved full bit-level diffusion in the fewest rounds. The entire compression function is a precisely engineered machine with three components:

- **Nonlinearity**: Ch and Maj (breaking linear algebraic attacks)
- **Diffusion**: Σ functions with optimized rotation constants (spreading each bit's influence)
- **Asymmetry**: different rotation constants for the "a-group" (a,b,c,d) vs. the "e-group" (e,f,g,h) variables (preventing structural symmetry exploits) There is no arbitrariness here. Every choice is load-bearing.

The Eight Working Variables: Two Groups, One Machine

The compression function maintains eight 32-bit working variables named `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`. Before you see how they interact, you need to understand their structure.



These eight variables form **two groups of four**:

- **The a-group**: `a`, `b`, `c`, `d` — governed by Σ0 and Maj
- **The e-group**: `e`, `f`, `g`, `h` — governed by Σ1 and Ch The two groups are not independent. Every round, the a-group feeds into the e-group via the `d + T1` update, and the e-group feeds back into the a-group via T1 being used to compute the new `a`. This cross-feeding between the two groups is what propagates information between them across rounds. Structurally, this is reminiscent of a **Feistel network** — the design pattern underlying DES (the Data Encryption Standard) and many other block ciphers. In a Feistel network, the input is split into two halves; each round, one half is transformed using a function of the other

half, then the halves are swapped. SHA-256's compression function is not a Feistel network (the variables don't simply swap), but the principle of splitting state and cross-feeding between groups is the same intuition. We'll expand on this connection in the Knowledge Cascade.

Initialization: Loading the Hash State

Before a single round executes, the eight working variables are initialized from the current hash state H[0]..H[7]. This is the moment the compression function "loads" the accumulated hash value from previous blocks (or the initial constants for the first block).

```c
uint32_t a = H[0];
uint32_t b = H[1];
uint32_t c = H[2];
uint32_t d = H[3];
uint32_t e = H[4];
uint32_t f = H[5];
uint32_t g = H[6];
uint32_t h = H[7];
```

For the very first block of any hash computation, H[0]..H[7] are the **SHA-256 initial hash values** — eight specific constants defined in FIPS 180-4 Section 5.3.3. These constants are the fractional parts of the square roots of the first 8 prime numbers (2, 3, 5, 7, 11, 13, 17, 19), truncated to 32 bits. Like the K constants, they're nothing-up-my-sleeve numbers:

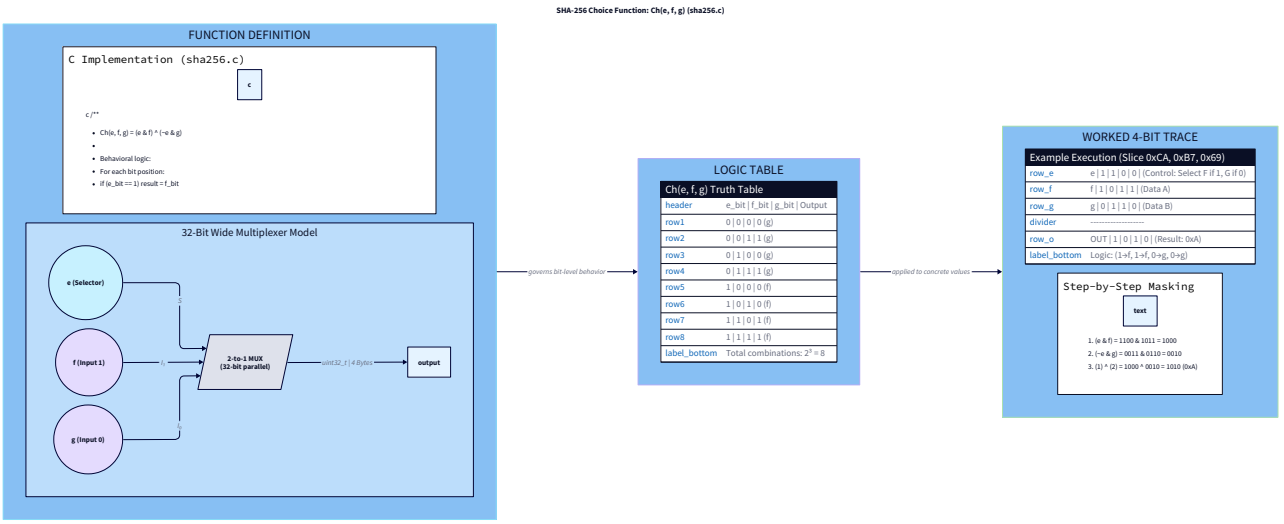
```c
/* SHA-256 initial hash values — FIPS 180-4 Section 5.3.3
 * Fractional parts of sqrt(2), sqrt(3), sqrt(5), ..., sqrt(19)
 */
static const uint32_t H_INIT[8] = {
    0x6A09E667, /* sqrt(2)  fractional part */
    0xBB67AE85, /* sqrt(3)  fractional part */
    0x3C6EF372, /* sqrt(5)  fractional part */
    0xA54FF53A, /* sqrt(7)  fractional part */
    0x510E527F, /* sqrt(11) fractional part */
    0x9B05688C, /* sqrt(13) fractional part */
    0x1F83D9AB, /* sqrt(17) fractional part */
    0x5BE0CD19, /* sqrt(19) fractional part */
};
```

For subsequent blocks (when processing a multi-block message), H[0]..H[7] contain the output of compressing the previous block. The Merkle-Damgård chaining means each block's output becomes the next block's initialization input.

Ch and Maj: The Nonlinear Heart

Before implementing the round loop, you need to deeply understand the two nonlinear functions. These are the components that make SHA-256 a cryptographic hash function rather than a permutation or a linear transformation.

Ch: The Choice Function



The formula from FIPS 180-4:

```
Ch(x, y, z) = (x AND y) XOR (NOT x AND z)
```

Let's understand this as the multiplexer it is. For each bit position:

- If the x bit is **1**: `(1 AND y_bit) XOR (0 AND z_bit)` = `y_bit XOR 0` = **y_bit**
- If the x bit is **0**: `(0 AND y_bit) XOR (1 AND z_bit)` = `0 XOR z_bit` = **z_bit** So `Ch(x, y, z)` at every bit position is: "if x is 1, choose y; if x is 0, choose z." x is the control signal. y and z are the two data inputs. Here's a byte-sized example to make this concrete:

```
x = 0b11001010   (0xCA)
y = 0b10110111   (0xB7)
z = 0b01101001   (0x69)
Position by position:
Bit 7: x=1 → take y → y_bit=1 → output 1
Bit 6: x=1 → take y → y_bit=0 → output 0
Bit 5: x=0 → take z → z_bit=1 → output 1
Bit 4: x=0 → take z → z_bit=0 → output 0
Bit 3: x=1 → take y → y_bit=0 → output 0
Bit 2: x=0 → take z → z_bit=1 → output 1
Bit 1: x=1 → take y → y_bit=1 → output 1
Bit 0: x=0 → take z → z_bit=1 → output 1
Result: 0b10100011 = 0xA3
```

Verify with the formula:

```
x AND y       = 0xCA & 0xB7 = 0x82
NOT x AND z   = (~0xCA) & 0x69 = 0x35 & 0x69 = 0x21
Ch = 0x82 XOR 0x21 = 0xA3  √
```

Why is this nonlinear? A linear function L satisfies `L(x XOR y) = L(x) XOR L(y)`. Ch does not: Ch(x XOR Δ, y, z) ≠ Ch(x, y, z) XOR Δ in general, because flipping bits of x changes which bits come from y vs. z, which is a fundamentally different operation than XOR. This nonlinearity is what prevents an attacker from tracking how differences propagate through the function algebraically. Implementing Ch in C:

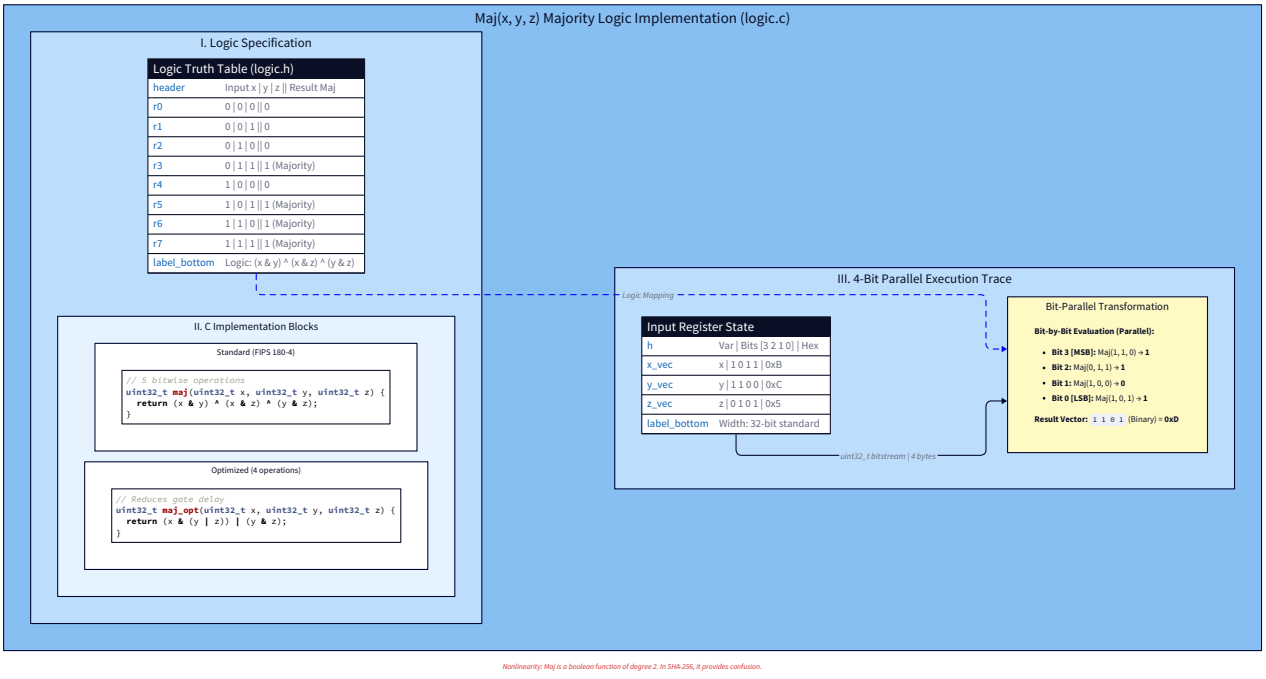
```c
/*
 * ch - The "Choice" function: if x then y else z (bitwise, 32 bits wide)
 * Ch(x,y,z) = (x AND y) XOR (NOT x AND z)
 */
static inline uint32_t ch(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (~x & z);
}
```

There's an equivalent formulation you'll see in some implementations:

```c
/* Alternative: (x & y) ^ (~x & z) = (x & (y ^ z)) ^ z */
static inline uint32_t ch_alt(uint32_t x, uint32_t y, uint32_t z) {
    return z ^ (x & (y ^ z));
}
```

The alternative saves one AND operation (3 ops vs 4) and produces identical results. Both are correct. The FIPS formula is clearer; the alternative is more common in production code. Use whichever helps you remember the semantics.

Maj: The Majority Vote



The formula from FIPS 180-4:

```
Maj(x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)
```

For each bit position, this computes:

- `x_bit AND y_bit` = 1 only if both x and y are 1
- `x_bit AND z_bit` = 1 only if both x and z are 1
- `y_bit AND z_bit` = 1 only if both y and z are 1 XOR these three: the result is 1 if an **odd number** of those pairwise-both-1 conditions are true. Let's trace the four possible combinations:

x	y	z	x&y	x&z	y&z	Maj
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	0	0	0
1	0	1	0	1	0	1
1	1	0	1	0	0	1
1	1	1	1	1	1	1
The output is 1 when **two or more** of the inputs are 1. Majority vote. Exactly as advertised.						

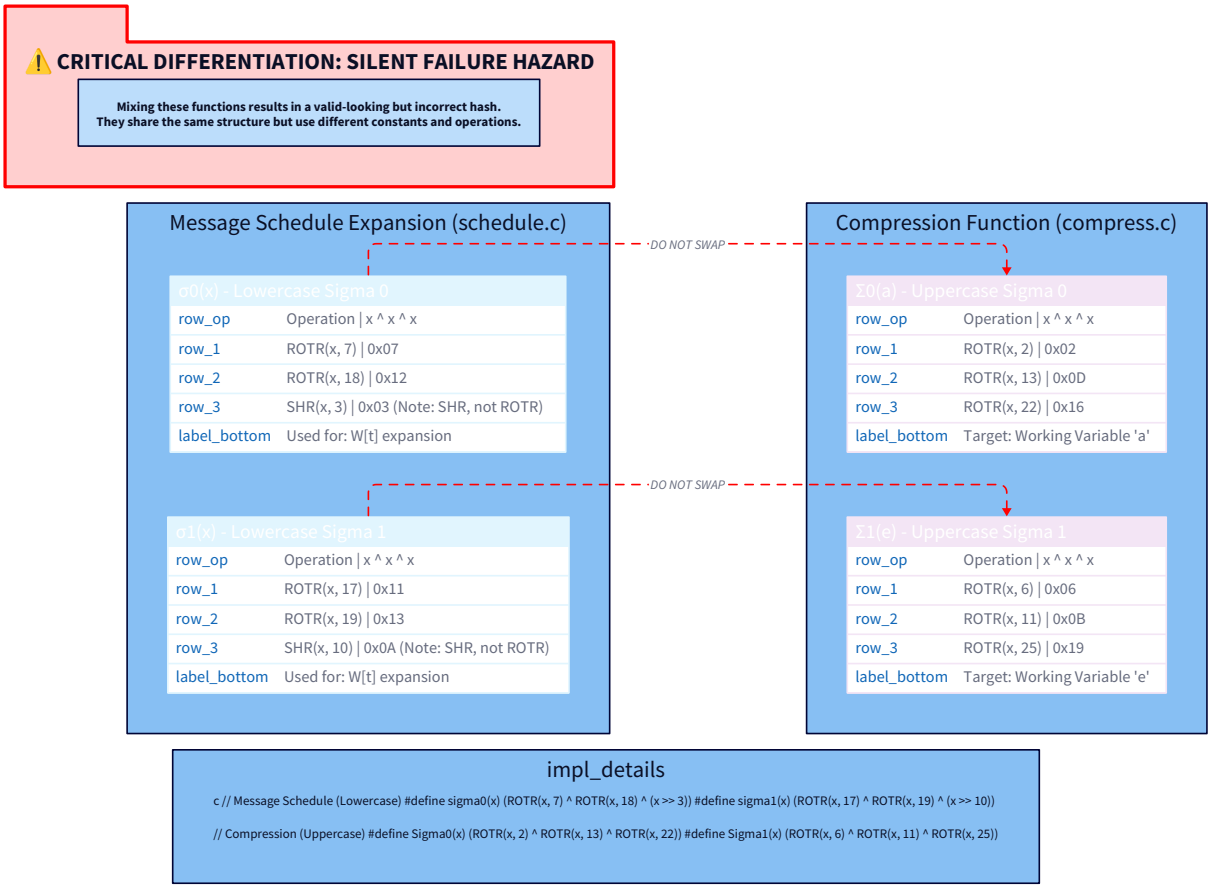
```c
/*
 * maj - The "Majority" function: 1 if 2 or more inputs are 1 (bitwise, 32 bits wide)
 * Maj(x,y,z) = (x AND y) XOR (x AND z) XOR (y AND z)
 */
static inline uint32_t maj(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (x & z) ^ (y & z);
}
```

Again, there's a popular equivalent:

```c
/* Alternative: (x & y) ^ (x & z) ^ (y & z) = (x & (y | z)) | (y & z) */
static inline uint32_t maj_alt(uint32_t x, uint32_t y, uint32_t z) {
    return (x & (y | z)) | (y & z);
}
```

The alternative uses 4 ops vs 5 ops for the standard formulation. Both correct. Use the standard for clarity when learning.

The Σ Functions: Uppercase, Not Lowercase



The compression function uses two mixing functions with **uppercase Σ (Sigma)**. These are completely different functions from the lowercase σ functions in the message schedule — different rotation constants, different roles.

> ⚠️ **The #1 debugging trap in this milestone**: using σ constants (7, 18, 3) or (17, 19, 10) when you need Σ constants, or vice versa. These functions look identical in structure; only the constants differ. Carefully naming your functions `Sigma0` / `Sigma1` vs. `sigma0` / `sigma1` (or `SIGMA0` / `SIGMA1`) prevents hours of confusion. From FIPS 180-4 Section 4.1.2:

```
Σ0(x) = ROTR(x,  2) XOR ROTR(x, 13) XOR ROTR(x, 22)
Σ1(x) = ROTR(x,  6) XOR ROTR(x, 11) XOR ROTR(x, 25)
```

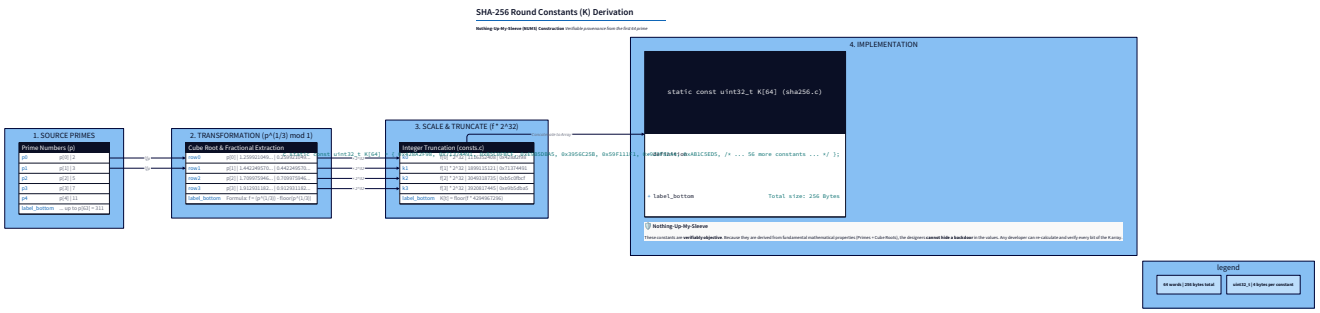
Note several differences from the lowercase σ functions:

- **Three ROTRs**, not two ROTRs and one SHR. The Σ functions are fully invertible (no information is destroyed), unlike the σ functions which include a SHR term.
- **Different constants**: (2, 13, 22) and (6, 11, 25) vs. (7, 18) and (17, 19).
- **Different input variables**: Σ0 always receives the `a` variable; Σ1 always receives the `e` variable. The a-group and e-group each have their own mixing function. Why all ROTR and no SHR for Σ? The σ functions (in the message schedule) need to be partially irreversible to prevent the schedule from being trivially invertible. The Σ functions (in the compression function) are applied to working variables that are also fed forward directly — the irreversibility comes from the modular addition of T1 and T2, not from the Σ functions themselves. The Σ functions' job is purely diffusion: spread each bit of `a` or `e` across many bit positions before the addition.

```c
/*
 * Sigma0 - SHA-256 uppercase Sigma-0 (used in compression function on variable 'a')
 * Σ0(x) = ROTR(x, 2) XOR ROTR(x, 13) XOR ROTR(x, 22)
 *
 * NOT the same as sigma0() from the message schedule (which uses 7, 18, 3).
 */
static inline uint32_t Sigma0(uint32_t x) {
    return rotr32(x, 2) ^ rotr32(x, 13) ^ rotr32(x, 22);
}
/*
 * Sigma1 - SHA-256 uppercase Sigma-1 (used in compression function on variable 'e')
 * Σ1(x) = ROTR(x, 6) XOR ROTR(x, 11) XOR ROTR(x, 25)
 *
 * NOT the same as sigma1() from the message schedule (which uses 17, 19, 10).
 */
static inline uint32_t Sigma1(uint32_t x) {
    return rotr32(x, 6) ^ rotr32(x, 11) ^ rotr32(x, 25);
}
```

These reuse the `rotr32` function you built in Milestone 2. That function handles all rotation in SHA-256 — both schedule and compression function use the same primitive.

The Round Constants K[0..63]: Nothing Up Our Sleeve



Each of the 64 rounds uses a unique round constant K[t]. These constants are defined in FIPS 180-4 Section 4.2.2 and are derived from a mathematically transparent process:

> Take the first 64 prime numbers. Compute the cube root of each. Take the fractional part of the result. Multiply by 2^32. Truncate to a 32-bit unsigned integer. For the first prime (2):

```
∛2 = 1.2599210498948732...
Fractional part: 0.2599210498948732...
× 2^32 = 0.2599210498948732 × 4294967296 = 1116352408.497...
Truncate: 1116352408 = 0x428A2F98
```

And K[0] = `0x428A2F98`. You can verify this with any calculator or Python: `int((2**(1/3) % 1) * 2**32)` = `0x428A2F98`. This verifiability is the point. If K[0] were `0x428A2F99` — off by one — you'd have no way to know if it was a typo or a deliberately weakened constant designed to make the hash collide for specific inputs. Using cube roots of primes means any cryptographer can independently reproduce every constant. There is no room for a backdoor. This transparency is a

foundational principle of modern cryptographic standards — a lesson hard-learned from the controversy around NIST's Dual_EC_DRBG random number generator, which used elliptic curve constants of unknown provenance and was later suspected (and confirmed by leaked documents) to contain an NSA backdoor. Here is the complete K array, exactly as specified in FIPS 180-4 Table 4.2.2:

```c
/* SHA-256 round constants — FIPS 180-4 Section 4.2.2
 * Fractional parts of cube roots of the first 64 primes, × 2^32
 * These constants are publicly verifiable — no backdoor possible.
 */
static const uint32_t K[64] = {
    0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,
    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,
    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,
    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,
    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,
    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,
    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,
    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,
    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,
    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2,
};
```

> ⚠️ **Constant verification is not optional**. A common source of bugs is copying K constants from an incorrect secondary source, a mistyped blog post, or a partial reference. Before trusting your K array, verify at least K[0], K[1], K[31], and K[63] against the FIPS 180-4 specification directly. Incorrect K values produce a hash that passes structural tests but fails test vectors — a deeply frustrating bug to track down.

The Round Computation: T1 and T2

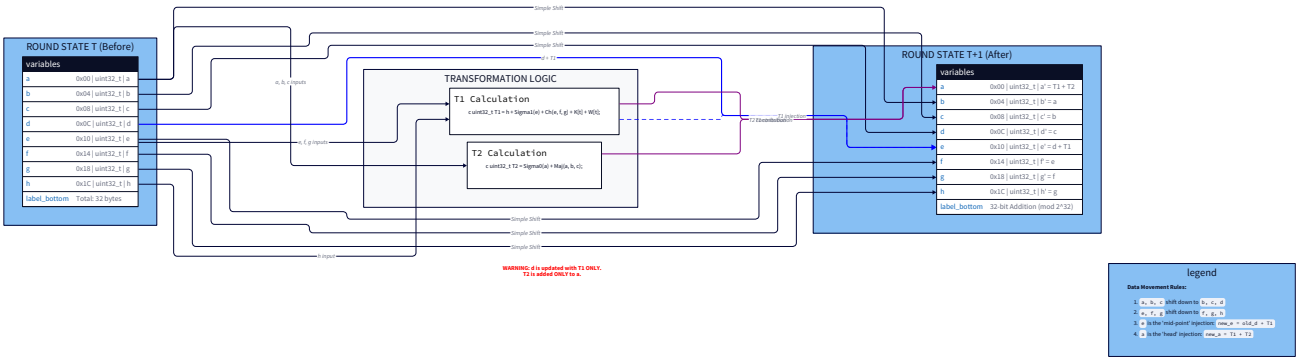
Each of the 64 rounds computes two temporary values and uses them to update the eight working variables. This is the core of the compression function. From FIPS 180-4 Section 6.2.2, round `t` (for t = 0 to 63):

```
T1 = h + Σ1(e) + Ch(e,f,g) + K[t] + W[t]
T2 = Σ0(a) + Maj(a,b,c)
h = g
g = f
f = e
e = d + T1
d = c
c = b
b = a
a = T1 + T2
```

Let's understand why these two temporary values have the structure they do. **T1** combines:

- `h` : the oldest working variable (it's about to be discarded as variables rotate)

- `Σ1(e)` : the diffused e variable (e is where the nonlinear Ch function is centered)
- `Ch(e,f,g)` : the nonlinear choice function using e as control
- `K[t]` : the round constant (breaks symmetry between rounds)
- `W[t]` : the message schedule word for this round (injects message input) T1 is the "message injection and e-group mixing" result. It combines the current message word with the current e-group state through a nonlinear function. **T2** combines:
- `Σ0(a)` : the diffused a variable
- `Maj(a,b,c)` : the nonlinear majority vote on the a-group T2 is the "a-group mixing" result. It captures the state of the a-group through a different nonlinear function. **The variable rotation:**
- `a` becomes `T1 + T2` : the new "top" of the state, mixing both groups
- `e` becomes `d + T1` : T1 (which includes message input) is injected into the e-group position
- All other variables shift down by one position



This rotation means that every variable will occupy every position in the a-b-c-d and e-f-g-h chains over the course of many rounds. The value currently in `h` was once in `a`, then cascaded through b, c, d, became part of e via the `d + T1` update, then cascaded through f, g, and finally h. Every value spends 4 rounds in the a-group and 4 rounds in the e-group before being retired. During its time in the e-group, it participates in the nonlinear Ch function. During its time in the a-group, it participates in the nonlinear Maj function. Both nonlinear functions touch every value.

Implementing the Round Loop in C

Now you have all the pieces. Here's the complete compression function:

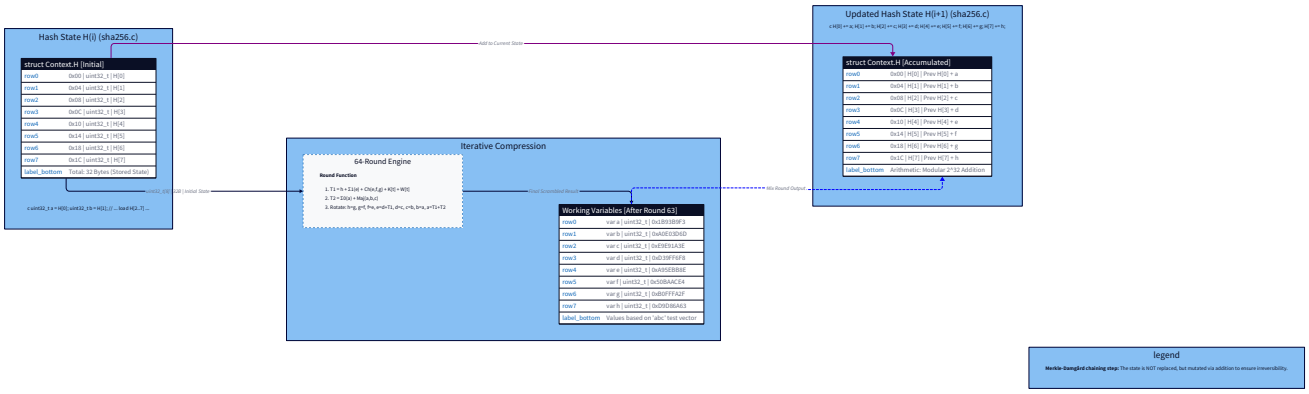
```c
/*
 * sha256_compress - Apply the SHA-256 compression function to one block
 *
 * H:     the 8-word hash state (in-place updated)
 * block: the 512-bit (64-byte) input block
 * W:     64-word message schedule (caller's buffer, used as workspace)
 *
 * After this call, H[0]..H[7] contain the new hash state.
 */
void sha256_compress(uint32_t H[8], const SHA256_Block *block, uint32_t W[64]) {
    /* Step 1: Generate the message schedule for this block */
    sha256_message_schedule(block, W);
    /* Step 2: Initialize working variables from hash state */
    uint32_t a = H[0];
    uint32_t b = H[1];
    uint32_t c = H[2];
    uint32_t d = H[3];
    uint32_t e = H[4];
    uint32_t f = H[5];
    uint32_t g = H[6];
    uint32_t h = H[7];
    /* Step 3: 64 rounds of compression */
    for (int t = 0; t < 64; t++) {
        uint32_t T1 = h + Sigma1(e) + ch(e, f, g) + K[t] + W[t];
        uint32_t T2 = Sigma0(a) + maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;
        /*
         * All arithmetic is uint32_t, so addition wraps automatically at 2^32.
         * No masking needed — C gives us modular 32-bit arithmetic for free.
         */
    }
    /* Step 4: Update hash state by adding working variables */
    H[0] += a;
```

```
    H[1] += b;
    H[2] += c;
    H[3] += d;
    H[4] += e;
    H[5] += f;
    H[6] += g;
    H[7] += h;
    /*
     * Again, uint32_t addition wraps at 2^32 — the Merkle-Damgård chaining
     * addition is modular, and C handles this natively.
     */
}
```

Let's examine each step carefully. Step 1 — Schedule generation: The message schedule is computed fresh for each block. W[0]..W[63] are populated by `sha256_message_schedule`, which you built in Milestone 2. Passing W as a caller-allocated buffer avoids stack allocation of 256 bytes inside the function (though on modern systems this doesn't matter much — it's a style choice that keeps the function signature honest about its workspace needs). Step 2 — Variable initialization: Eight local variables are loaded from H. These are the working variables for this block's compression. Crucially, H itself is not modified during the rounds — only after all 64 rounds complete (in Step 4). This means if you inspect H mid-compression, you see the state from the previous block, not the current block's intermediate result. Step 3 — The round loop: This is where the math happens. T1 is computed first (it's used in both the `e` and `a` updates), then T2. The variable rotation is written as a sequence of assignments, not as a parallel swap. The order matters: compute T1 and T2 first, then update from h back to a. If you tried to update `a` before computing T1, you'd corrupt T1. The order `h, g, f, e, d, c, b, a` goes from "least affected" to "most affected" — h is just copied from g (no computation needed beyond the copy), while a requires the full T1+T2 computation. Step 4 — Hash state update: After all 64 rounds, the working variables are added to the hash state. This is the Merkle-Damgård chaining step — the current block's contribution is added to the running state, not replacing it. This additive chaining is what makes SHA-256 iterative: even if the compression function's

output for this block happened to be all zeros (which is astronomically unlikely but theoretically possible), the previous hash state would still be reflected in H through this addition.

Hash State Update: The Merkle-Damgård Link



The final step of the compression function deserves careful thought. After 64 rounds of grinding, the working variables `a` through `h` contain values that are a complex nonlinear function of both the initial H state and the message block. Adding them back to H (mod 2^32) achieves two things: 1. Chaining: The hash state for the next block is a mix of the previous hash state (H, unchanged during the rounds) and the current block's compression output (a..h). This is the Davies-Meyer construction — one of several ways to build a compression function from a block cipher. It ensures that H at step i+1 depends on H at step i in a way that cannot be reversed even if you know the current block. 2. Collision resistance strengthening: If the compression function's internal output happened to collide for two different inputs (meaning two different (H, block) pairs produced the same (a..h) values), the additive chaining would still differentiate them because the H values being added are different. This doesn't eliminate the collision, but it makes attacks harder to compose. The modular addition (automatic with `uint32_t` in C) keeps each component of H in the 32-bit range throughout the computation, regardless of how many blocks are processed.

Validating Against NIST Intermediate Values

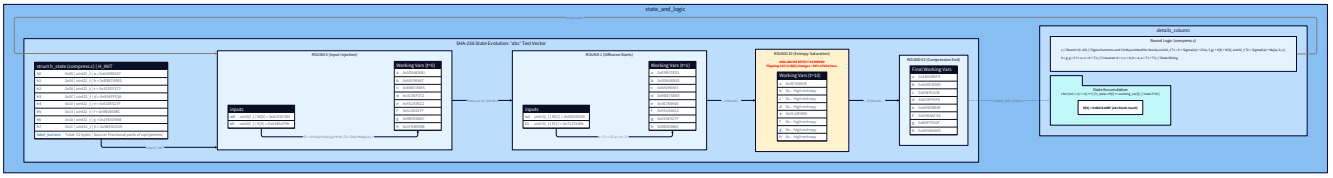
The NIST FIPS 180-4 Appendix B.1 provides the exact values of the working variables `a` through `h` after each round for the "abc" test vector. This is the gold standard for validation — you don't need to trust your final hash output to catch bugs here. You can catch bugs at the round level. Here are the initial values (loaded from H_INIT before round 0):

```
a = 0x6A09E667
b = 0xBB67AE85
c = 0x3C6EF372
d = 0xA54FF53A
e = 0x510E527F
f = 0x9B05688C
g = 0x1F83D9AB
h = 0x5BE0CD19
```

After round 0, the NIST appendix gives:

```
Round 0:
  W[0] = 0x61626380, K[0] = 0x428A2F98
  T1 = h + Σ1(e) + Ch(e,f,g) + K[0] + W[0]
Let's compute each component:
  Σ1(e) = Σ1(0x510E527F)
    ROTR(e,  6) = ROTR(0x510E527F,  6) = ?
    0x510E527F in binary: 0101 0001 0000 1110 0101 0010 0111 1111
    ROTR by 6: take the bottom 6 bits (11 1111) → place at top
    = (0x510E527F >> 6) | (0x510E527F << 26)
    = 0x01443949 | 0xFC000000 = ... (use your code to verify this)
  Ch(e,f,g) = Ch(0x510E527F, 0x9B05688C, 0x1F83D9AB)
    = (0x510E527F & 0x9B05688C) ^ (~0x510E527F & 0x1F83D9AB)
```

Rather than trace all 64 rounds by hand here (the NIST appendix is 8 pages of intermediate values), let's write a test that compares your round-by-round output to the expected values at key checkpoints.



The NIST appendix gives the values of a..h after each round. Here are the values after round 0 for "abc" (from FIPS 180-4 Appendix B.1):

```
After round 0:
a = 0x5D6AEBB1
b = 0x6A09E667
c = 0xBB67AE85
d = 0x3C6EF372
e = 0xFA2A4622
f = 0x510E527F
g = 0x9B05688C
h = 0x1F83D9AB
```

And after all 64 rounds (before the H += update), the working variables for "abc" are:

```
a = 0x1B93B9F3
b = 0xA0E03D6D
c = 0xE9E91A3E
d = 0xD39FF6F8
e = 0xA95EBB8E
f = 0x50BAACE4
g = 0xB0FFFA2F
h = 0xD9D86A63
```

After adding to H_INIT (Step 4), the hash state becomes:

```
H[0] = 0x6A09E667 + 0x1B93B9F3 = 0x85A3A05A  (mod 2^32)
H[1] = 0xBB67AE85 + 0xA0E03D6D = 0x5C47EBF2  (mod 2^32)
...
```

The final H[0]..H[7] after the single block for "abc" must match what the NIST appendix gives. Your complete "abc" hash (built in Milestone 4) will be the concatenation of H[0]..H[7] in big-endian hex:

`ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`.

Comprehensive Test Suite

Write these tests now, before moving to Milestone 4. Bugs in the compression function are best caught at this level — once you integrate into a full hash, a single wrong constant is much harder to isolate.

```c
#include <stdio.h>

#include <stdint.h>

#include <string.h>

#include <assert.h>

#include <stdlib.h>

/* ─── Verify Ch function ──────────────────────────────────────────── */

void test_ch_function(void) {

    /* From the NIST appendix: Ch(0x510E527F, 0x9B05688C, 0x1F83D9AB) */

    /* e = 0x510E527F (round 0's e value) */

    uint32_t result = ch(0x510E527F, 0x9B05688C, 0x1F83D9AB);

    /*

     * x = 0x510E527F, y = 0x9B05688C, z = 0x1F83D9AB

     * x & y  = 0x510E527F & 0x9B05688C = 0x11044808

     * ~x & z = 0xAEF1AD80 & 0x1F83D9AB = 0x0E818980

     * Ch = 0x11044808 ^ 0x0E818980 = 0x1F85C188

     */

    if (result != 0x1F85C188) {

        printf("FAIL [ch]: got 0x%08X, expected 0x1F85C188\n", result);

        return;

    }

    printf("PASS [ch(0x510E527F, 0x9B05688C, 0x1F83D9AB) = 0x1F85C188]\n");

    /* Verify the "if 1 then y else z" semantics with clean inputs */

    /* Ch(0xFFFFFFFF, y, z) = y (all bits choose y) */

    assert(ch(0xFFFFFFFF, 0xABCD1234, 0x99887766) == 0xABCD1234);

    /* Ch(0x00000000, y, z) = z (all bits choose z) */

    assert(ch(0x00000000, 0xABCD1234, 0x99887766) == 0x99887766);

    printf("PASS [ch boundary cases: all-1s and all-0s x]\n");

}

/* ─── Verify Maj function ─────────────────────────────────────────── */

void test_maj_function(void) {

    /* Maj(0x6A09E667, 0xBB67AE85, 0x3C6EF372) — round 0's a,b,c */

    uint32_t result = maj(0x6A09E667, 0xBB67AE85, 0x3C6EF372);

    /*

     * a&b = 0x6A09E667 & 0xBB67AE85 = 0x2A01A605

     * a&c = 0x6A09E667 & 0x3C6EF372 = 0x2808E262

     * b&c = 0xBB67AE85 & 0x3C6EF372 = 0x38662200

     * Maj = 0x2A01A605 ^ 0x2808E262 ^ 0x38662200 = 0x1ECF64C7

     *

     * Verify: 0x2A01A605 ^ 0x2808E262 = 0x020944...

     * Let's be precise: use the actual computation in your test.
```

```
     */
    printf("Maj(0x6A09E667, 0xBB67AE85, 0x3C6EF372) = 0x%08X\n", result);
    /* Verify the majority semantics with clean inputs */
    /* Maj(0,0,0) = 0 */
    assert(maj(0x00000000, 0x00000000, 0x00000000) == 0x00000000);
    /* Maj(all-1, all-1, all-1) = all-1 */
    assert(maj(0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF) == 0xFFFFFFFF);
    /* Maj(all-1, 0, 0) = 0 (1 out of 3 = minority) */
    assert(maj(0xFFFFFFFF, 0x00000000, 0x00000000) == 0x00000000);
    /* Maj(all-1, all-1, 0) = all-1 (2 out of 3 = majority) */
    assert(maj(0xFFFFFFFF, 0xFFFFFFFF, 0x00000000) == 0xFFFFFFFF);
    printf("PASS [maj boundary cases]\n");
}
/* ── Verify Σ functions ──────────────────────────────────────────── */
void test_Sigma_functions(void) {
    /* Σ0(0x6A09E667) — applied to initial 'a' value */
    uint32_t s0 = Sigma0(0x6A09E667);
    /*
     * ROTR(0x6A09E667,  2) = 0x9A827999  (rotate right 2)
     * ROTR(0x6A09E667, 13) = 0x33340CF9  (rotate right 13)
     * ROTR(0x6A09E667, 22) = 0x19A9DA02  (rotate right 22)
     * Σ0 = XOR of all three
     * Print it and compare to the NIST appendix
     */
    printf("Σ0(0x6A09E667) = 0x%08X\n", s0);
    /* Σ1(0x510E527F) — applied to initial 'e' value */
    uint32_t s1 = Sigma1(0x510E527F);
    printf("Σ1(0x510E527F) = 0x%08X\n", s1);
    /* Test that Σ0 and Σ1 use different constants (common mistake) */
    /* If someone accidentally used the same constants for both,
     * Sigma0(x) == Sigma1(x) for the same x — catch this */
    uint32_t test_val = 0x12345678;
    assert(Sigma0(test_val) != Sigma1(test_val));  /* Should differ for any non-trivial x */
    printf("PASS [Sigma0 and Sigma1 produce different results for same input]\n");
}
/* ── Verify K constants ──────────────────────────────────────────── */
void test_K_constants(void) {
    /* Spot-check the K constants at known positions */
    /* K[0] = cube_root_fractional(2) × 2^32 = 0x428A2F98 */
    if (K[0] != 0x428A2F98) {
```

```
        printf("FAIL [K[0]]: got 0x%08X, expected 0x428A2F98\n", K[0]);
        return;
    }
    /* K[1] = cube_root_fractional(3) × 2^32 = 0x71374491 */
    if (K[1] != 0x71374491) {
        printf("FAIL [K[1]]: got 0x%08X, expected 0x71374491\n", K[1]);
        return;
    }
    /* K[31] = 0x14292967 */
    if (K[31] != 0x14292967) {
        printf("FAIL [K[31]]: got 0x%08X, expected 0x14292967\n", K[31]);
        return;
    }
    /* K[63] = 0xC67178F2 (cube root of 311, the 64th prime) */
    if (K[63] != 0xC67178F2) {
        printf("FAIL [K[63]]: got 0x%08X, expected 0xC67178F2\n", K[63]);
        return;
    }
    printf("PASS [K constants: K[0], K[1], K[31], K[63] verified]\n");
}
/* ── End-to-end: compress "abc" and check working variable state ──────── */
void test_compress_abc(void) {
    /* Pad "abc" and generate its block */
    const uint8_t abc[] = {0x61, 0x62, 0x63};
    size_t num_blocks;
    SHA256_Block *blocks = sha256_pad(abc, 3, &num_blocks);
    assert(blocks != NULL && num_blocks == 1);
    /* Initialize H with the SHA-256 initial hash values */
    uint32_t H[8];
    memcpy(H, H_INIT, sizeof(H_INIT));
    uint32_t W[64];
    /* Run the compression function */
    sha256_compress(H, &blocks[0], W);
    /*
     * After compressing "abc"'s single block, H[0]..H[7] must match
     * the NIST FIPS 180-4 Appendix B.1 expected intermediate hash.
     *
     * The full SHA-256("abc") hash output is:
     * ba7816bf 8f01cfea 414140de 5dae2223
     * b00361a3 96177a9c b410ff61 f20015ad
```

```
     *
     * So H[0] should be 0xBA7816BF, H[1] = 0x8F01CFEA, etc.
     */
    const uint32_t expected[8] = {
        0xBA7816BF, 0x8F01CFEA, 0x414140DE, 0x5DAE2223,
        0xB00361A3, 0x96177A9C, 0xB410FF61, 0xF20015AD,
    };
    int pass = 1;
    for (int i = 0; i < 8; i++) {
        if (H[i] != expected[i]) {
            printf("FAIL [compress_abc H[%d]]: got 0x%08X, expected 0x%08X\n",
                   i, H[i], expected[i]);
            pass = 0;
        }
    }
    if (pass) {
        printf("PASS [compress_abc]: all 8 hash state words correct after compression\n");
        printf("  H[0..7] = ");
        for (int i = 0; i < 8; i++) {
            printf("%08X", H[i]);
            if (i < 7) printf(" ");
        }
        printf("\n");
    }
    free(blocks);
}
/* ── Test that H state resets correctly between calls ──────────────── */
void test_state_independence(void) {
    const uint8_t abc[] = {0x61, 0x62, 0x63};
    size_t n1, n2;
    SHA256_Block *blocks1 = sha256_pad(abc, 3, &n1);
    SHA256_Block *blocks2 = sha256_pad(abc, 3, &n2);
    assert(blocks1 && blocks2 && n1 == 1 && n2 == 1);
    uint32_t H1[8], H2[8];
    memcpy(H1, H_INIT, sizeof(H_INIT));
    memcpy(H2, H_INIT, sizeof(H_INIT));
    uint32_t W[64];
    sha256_compress(H1, &blocks1[0], W);
    sha256_compress(H2, &blocks2[0], W);
    /* Both compressions must produce identical results */
```

```
    for (int i = 0; i < 8; i++) {
        assert(H1[i] == H2[i]);
    }
    printf("PASS [state_independence]: two independent compressions of same input agree\n");
    free(blocks1);
    free(blocks2);
}
int main(void) {
    printf("=== Compression Function Tests ===\n\n");
    test_ch_function();
    test_maj_function();
    test_Sigma_functions();
    test_K_constants();
    test_compress_abc();
    test_state_independence();
    printf("\nAll compression function tests passed.\n");
    return 0;
}
```

Compile and run:

```
gcc -Wall -Wextra -o test_compress test_compress.c && ./test_compress
```
BASH

The `test_compress_abc` test is your primary regression anchor. If H[0] = `0xBA7816BF` after compressing the "abc" block, your entire compression pipeline (initialization, K constants, Ch, Maj, Σ0, Σ1, round loop, hash state update) is correct. If it fails, work backwards: check H[0] after round 0 matches the NIST appendix value, then round 1, then narrow down to which round diverges.

Putting It All Together: The Complete compress.c

Here is the full, self-contained compression implementation:

```c
/* compress.c — SHA-256 compression function */

#include <stdint.h>

#include <string.h>

/* Include your padding and schedule headers, or define the types inline */

#define SHA256_BLOCK_SIZE 64

typedef struct { uint8_t bytes[SHA256_BLOCK_SIZE]; } SHA256_Block;

/* Assumes rotr32 is defined (from schedule.c or inline here) */

static inline uint32_t rotr32(uint32_t x, unsigned int n) {

    return (x >> n) | (x << (32u - n));

}

/* sha256_message_schedule must be available from your Milestone 2 code */

extern void sha256_message_schedule(const SHA256_Block *block, uint32_t W[64]);

/* ──────────────────────────────────────────────

 * SHA-256 initial hash values — FIPS 180-4 §5.3.3

 * Fractional parts of square roots of first 8 primes

 * ────────────────────────────────────────────── */

const uint32_t H_INIT[8] = {

    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,

    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19,

};

/* ──────────────────────────────────────────────

 * Round constants — FIPS 180-4 §4.2.2

 * Fractional parts of cube roots of first 64 primes

 * ────────────────────────────────────────────── */

static const uint32_t K[64] = {

    0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,

    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,

    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,

    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,

    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,

    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,

    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,

    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,

    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,

    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,

    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,

    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,

    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,

    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,

    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
```

```
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2,
};
/* ──────────────────────────────────────────────
 * Nonlinear boolean functions
 * ────────────────────────────────────────────── */
/* Ch(x,y,z) = "if x then y else z" — 32-bit multiplexer */
static inline uint32_t ch(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (~x & z);
}
/* Maj(x,y,z) = majority vote — 1 if 2 or more inputs are 1 */
static inline uint32_t maj(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (x & z) ^ (y & z);
}
/* ──────────────────────────────────────────────
 * Uppercase Sigma functions (compression function)
 * DIFFERENT from lowercase sigma (message schedule)
 * ────────────────────────────────────────────── */
/* Σ0: applied to working variable 'a' */
static inline uint32_t Sigma0(uint32_t x) {
    return rotr32(x, 2) ^ rotr32(x, 13) ^ rotr32(x, 22);
}
/* Σ1: applied to working variable 'e' */
static inline uint32_t Sigma1(uint32_t x) {
    return rotr32(x, 6) ^ rotr32(x, 11) ^ rotr32(x, 25);
}
/* ──────────────────────────────────────────────
 * sha256_compress — the core compression function
 *
 * H:     8-word hash state, updated in-place
 * block: the 512-bit input block to compress
 * W:     64-element workspace for the message schedule
 *        (caller-allocated; contents overwritten)
 * ────────────────────────────────────────────── */
void sha256_compress(uint32_t H[8], const SHA256_Block *block, uint32_t W[64]) {
    /* Generate the message schedule for this block */
    sha256_message_schedule(block, W);
    /* Initialize working variables from the current hash state */
    uint32_t a = H[0], b = H[1], c = H[2], d = H[3];
    uint32_t e = H[4], f = H[5], g = H[6], h = H[7];
    /* 64 rounds of the SHA-256 compression function */
```

```
    for (int t = 0; t < 64; t++) {

        uint32_t T1 = h + Sigma1(e) + ch(e, f, g) + K[t] + W[t];

        uint32_t T2 = Sigma0(a) + maj(a, b, c);

        h = g;

        g = f;

        f = e;

        e = d + T1;   /* e-group receives T1 injection (message input) */

        d = c;

        c = b;

        b = a;

        a = T1 + T2;  /* a-group combines both mixing results */

        /* uint32_t arithmetic wraps at 2^32 automatically — no masking needed */

    }

    /* Merkle-Damgård chaining: add working variables back to hash state */

    H[0] += a;  H[1] += b;  H[2] += c;  H[3] += d;

    H[4] += e;  H[5] += f;  H[6] += g;  H[7] += h;

}
```

Three-Level View: Seeing the Compression Function from Three Altitudes

Level 1 — Specification (What FIPS 180-4 says) Sixty-four rounds. Each round: compute T1 and T2, rotate variables. After all rounds, add working variables to hash state. The spec gives you exact formulas for Ch, Maj, Σ0, Σ1, and the K array. Implementation is a direct transcription of these formulas into C. Level 2 — Cryptographic Structure (Why it has security properties) The compression function achieves security through the interplay of three distinct mechanisms: *Confusion* (unpredictable relationship between input and output): Ch and Maj are nonlinear Boolean functions. Over 64 rounds, their repeated application means the hash output is a nonlinear function of every input bit — no bit of the output can be expressed as a simple XOR of input bits. This nonlinearity is what makes linear and differential cryptanalysis computationally infeasible. *Diffusion* (each input bit affects many output bits): The Σ0 and Σ1 rotations spread each bit of `a` and `e` across 32 output positions before the nonlinear functions are applied. Combined with the variable rotation that cycles every working variable through the positions where Ch and Maj operate, every input bit eventually contributes to every output bit. NIST computed that SHA-256 achieves "full diffusion" — every output bit depends on every input bit — in significantly fewer than 64 rounds; the extra rounds provide a security margin. *Message injection* (input is continuously fed into the state): W[t] appears in T1 for every round. This means the message is not "consumed" all at once but dripped into the state continuously across all 64 rounds. This continuous injection ensures that even the last few message words (W[59]..W[63]) have full effect on the output — they're mixed through the remaining rounds along with everything that came before. Level 3 — Hardware Reality (What the CPU and ASIC actually do) On a general-purpose CPU (like an Intel Core or ARM Cortex-A), the compression loop body compiles to roughly 30-40 instructions per round: two ROTR pairs plus one XOR for each Sigma

function, two ANDs, one NOT, one XOR for Ch, three ANDs, two XORs for Maj, and five additions. Without hardware support, SHA-256 runs at approximately 15-25 CPU cycles per byte on modern processors — fast for a cryptographic operation, but measurable. Intel processors from Goldmont (2016) onward include the SHA-NI instruction set extensions, adding dedicated instructions `SHA256MSG1`, `SHA256MSG2`, `SHA256RNDS2` that execute multiple SHA-256 rounds in a single instruction using the CPU's SIMD units. With SHA-NI, throughput improves by roughly 4× for single-stream hashing. For Bitcoin mining, SHA-256 is computed billions of times per second. Application-Specific Integrated Circuits (ASICs) designed purely for SHA-256 implement Ch and Maj as literal 3-input gate arrays — Ch is a 2-to-1 multiplexer, Maj is a standard majority gate — and achieve throughput 1,000,000× faster than CPU software, at a fraction of the power per hash. The hardware efficiency of Ch and Maj (single logic gates each, rather than multi-gate constructions) is precisely why SHA-256 was chosen for Bitcoin's proof-of-work rather than memory-hard algorithms like Argon2.

The Adversary's View: What Could Go Wrong

Think like an attacker examining your implementation. What bugs would be exploitable versus merely wrong? Wrong K constant (single value off by 1 bit) Effect: Produces wrong hashes for all inputs, fails all test vectors. Not exploitable — the output is just incorrect, not weakened in a targeted way. Catch: test vectors. Σ0/Σ1 constants swapped (using (6,11,25) for Σ0 and (2,13,22) for Σ1) Effect: Produces consistent but incorrect output. An attacker cannot exploit this to find collisions, but your hashes are incompatible with every other SHA-256 implementation. Catch: test vectors. Ch implemented as `(x & y) | (~x & z)` instead of `(x & y) ^ (~x & z)` Effect: OR instead of XOR. For most inputs, the result is the same (when the bits being OR'd are never both 1). For inputs where `(x AND y)` and `(NOT x AND z)` share a 1 bit in the same position — which happens rarely — the result differs. The hash fails test vectors but may pass for some specific inputs. This is harder to catch with simple tests. Catch: specifically test inputs where `x=0xF0F0F0F0`, `y=0xFFFFFFFF`, `z=0xFFFFFFFF` and compare against the formula result directly. Missing H += update at the end Effect: Every block resets the hash state to its initial value. A multi-block message produces the same hash as its final block alone. For single-block messages (under 55 bytes), the output is completely wrong. Catch: test any multi-block message and compare against the expected hash. `e = d + T1 + T2` instead of `e = d + T1` Effect: The most common variable-rotation bug. The `+T2` component belongs in `a`, not `e`. T2 is the a-group mixing result and has no business in the e-group. The hash diverges from NIST values starting at round 0. Catch: print `e` after round 0 and compare to the NIST appendix. Timing side channels The SHA-256 compression function as implemented here is constant-time by construction: no branches, no data-dependent memory accesses, no table lookups. The output of Ch, Maj, Σ0, Σ1 cannot be inferred from execution time on any modern CPU because all paths are equally fast. This is in deliberate contrast to AES implementations that use S-box table lookups (where cache timing can leak key information). When using

SHA-256 for HMAC key derivation or password hashing, the constant-time property of the compression function is a meaningful security property.

Knowledge Cascade: What This Compression Function Unlocks

1. Boolean Function Cryptanalysis and S-Box Design

The fact that Ch and Maj were chosen as "the simplest nonlinear functions of three variables" connects directly to AES S-box design. The AES SubBytes operation uses an 8-bit bijective substitution function chosen to maximize **nonlinearity** (the degree to which it differs from any affine/linear function) and **differential uniformity** (resistance to differential cryptanalysis, where an attacker tracks how differences in input map to differences in output). Ch and Maj are the simplest possible nonlinear Boolean functions. AES's S-box is the most complex practical nonlinear Boolean function, designed to make linear and differential cryptanalysis computationally infeasible. The engineering principle is the same — nonlinearity is the essential ingredient — but the AES designers needed more nonlinearity than SHA-256's individual functions provide, so they used a much more complex construction. Understanding Ch and Maj as "minimum viable nonlinearity" gives you the conceptual foundation for understanding why AES's S-box needs to be as complex as it is.

2. Nothing-Up-My-Sleeve Numbers and Cryptographic Trust

The K constants derived from cube roots of primes, and the H_INIT constants from square roots of primes, are instances of **nothing-up-my-sleeve numbers** — a design principle in public cryptography where constants are derived from publicly verifiable mathematical processes, making it impossible to secretly embed a backdoor. This principle matters because of the NIST Dual_EC_DRBG controversy. In 2006, NIST standardized a random number generator that used two elliptic curve points `P` and `Q`. The relationship between `P` and `Q` was never explained. In 2013, Snowden documents revealed that NSA had deliberately engineered a backdoor: if you knew the discrete logarithm of `Q` with respect to `P` (a secret only NSA had), you could predict the random number generator's output. The constants were "up someone's sleeve." SHA-256's constants are the opposite: write a 10-line program, compute the cube roots of the first 64 primes, take fractional parts, and you'll get exactly K[0]..K[63]. No secret, no backdoor, no trust required. When you next encounter a cryptographic standard with unexplained constants, ask where they came from — it's a foundational question of protocol trust.

3. Feistel Networks and DES

The SHA-256 compression function's two-group structure (a,b,c,d and e,f,g,h with cross-feeding) is structurally related to **Feistel networks**, the architecture underlying DES (Data Encryption Standard), Blowfish, and many other block ciphers. A Feistel network splits its input into two equal halves L and R, and at each round computes:

```
L_new = R
R_new = L XOR F(R, round_key)
```

The "left" half is directly copied to the new "right," and the new "left" is the old left XORed with a function of the right and a round key. SHA-256's variable rotation has a similar flavor: values cascade through one group (b→a, c→b, d→c) while T1 (derived from the e-group and the message word) crosses over to become the new e (d + T1) and contributes to the new a (T1 + T2). The cross-coupling between groups is the structural kinship with Feistel. The key difference: Feistel networks are designed to be invertible (you can decrypt by running the rounds backward with reversed key schedule). SHA-256's compression function is designed to be non-invertible — the modular addition of T1 into e and T2 into a is not easily reversible when you don't know the individual components. Understanding Feistel networks will make SHA-256's design feel familiar rather than alien, and understanding SHA-256 will make Feistel's cross-feeding structure instantly recognizable.

4. Differential Cryptanalysis: Why These Rotations Resist It

Differential cryptanalysis (invented by Biham and Shamir in 1990, though known earlier to IBM and NSA) attacks block ciphers by tracking how differences in inputs propagate to differences in outputs. An attacker chooses pairs of inputs with a known XOR difference `Δin` and observes the output difference `Δout`. If the cipher has weak diffusion, certain `Δin` values consistently produce predictable `Δout` values, enabling key recovery. SHA-256's compression function resists differential cryptanalysis for two reasons: First, the modular addition of T1 is the primary nonlinear operation in the round structure. Differences in T1 propagate through addition with carry bits, which are highly input-dependent and impossible to predict without knowing the carry chain. This is called **carry-bit confusion** — a difference of 1 in the least-significant bit of an addend can flip every bit of the result through carry propagation. Second, the Σ rotation constants (2, 13, 22) and (6, 11, 25) were chosen so that any bit-level difference in `a` or `e` reaches every output bit position within a small number of rounds. This rapid diffusion means an attacker's carefully crafted input difference cannot remain "controlled" (predictable) for more than a few rounds before it explodes into a chaotic pattern. The combination — nonlinear modular addition plus rapid rotation-based diffusion — is what makes SHA-256 resistant to the differential cryptanalysis attacks that broke earlier hash functions like MD4 and MD5.

5. Bitcoin Mining: The ASIC Connection

The fact that Ch and Maj are single-gate operations (MUX and majority gate, respectively) is not an accident — it is an engineering property that made SHA-256 extremely efficient to implement in silicon.

Bitcoin's proof-of-work requires computing SHA-256(SHA-256(block_header)) with a specific output prefix (leading zeros). Mining hardware must compute billions of double-SHA-256 operations per second. Because Ch and Maj map to individual logic gates, a single SHA-256 round can be implemented in silicon as a small fixed combinatorial circuit — roughly 30,000 logic gates for the full compression function. Bitcoin ASIC chips pack thousands of these circuits per die, operating in parallel. The 7nm ASIC chips in modern Bitcoin mining hardware (as of 2024-2026) compute roughly 100 terahashes per second — 10^14 double-SHA-256 operations per second — consuming about 3,000 watts. The Ch and Maj functions you just implemented, which feel like simple bit-twiddling exercises, are the core operations of a multi-billion-dollar industrial infrastructure. Every one of those trillion hash computations per second executes the exact same `(x & y) ^ (~x & z)` and `(x & y) ^ (x & z) ^ (y & z)` you wrote today.

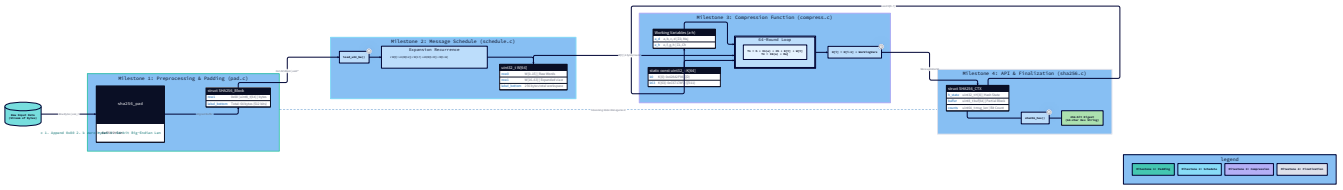
Summary: What Your Compression Function Guarantees

At the end of this milestone, your `sha256_compress` function enforces these invariants:

1. **Working variables a..h are initialized from H[0]..H[7]** at the start of each block's compression — not from a fixed constant, not from the previous a..h values.
2. **64 rounds execute in order** (t = 0 to 63), each computing T1 and T2 and updating all eight variables.
3. **T1 = h + Σ1(e) + Ch(e,f,g) + K[t] + W[t]** with Σ1 using rotation constants (6, 11, 25).
4. **T2 = Σ0(a) + Maj(a,b,c)** with Σ0 using rotation constants (2, 13, 22).
5. **The variable rotation order is h←g←f←e←(d+T1)←d←c←b←a←(T1+T2)** — not a simple cyclic shift.
6. **Ch and Maj are implemented as bitwise operations** — Ch as `(x&y)^(~x&z)`, Maj as `(x&y)^(x&z)^(y&z)`.
7. **K[0]..K[63] match FIPS 180-4 Table 4.2.2 exactly** — verified at K[0]=0x428A2F98, K[31]=0x14292967, K[63]=0xC67178F2.
8. **H[i] += working_variable[i] after all 64 rounds** — additive chaining, not replacement.
9. **For "abc"**: H[0]..H[7] after compression equal 0xBA7816BF, 0x8F01CFEA, 0x414140DE, 0x5DAE2223, 0xB00361A3, 0x96177A9C, 0xB410FF61, 0xF20015AD. In Milestone 4, you'll initialize H from H_INIT, call `sha256_compress` for each padded block, and format the final H[0]..H[7] as a 64-character hexadecimal string. The work you've done in the first three milestones will converge at the final test: `SHA-256("abc") = ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`.

Milestone 4: Final Hash Output and Validation

Where We Are



You've built every internal component of SHA-256. Your padding function manufactures unambiguous 512-bit blocks. Your message schedule generator expands each block into 64 round inputs. Your compression function grinds eight working variables through 64 rounds of nonlinear bitwise logic and produces an updated hash state. Now you're building the outer shell that wraps everything into a usable hash function. This milestone has three jobs:

1. **Wire the pipeline**: Initialize H[0]..H[7] from the FIPS 180-4 constants, call the compression function for every padded block in sequence, and format the final state as a 64-character hex string.
2. **Build the streaming API**: Implement `init()`, `update()`, and `finalize()` so callers can feed data in arbitrary-sized chunks — the way real programs actually deliver data.
3. **Validate against NIST test vectors**: Run three Known Answer Tests that confirm your entire implementation is byte-for-byte correct. This sounds simpler than Milestones 2 and 3. You've done the hard cryptographic work. The rest is just wiring. That intuition is dangerously wrong — and understanding *why* it's wrong is the most important thing you'll learn in this milestone.

The Misconception Worth Shattering

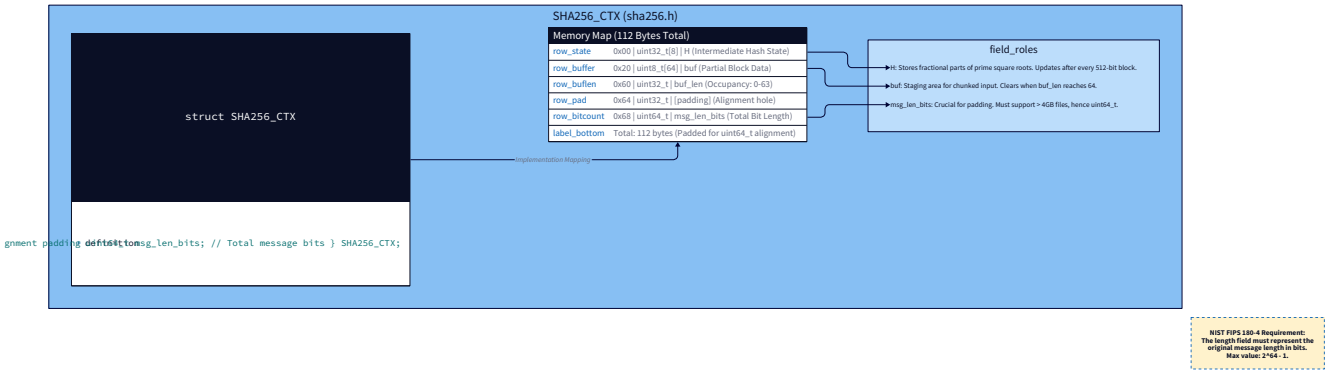
Here's what developers assume once the single-block compression function works:

> *"The streaming API is just a convenience wrapper. I'll keep the H values around between calls, call compress() whenever I have data, and finalize() pads whatever's left. State management between calls is obvious — the H array persists across update() calls. This is an afternoon of glue code."* Let's trace exactly why this breaks. Suppose a caller feeds data in 100-byte chunks to hash a 200-byte message. Your `update()` receives the first 100 bytes. A 512-bit block is 64 bytes. So after the first `update()`:

- You have 100 bytes of input
- You can process one complete 64-byte block (bytes 0–63)
- You have **36 bytes left over** — a partial block that must be buffered The second `update()` receives another 100 bytes:
- You now have 36 (buffered) + 100 (new) = 136 bytes
- You can process two complete 64-byte blocks (bytes 64–191 of the original message)
- You have **8 bytes left over** in the buffer When `finalize()` is called:
- You have 8 bytes in the buffer
- You must apply padding: append 0x80, zero-fill, append the 64-bit bit length
- The total message length in bits is `200 × 8 = 1600` — but you never stored the total length in the H state. You must have been tracking it separately in the context
- 8 bytes in the buffer + 1 byte for 0x80 + 55 bytes of zeros = 64 bytes → exactly one more block to compress Now change the chunk size to 64 bytes exactly. First `update()`: 64 bytes → one complete block, compress it, nothing buffered. Second `update()`: 64 bytes → one complete block, compress it, nothing buffered. Third `update()` (the remaining 72 bytes): 72 bytes → one complete block (bytes 128–191), compress it, 8 bytes buffered. `finalize()`: same 8 bytes, same padding. If your implementation produces the same hash regardless of chunk size, it is correct. Change the chunk size again: feed all 200 bytes in one `update()` call. You get three complete blocks (bytes 0–191), with 8 bytes left. `finalize()` pads and compresses the last block. Same result required. **The bug surface is enormous.** Every combination of message length and chunk size is a separate test case. A single off-by-one in your buffer offset tracking will cause failures that only appear with specific chunk sizes — the kind of bug that passes your test suite but silently corrupts hashes in production when data arrives in network-sized packets. There's a second, even more dangerous failure mode: **state contamination between invocations**. If H[0]..H[7] retain values from a previous hash computation, your next hash is not SHA-256(message) — it's SHA-256 with a corrupted initial state that no other implementation in the world will reproduce. Worse, if two different messages share some prefix bytes that happened to fill the buffer in a previous call, those buffer bytes are silently prepended to the next message. You've invented a hash function that is incompatible with SHA-256 and depends on call order. This is the same class of vulnerability as **IV reuse** in symmetric encryption — we'll return to this in the Knowledge Cascade. This is where real-world SHA-256 bugs live. Let's build it correctly from the start.

Designing the Context Structure

A streaming hash API requires a **context** — a struct that holds all mutable state between `init()`, `update()`, and `finalize()` calls. Everything your hash computation needs to remember between calls must live in this struct.



What state do you need to track? **The hash state** (`H[8]`): Eight 32-bit words, updated after each block is compressed. This starts as the FIPS 180-4 initial values and accumulates the running hash. **The message buffer** (`buf[64]`): Up to 63 bytes of input that haven't yet formed a complete 512-bit block. When `update()` receives data, some bytes may fill this buffer, some may trigger compression of complete blocks, and the remainder goes back into the buffer. **The buffer offset** (`buf_len`): How many bytes are currently in the buffer (0 to 63). This tells you where to write next into the buffer and how many bytes are available. **The total**

message length (`msg_len_bits`): The total number of bits in the complete message seen so far across all `update()` calls. This is written into the final padding block's 64-bit length field. You must track this cumulatively — the compression function doesn't record it.

```c
/* sha256.h — SHA-256 streaming API */

#ifndef SHA256_H

#define SHA256_H

#include <stdint.h>

#include <stddef.h>

#define SHA256_BLOCK_SIZE  64    /* 512 bits in bytes */

#define SHA256_DIGEST_SIZE 32    /* 256 bits in bytes */

/*

 * SHA256_CTX — All mutable state for an in-progress SHA-256 computation.

 *

 * Holds everything needed to continue hashing after an update() call returns.

 * Must be initialized with sha256_init() before use.

 * Must not be reused after sha256_finalize() without calling sha256_init() again.

 */

typedef struct {

    uint32_t H[8];                  /* Running hash state (8 × 32-bit words)    */

    uint8_t  buf[SHA256_BLOCK_SIZE]; /* Partial block buffer (0–63 bytes)         */

    uint32_t buf_len;               /* Bytes currently in buf (0–63)             */

    uint64_t msg_len_bits;          /* Total message length in bits (cumulative) */

} SHA256_CTX;

/* API */

void sha256_init(SHA256_CTX *ctx);

void sha256_update(SHA256_CTX *ctx, const uint8_t *data, size_t len);

void sha256_finalize(SHA256_CTX *ctx, uint8_t digest[SHA256_DIGEST_SIZE]);

/* Convenience: hash a complete message in one call */

void sha256(const uint8_t *data, size_t len, uint8_t digest[SHA256_DIGEST_SIZE]);

#endif /* SHA256_H */
```

Why `uint64_t` for `msg_len_bits`? Because the SHA-256 specification supports messages up to 2^64 - 1 bits long (the 64-bit length field in the padding encodes this). For a 1 GB message, the byte count is about 10^9 — fits in 32 bits — but the bit count is 8 × 10^9, which requires more than 32 bits (max 32-bit value is ~4.3 × 10^9). Using `uint64_t` ensures correctness for all valid message sizes. Why `uint32_t buf_len` instead of `size_t`? Because `buf_len` is always in the range 0–63. `uint32_t` makes this range explicit and avoids any question about signed/unsigned issues on 32-bit platforms where `size_t` might not be 64-bit. Either works; this choice is defensive clarity.

Implementing `sha256_init()`

Initialization is the simplest function in the API, but it is the most security-critical. Every `sha256_init()` call must produce a clean state, completely independent of any previous computation that used the same `SHA256_CTX` memory.

```c
/* SHA-256 initial hash values — FIPS 180-4 Section 5.3.3
 * Fractional parts of sqrt(2), sqrt(3), sqrt(5), sqrt(7),
 * sqrt(11), sqrt(13), sqrt(17), sqrt(19), truncated to 32 bits.
 */
static const uint32_t SHA256_H_INIT[8] = {
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19,
};
/*
 * sha256_init — Initialize a SHA-256 context for a new hash computation.
 *
 * Call this before the first sha256_update() call. Call it again to reuse
 * a context for a new message — do NOT start a new hash without calling
 * sha256_init() first, or the previous hash's state will corrupt the new one.
 */
void sha256_init(SHA256_CTX *ctx) {
    /* Load the FIPS 180-4 initial hash values */
    for (int i = 0; i < 8; i++) {
        ctx->H[i] = SHA256_H_INIT[i];
    }
    /* Clear the partial block buffer */
    memset(ctx->buf, 0, SHA256_BLOCK_SIZE);
    /* No bytes buffered yet */
    ctx->buf_len = 0;
    /* No bits of message seen yet */
    ctx->msg_len_bits = 0;
}
```

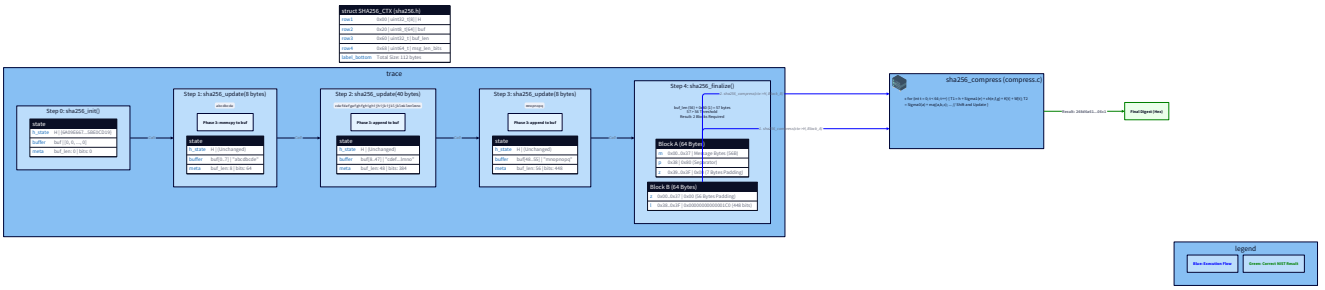
You could use `memcpy` instead of the loop for the H values — both are correct. The explicit loop makes it obvious exactly which values are being loaded and from where. In production code, `memcpy(ctx->H, SHA256_H_INIT, sizeof(SHA256_H_INIT))` is idiomatic and equally correct. The `memset` on `buf` is not strictly required (stale bytes in the buffer only matter if they're in positions 0..buf_len-1, and buf_len starts at 0, so no stale bytes are in play). But clearing the buffer is defensive hygiene: if there was a previous computation using this context struct, sensitive message bytes from that computation may still be in `buf`. Clearing them prevents sensitive data from lingering in memory beyond its useful life — a practice we'll revisit in the security discussion at the end of this milestone.

Implementing `sha256_update()`

This is the most complex function in the API. Every correctness property of your streaming hash depends on getting the buffer management right.

The logic has three phases:

1. If there's already data in the buffer, try to fill it to a complete block with the new input. If you fill it, compress the block and clear the buffer.
2. While the remaining input has enough bytes for complete blocks, compress them directly (without going through the buffer).
3. Copy whatever remains into the buffer. This design ensures the buffer always holds 0–63 bytes and is never compressed until it contains exactly 64 bytes.



```c
/*
 * sha256_update — Feed more data into an in-progress SHA-256 computation.
 *
 * ctx:  initialized SHA-256 context (from sha256_init)
 * data: pointer to the input bytes
 * len:  number of bytes to hash
 *
 * May be called multiple times with arbitrary chunk sizes.
 * Each call updates ctx in-place.
 */
void sha256_update(SHA256_CTX *ctx, const uint8_t *data, size_t len) {
    uint32_t W[64]; /* Message schedule workspace — allocated on the stack */
    /* Track the total message length for the final padding */
    ctx->msg_len_bits += (uint64_t)len * 8;
    /*
     * Note: cast len to uint64_t BEFORE multiplying by 8.
     * If len is 0x20000000 (512MB) and size_t is 32 bits,
     * 'len * 8' would overflow. The cast prevents this.
     * On 64-bit systems where size_t is 64 bits this is less
     * of a concern, but the cast is correct on all platforms.
     */
    /* ── Phase 1: Fill the partial block buffer if it has data ── */
    if (ctx->buf_len > 0) {
        /* How many bytes do we need to fill the buffer to 64 bytes? */
        uint32_t space = SHA256_BLOCK_SIZE - ctx->buf_len;
        uint32_t take  = (len < space) ? (uint32_t)len : space;
        memcpy(ctx->buf + ctx->buf_len, data, take);
        ctx->buf_len += take;
        data         += take;
        len          -= take;
        /* If the buffer is now full, compress it */
        if (ctx->buf_len == SHA256_BLOCK_SIZE) {
            sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
            ctx->buf_len = 0;
            /* No need to memset buf here — buf_len=0 marks it as logically empty */
        }
    }
    /* ── Phase 2: Compress complete blocks directly from input ── */
    while (len >= SHA256_BLOCK_SIZE) {
        sha256_compress(ctx->H, (const SHA256_Block *)data, W);
```

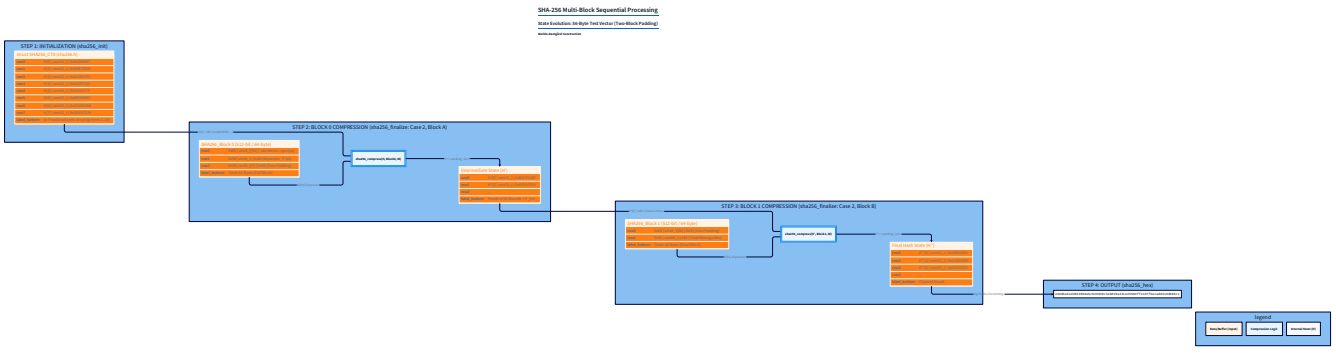
```
        data += SHA256_BLOCK_SIZE;
        len  -= SHA256_BLOCK_SIZE;
    }
    /* ── Phase 3: Buffer any remaining bytes ── */
    if (len > 0) {
        memcpy(ctx->buf + ctx->buf_len, data, len);
        ctx->buf_len += (uint32_t)len;
    }
}
```

Let's trace each phase with the 200-byte / 100-byte-chunk example from earlier: **First `update(100 bytes)`:**

- `msg_len_bits` = 800
- Phase 1: `buf_len` = 0, so skip Phase 1
- Phase 2: 100 ≥ 64, compress bytes 0–63 directly; len = 36, data advances to byte 64; 36 < 64, stop
- Phase 3: copy bytes 64–99 into buf; `buf_len` = 36 **Second `update(100 bytes)`:**
- `msg_len_bits` = 1600
- Phase 1: `buf_len` = 36; `space` = 64 - 36 = 28; `take` = min(100, 28) = 28; copy 28 bytes into buf (filling it); `buf_len` = 64; compress buf (bytes 64–127 of original); `buf_len` = 0; len = 72; data advances 28 bytes
- Phase 2: 72 ≥ 64, compress next 64 bytes (bytes 128–191); len = 8; 8 < 64, stop
- Phase 3: copy 8 bytes into buf; `buf_len` = 8 Result: 3 blocks compressed (bytes 0–191), 8 bytes buffered. ✓ `finalize()` (coming next): 8 bytes in buffer, padding them to a complete block. The key design invariant: **after every `update()` call, `buf_len` is in the range [0, 63]**. Never 64. If it ever reaches 64, the block gets compressed immediately and `buf_len` resets to 0.

Implementing `sha256_finalize()`

Finalization applies padding to whatever remains in the buffer and produces the digest. This function is essentially a specialized version of your Milestone 1 padding logic — but applied to the *buffered remainder* of the message, not the whole message from scratch.



The challenge: you don't have a separate "padded message" to work with. You have a buffer that's 0–63 bytes full, and you need to compress exactly one or two more blocks that together contain:

1. The remaining message bytes (already in the buffer)
2. The 0x80 separator byte
3. Zero-padding
4. The 64-bit big-endian total message length The same boundary case from Milestone 1 reappears: if the buffer contains 56 or more bytes, there's not enough room for the 0x80, the zero-padding, and the 8-byte length field in one block — you need two additional blocks.

```c
/*
 * sha256_finalize — Complete the hash computation and write the digest.
 *
 * ctx:    the in-progress SHA-256 context (modified in place)
 * digest: output buffer receiving the 32-byte (256-bit) SHA-256 digest
 *
 * After this call, ctx is in an indeterminate state. Call sha256_init()
 * before reusing ctx for a new hash computation.
 */
void sha256_finalize(SHA256_CTX *ctx, uint8_t digest[SHA256_DIGEST_SIZE]) {
    uint32_t W[64];
    /*
     * We need to pad ctx->buf[0..buf_len-1] with:
     *   byte 0x80 at position buf_len
     *   zero bytes from buf_len+1 to 54 (or to the end of the block)
     *   8-byte big-endian total message length at positions 56–63
     *       of the LAST padding block
     *
     * If buf_len <= 55: everything fits in one more block (buf + padding + length)
     * If buf_len >= 56: we need two more blocks
     *     Block A: buf + 0x80 + zeros to fill 64 bytes
     *     Block B: 56 zero bytes + 8-byte length
     */
    /* ── Append the 0x80 separator byte ── */
    ctx->buf[ctx->buf_len] = 0x80;
    ctx->buf_len++;
    if (ctx->buf_len <= 56) {
        /* ── Case 1: One padding block ── */
        /* Zero-fill from buf_len to position 55 (inclusive) */
        memset(ctx->buf + ctx->buf_len, 0x00, 56 - ctx->buf_len);
        /* Write 64-bit big-endian total message length at bytes 56–63 */
        ctx->buf[56] = (uint8_t)(ctx->msg_len_bits >> 56);
        ctx->buf[57] = (uint8_t)(ctx->msg_len_bits >> 48);
        ctx->buf[58] = (uint8_t)(ctx->msg_len_bits >> 40);
        ctx->buf[59] = (uint8_t)(ctx->msg_len_bits >> 32);
        ctx->buf[60] = (uint8_t)(ctx->msg_len_bits >> 24);
        ctx->buf[61] = (uint8_t)(ctx->msg_len_bits >> 16);
        ctx->buf[62] = (uint8_t)(ctx->msg_len_bits >>  8);
        ctx->buf[63] = (uint8_t)(ctx->msg_len_bits >>  0);
        /* Compress the single padding block */
```

```
        sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
    } else {
        /* ── Case 2: Two padding blocks ── */
        /* Block A: zero-fill the remainder of the current buffer */
        memset(ctx->buf + ctx->buf_len, 0x00, SHA256_BLOCK_SIZE - ctx->buf_len);
        sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
        /* Block B: 56 zero bytes + 8-byte length */
        memset(ctx->buf, 0x00, 56);  /* Reuse ctx->buf for Block B */
        ctx->buf[56] = (uint8_t)(ctx->msg_len_bits >> 56);
        ctx->buf[57] = (uint8_t)(ctx->msg_len_bits >> 48);
        ctx->buf[58] = (uint8_t)(ctx->msg_len_bits >> 40);
        ctx->buf[59] = (uint8_t)(ctx->msg_len_bits >> 32);
        ctx->buf[60] = (uint8_t)(ctx->msg_len_bits >> 24);
        ctx->buf[61] = (uint8_t)(ctx->msg_len_bits >> 16);
        ctx->buf[62] = (uint8_t)(ctx->msg_len_bits >>  8);
        ctx->buf[63] = (uint8_t)(ctx->msg_len_bits >>  0);
        sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
    }
    /* ── Format the digest: H[0]..H[7] as big-endian bytes ── */
    for (int i = 0; i < 8; i++) {
        digest[i * 4 + 0] = (uint8_t)(ctx->H[i] >> 24);
        digest[i * 4 + 1] = (uint8_t)(ctx->H[i] >> 16);
        digest[i * 4 + 2] = (uint8_t)(ctx->H[i] >>  8);
        digest[i * 4 + 3] = (uint8_t)(ctx->H[i] >>  0);
    }
    /*
     * Security: clear the internal buffer to prevent message bytes
     * from lingering in memory after the hash is produced.
     * See the security discussion later in this milestone.
     */
    memset(ctx->buf, 0, SHA256_BLOCK_SIZE);
    memset(W, 0, sizeof(W));
}
```

The boundary condition `buf_len <= 56` deserves careful attention. After appending 0x80, `buf_len` has been incremented. So:

- If `buf_len` after the increment is 1 (empty buffer), the 0x80 went into position 0, and we have positions 1–55 for zeros, then positions 56–63 for the length. Fits in one block. ✓
- If `buf_len` after the increment is 56, the 0x80 went into position 55, positions 56–63 are the length field. No zeros needed. Still fits. ✓
- If `buf_len` after the increment is 57, the 0x80 went into position 56, which is where the length field starts. Doesn't fit. Two blocks needed. ✓ The boundary is `buf_len <= 56` (after increment), which means the original buffer had at most 55 bytes — exactly matching the Milestone 1 analysis.

Formatting the Hex Output

 H[0..7] to Hex String — Final Output Encoding

The digest bytes must be formatted as a 64-character lowercase hexadecimal string. Each of the 32 digest bytes maps to exactly two hex characters (`00` through `ff`).

```c
/*
 * sha256_hex — Convert a 32-byte binary digest to a 64-character hex string.
 *
 * digest: the 32-byte binary SHA-256 output (from sha256_finalize)
 * hex:    output buffer of at least 65 bytes (64 hex chars + null terminator)
 */
void sha256_hex(const uint8_t digest[SHA256_DIGEST_SIZE], char hex[65]) {
    static const char hex_chars[] = "0123456789abcdef";
    for (int i = 0; i < SHA256_DIGEST_SIZE; i++) {
        hex[i * 2 + 0] = hex_chars[(digest[i] >> 4) & 0xF]; /* High nibble */
        hex[i * 2 + 1] = hex_chars[(digest[i] >> 0) & 0xF]; /* Low nibble  */
    }
    hex[64] = '\0';  /* Null terminator — makes it a valid C string */
}
```

A nibble is half a byte — 4 bits. Each byte has a high nibble (the upper 4 bits) and a low nibble (the lower 4 bits). The high nibble `(digest[i] >> 4) & 0xF` selects bits 7–4, mapping them to a value 0–15. The low nibble `(digest[i] >> 0) & 0xF` — or just `digest[i] & 0xF` — selects bits 3–0. The `hex_chars` lookup table converts 0–15 to their ASCII hex characters: 0→'0', 10→'a', 15→'f'. Using lowercase characters (`"0123456789abcdef"` rather than `"0123456789ABCDEF"`) matches the NIST test vector format and the convention used by virtually every Unix/Linux tool (`sha256sum` , `openssl dgst` , `python3 hashlib`).

The One-Call Convenience Function

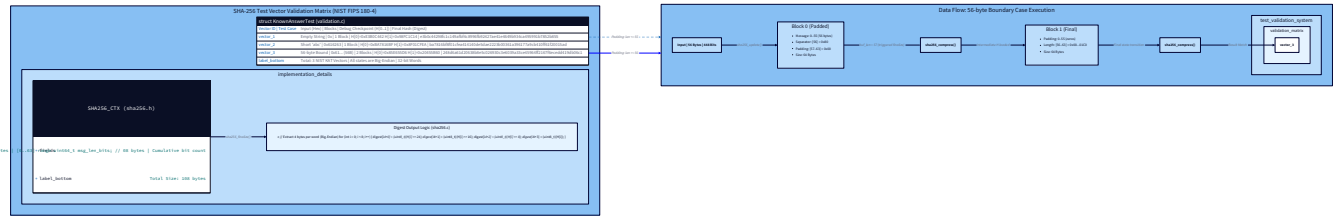
Most callers don't need streaming — they have all their data in memory at once. Provide a convenience function that calls `init` , a single `update` , and `finalize` :

```c
/*
 * sha256 — Hash a complete message in one call.
 *
 * data:   pointer to the message bytes (NULL is allowed when len == 0)
 * len:    length of the message in bytes
 * digest: output buffer receiving the 32-byte SHA-256 digest
 *
 * Equivalent to: sha256_init + sha256_update + sha256_finalize.
 * The caller is responsible for converting digest to hex if needed.
 */
void sha256(const uint8_t *data, size_t len, uint8_t digest[SHA256_DIGEST_SIZE]) {
    SHA256_CTX ctx;
    sha256_init(&ctx);
    if (len > 0) {
        sha256_update(&ctx, data, len);
    }
    sha256_finalize(&ctx, digest);
}
```

This is not a shortcut that bypasses the streaming API — it *uses* the streaming API. This means the one-call function and the multi-call streaming function are guaranteed to produce identical results for the same input, because they execute the same code path.

Validating Against NIST Test Vectors

This is the moment your entire implementation is put to the test. NIST Known Answer Tests (KATs) are authoritative — they define what SHA-256 *is*. If your output matches these test vectors, your implementation is correct. If not, there's a bug somewhere in the pipeline.



Three test vectors from NIST FIPS 180-4 and the associated test vector document:

Input	Expected Output
`""` (empty string, 0 bytes)	`e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`
`"abc"` (3 bytes)	`ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`
`"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"` (448 bits, 56 bytes)	`248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1`
The third test vector is deliberately chosen to be exactly 56 bytes long. Recall from Milestone 1: a 56-byte message requires two padded blocks. This test vector exercises the two-block padding code path and the multi-block sequential processing in one shot.	

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>

/* ─────────────────────────────────────────────────────────────── */
 * Helper: compare expected hex string with binary digest
 * ─────────────────────────────────────────────────────────────── */
static int check_digest(const uint8_t *digest, const char *expected_hex,
                        const char *test_name) {
    char actual_hex[65];
    sha256_hex(digest, actual_hex);
    if (strcmp(actual_hex, expected_hex) == 0) {
        printf("PASS [%s]\n  %s\n", test_name, actual_hex);
        return 1;
    } else {
        printf("FAIL [%s]\n  got:      %s\n  expected: %s\n",
               test_name, actual_hex, expected_hex);
        return 0;
    }
}

/* ─────────────────────────────────────────────────────────────── */
 * Test 1: Empty string
 * Expected: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
 * ─────────────────────────────────────────────────────────────── */
void test_empty_string(void) {
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256(NULL, 0, digest);
    check_digest(digest,
        "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
        "SHA-256(\"\")");
}

/* ─────────────────────────────────────────────────────────────── */
 * Test 2: "abc"
 * Expected: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
 * ─────────────────────────────────────────────────────────────── */
void test_abc(void) {
    const uint8_t msg[] = "abc";
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256(msg, 3, digest);
    check_digest(digest,
```

```
        "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad",
        "SHA-256(\"abc\")");
}
/* ──────────────────────────────────────────────────────────────── */
 * Test 3: 56-byte string (two-block padding path)
 * "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
 * Expected: 248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1
 * ──────────────────────────────────────────────────────────────── */
void test_two_block_message(void) {
    const char *msg = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256((const uint8_t *)msg, strlen(msg), digest);
    check_digest(digest,
        "248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1",
        "SHA-256(56-byte string)");
}
/* ──────────────────────────────────────────────────────────────── */
 * Test 4: State independence — same context, two independent hashes
 * Both calls must produce the same "abc" hash.
 * ──────────────────────────────────────────────────────────────── */
void test_state_independence(void) {
    SHA256_CTX ctx;
    uint8_t digest1[SHA256_DIGEST_SIZE];
    uint8_t digest2[SHA256_DIGEST_SIZE];
    /* First hash */
    sha256_init(&ctx);
    sha256_update(&ctx, (const uint8_t *)"abc", 3);
    sha256_finalize(&ctx, digest1);
    /* Second hash — must reinitialize before reuse */
    sha256_init(&ctx);
    sha256_update(&ctx, (const uint8_t *)"abc", 3);
    sha256_finalize(&ctx, digest2);
    if (memcmp(digest1, digest2, SHA256_DIGEST_SIZE) == 0) {
        printf("PASS [state_independence]: identical results for two independent calls\n");
    } else {
        printf("FAIL [state_independence]: results differ — state was not reset\n");
    }
}
/* ──────────────────────────────────────────────────────────────── */
 * Test 5: Streaming — "abc" fed one byte at a time
```

```c
 * Must match the same result as feeding all 3 bytes at once.
 * ──────────────────────────────────────────────────────────── */
void test_streaming_one_byte_at_a_time(void) {
    SHA256_CTX ctx;
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256_init(&ctx);
    sha256_update(&ctx, (const uint8_t *)"a", 1);
    sha256_update(&ctx, (const uint8_t *)"b", 1);
    sha256_update(&ctx, (const uint8_t *)"c", 1);
    sha256_finalize(&ctx, digest);
    check_digest(digest,
        "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad",
        "SHA-256(\"abc\") via 3×1-byte updates");
}
/* ──────────────────────────────────────────────────────────────
 * Test 6: Streaming — 56-byte string fed in 7-byte chunks
 * Exercises the partial block buffer fill-and-compress logic.
 * ──────────────────────────────────────────────────────────── */
void test_streaming_chunked(void) {
    const char *msg = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";
    size_t total_len = strlen(msg);
    size_t chunk_size = 7;
    SHA256_CTX ctx;
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256_init(&ctx);
    for (size_t offset = 0; offset < total_len; offset += chunk_size) {
        size_t remaining = total_len - offset;
        size_t this_chunk = (remaining < chunk_size) ? remaining : chunk_size;
        sha256_update(&ctx, (const uint8_t *)msg + offset, this_chunk);
    }
    sha256_finalize(&ctx, digest);
    check_digest(digest,
        "248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1",
        "SHA-256(56-byte string) via 7-byte chunks");
}
/* ──────────────────────────────────────────────────────────────
 * Test 7: Streaming — exactly one block at a time (64-byte chunks)
 * Tests the Phase 2 fast path where blocks are compressed directly.
 * ──────────────────────────────────────────────────────────── */
void test_streaming_full_block_chunks(void) {
```

```
    const char *msg = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";
    size_t total_len = strlen(msg);
    SHA256_CTX ctx;
    uint8_t digest[SHA256_DIGEST_SIZE];
    sha256_init(&ctx);
    /* Feed 56 bytes in one shot — exactly the message length */
    sha256_update(&ctx, (const uint8_t *)msg, total_len);
    sha256_finalize(&ctx, digest);
    check_digest(digest,
        "248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1",
        "SHA-256(56-byte string) via single update");
}
int main(void) {
    printf("=== SHA-256 Final Output and Validation Tests ===\n\n");
    test_empty_string();
    test_abc();
    test_two_block_message();
    test_state_independence();
    test_streaming_one_byte_at_a_time();
    test_streaming_chunked();
    test_streaming_full_block_chunks();
    printf("\n=== All tests complete ===\n");
    return 0;
}
```

Compile with:

```
gcc -Wall -Wextra -o sha256_test sha256.c sha256_test.c && ./sha256_test
```
BASH

If any test fails, here's the debugging strategy:

1. **`SHA-256("")` fails**: The `sha256_finalize()` padding path for an empty buffer (buf_len = 0) is wrong. Check that after appending 0x80, buf_len becomes 1, which is ≤ 56, so one padding block is used.
2. **`SHA-256("abc")` fails but was passing in Milestone 3's direct compress test**: The init, update, or finalize functions have a bug independent of the compression function. Check that H is loaded from `SHA256_H_INIT` in `sha256_init()`.
3. **`SHA-256(56-byte string)` fails**: This is the two-block padding path. Check the `buf_len >= 56` branch in `sha256_finalize()`. After appending 0x80 to a 56-byte buffer, `buf_len` becomes 57, which triggers Block A + Block B.
4. **`state_independence` fails**: `sha256_init()` is not fully resetting the state — check that `buf_len` and `msg_len_bits` are both reset to 0, and that `H` is reloaded from `SHA256_H_INIT`, not left from the previous computation.
5. **Streaming tests fail but single-call test passes**: The buffer management in `sha256_update()` has an off-by-one in the `buf_len` tracking. Print `buf_len` after each `update()` call to trace where it goes wrong.

Tracing the 56-Byte Test Vector Manually

Let's trace the most complex test vector to build confidence. The 56-byte input `"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"` exercises the two-block padding path. After calling `sha256_update(ctx, msg, 56)`:

- `msg_len_bits` = 56 × 8 = 448
- Phase 1: `buf_len` = 0, skip
- Phase 2: 56 < 64, no complete blocks, skip
- Phase 3: copy all 56 bytes into buf; `buf_len` = 56 No compression has happened yet — the entire message sits in the buffer. When `sha256_finalize()` is called:
- Append 0x80 to `buf[56]` ; `buf_len` = 57
- 57 > 56 → two-block path **Block A** (first padding block):

```
Bytes  0–55:  the 56 message bytes ('a','b','c','d','b','c','d','e',...)
Byte   56:    0x80
Bytes 57–63:  0x00 (zero-fill)
```

Compress Block A. The message bytes participate in the schedule and compression for this block. **Block B** (second padding block):

```
Bytes  0–55:  0x00 (zero padding)
Bytes 56–63:  0x00 0x00 0x00 0x00 0x00 0x00 0x01 0xC0
```

The length field: 448 decimal = 0x1C0 = `0x00 0x00 0x00 0x00 0x00 0x00 0x01 0xC0` in big-endian. Compress Block B. Final H[0]..H[7] formatted as hex: `248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1`.

The Complete sha256.c

Here is the final, self-contained implementation combining all four milestones:

```c
/* sha256.c — Complete SHA-256 implementation (FIPS 180-4) */

#include "sha256.h"

#include <string.h>

/* ──────────────────────────────────────────────

 * Internal types

 * ────────────────────────────────────────────── */

typedef struct {

    uint8_t bytes[SHA256_BLOCK_SIZE];

} SHA256_Block;

/* ──────────────────────────────────────────────

 * Primitive: right-rotate a 32-bit value

 * ────────────────────────────────────────────── */

static inline uint32_t rotr32(uint32_t x, unsigned int n) {

    return (x >> n) | (x << (32u - n));

}

/* ──────────────────────────────────────────────

 * Message schedule functions (lowercase sigma)

 * ────────────────────────────────────────────── */

static inline uint32_t sigma0(uint32_t x) {

    return rotr32(x, 7) ^ rotr32(x, 18) ^ (x >> 3);

}

static inline uint32_t sigma1(uint32_t x) {

    return rotr32(x, 17) ^ rotr32(x, 19) ^ (x >> 10);

}

/* ──────────────────────────────────────────────

 * Compression function helpers (uppercase Sigma, Ch, Maj)

 * ────────────────────────────────────────────── */

static inline uint32_t Sigma0(uint32_t x) {

    return rotr32(x, 2) ^ rotr32(x, 13) ^ rotr32(x, 22);

}

static inline uint32_t Sigma1(uint32_t x) {

    return rotr32(x, 6) ^ rotr32(x, 11) ^ rotr32(x, 25);

}

static inline uint32_t ch(uint32_t x, uint32_t y, uint32_t z) {

    return (x & y) ^ (~x & z);

}

static inline uint32_t maj(uint32_t x, uint32_t y, uint32_t z) {

    return (x & y) ^ (x & z) ^ (y & z);

}

/* ──────────────────────────────────────────────
```

```
 * Round constants — FIPS 180-4 Section 4.2.2
 * ──────────────────────────────────────────── */
static const uint32_t K[64] = {
    0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,
    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,
    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,
    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,
    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,
    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,
    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,
    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,
    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,
    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2,
};
/* ──────────────────────────────────────────── */
 * Initial hash values — FIPS 180-4 Section 5.3.3
 * ──────────────────────────────────────────── */
static const uint32_t SHA256_H_INIT[8] = {
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19,
};
/* ──────────────────────────────────────────── */
 * Internal: load a big-endian 32-bit word from 4 bytes
 * ──────────────────────────────────────────── */
static inline uint32_t load_u32_be(const uint8_t *p) {
    return ((uint32_t)p[0] << 24) | ((uint32_t)p[1] << 16)
         | ((uint32_t)p[2] <<  8) | ((uint32_t)p[3] <<  0);
}
/* ──────────────────────────────────────────── */
 * Internal: compress one 512-bit block
 * ──────────────────────────────────────────── */
static void sha256_compress(uint32_t H[8], const SHA256_Block *block, uint32_t W[64]) {
    /* Build message schedule */
    for (int t = 0; t < 16; t++) {
```

```
        W[t] = load_u32_be(block->bytes + t * 4);
    }
    for (int t = 16; t < 64; t++) {
        W[t] = sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16];
    }
    /* Initialize working variables */
    uint32_t a = H[0], b = H[1], c = H[2], d = H[3];
    uint32_t e = H[4], f = H[5], g = H[6], h = H[7];
    /* 64 rounds */
    for (int t = 0; t < 64; t++) {
        uint32_t T1 = h + Sigma1(e) + ch(e, f, g) + K[t] + W[t];
        uint32_t T2 = Sigma0(a) + maj(a, b, c);
        h = g; g = f; f = e; e = d + T1;
        d = c; c = b; b = a; a = T1 + T2;
    }
    /* Update hash state */
    H[0] += a; H[1] += b; H[2] += c; H[3] += d;
    H[4] += e; H[5] += f; H[6] += g; H[7] += h;
}
/* ─────────────────────────────────────────────
 * Public API
 * ─────────────────────────────────────────── */
void sha256_init(SHA256_CTX *ctx) {
    for (int i = 0; i < 8; i++) ctx->H[i] = SHA256_H_INIT[i];
    memset(ctx->buf, 0, SHA256_BLOCK_SIZE);
    ctx->buf_len      = 0;
    ctx->msg_len_bits = 0;
}
void sha256_update(SHA256_CTX *ctx, const uint8_t *data, size_t len) {
    uint32_t W[64];
    ctx->msg_len_bits += (uint64_t)len * 8;
    if (ctx->buf_len > 0) {
        uint32_t space = SHA256_BLOCK_SIZE - ctx->buf_len;
        uint32_t take  = (len < space) ? (uint32_t)len : space;
        memcpy(ctx->buf + ctx->buf_len, data, take);
        ctx->buf_len += take;
        data += take; len -= take;
        if (ctx->buf_len == SHA256_BLOCK_SIZE) {
            sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
            ctx->buf_len = 0;
```

```
        }
    }
    while (len >= SHA256_BLOCK_SIZE) {
        sha256_compress(ctx->H, (const SHA256_Block *)data, W);
        data += SHA256_BLOCK_SIZE;
        len  -= SHA256_BLOCK_SIZE;
    }
    if (len > 0) {
        memcpy(ctx->buf + ctx->buf_len, data, len);
        ctx->buf_len += (uint32_t)len;
    }
}
void sha256_finalize(SHA256_CTX *ctx, uint8_t digest[SHA256_DIGEST_SIZE]) {
    uint32_t W[64];
    ctx->buf[ctx->buf_len] = 0x80;
    ctx->buf_len++;
    if (ctx->buf_len <= 56) {
        memset(ctx->buf + ctx->buf_len, 0x00, 56 - ctx->buf_len);
    } else {
        memset(ctx->buf + ctx->buf_len, 0x00, SHA256_BLOCK_SIZE - ctx->buf_len);
        sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
        memset(ctx->buf, 0x00, 56);
    }
    ctx->buf[56] = (uint8_t)(ctx->msg_len_bits >> 56);
    ctx->buf[57] = (uint8_t)(ctx->msg_len_bits >> 48);
    ctx->buf[58] = (uint8_t)(ctx->msg_len_bits >> 40);
    ctx->buf[59] = (uint8_t)(ctx->msg_len_bits >> 32);
    ctx->buf[60] = (uint8_t)(ctx->msg_len_bits >> 24);
    ctx->buf[61] = (uint8_t)(ctx->msg_len_bits >> 16);
    ctx->buf[62] = (uint8_t)(ctx->msg_len_bits >>  8);
    ctx->buf[63] = (uint8_t)(ctx->msg_len_bits >>  0);
    sha256_compress(ctx->H, (const SHA256_Block *)ctx->buf, W);
    for (int i = 0; i < 8; i++) {
        digest[i * 4 + 0] = (uint8_t)(ctx->H[i] >> 24);
        digest[i * 4 + 1] = (uint8_t)(ctx->H[i] >> 16);
        digest[i * 4 + 2] = (uint8_t)(ctx->H[i] >>  8);
        digest[i * 4 + 3] = (uint8_t)(ctx->H[i] >>  0);
    }
    /* Security: zero sensitive data from memory */
    memset(ctx->buf, 0, SHA256_BLOCK_SIZE);
```

```
    memset(W, 0, sizeof(W));
}
void sha256(const uint8_t *data, size_t len, uint8_t digest[SHA256_DIGEST_SIZE]) {
    SHA256_CTX ctx;
    sha256_init(&ctx);
    if (len > 0) sha256_update(&ctx, data, len);
    sha256_finalize(&ctx, digest);
}
void sha256_hex(const uint8_t digest[SHA256_DIGEST_SIZE], char hex[65]) {
    static const char h[] = "0123456789abcdef";
    for (int i = 0; i < SHA256_DIGEST_SIZE; i++) {
        hex[i * 2 + 0] = h[digest[i] >> 4];
        hex[i * 2 + 1] = h[digest[i] & 0xF];
    }
    hex[64] = '\0';
}
```

The Init/Update/Finalize Pattern: A Universal API

The API shape you've just built — `init()`, `update()`, `finalize()` — is not specific to SHA-256. It is **the universal API pattern for all streaming cryptographic operations**. Understanding it once means you immediately understand the interface to every cryptographic library you will ever use. Here's the same pattern applied across the cryptographic stack:

Operation	Init	Update	Finalize/Final
SHA-256 (this project)	`sha256_init(&ctx)`	`sha256_update(&ctx, data, len)`	`sha256_finalize(&ctx, digest)`
OpenSSL SHA-256	`SHA256_Init(&ctx)`	`SHA256_Update(&ctx, data, len)`	`SHA256_Final(digest, &ctx)`
HMAC-SHA-256	`HMAC_CTX_new(); HMAC_Init_ex(...)`	`HMAC_Update(ctx, data, len)`	`HMAC_Final(ctx, digest, &len)`
AES-GCM encryption	`EVP_EncryptInit_ex(...)`	`EVP_EncryptUpdate(...)`	`EVP_EncryptFinal_ex(...)`
BLAKE3	`blake3_hasher_init(&h)`	`blake3_hasher_update(&h, data, len)`	`blake3_hasher_finalize(&h, out, len)`
Python hashlib	`h = hashlib.sha256()`	`h.update(data)`	`h.digest()`
Go crypto/sha256	`h := sha256.New()`	`h.Write(data)`	`h.Sum(nil)`
Every single one of these is the same three-phase state machine:			
Init: Allocate a context, populate it with initial constants, zero out buffers and counters. The context is now ready to receive data. This is the "tape at the start" state.			
Update (called zero or more times): Absorb more data, buffering partial blocks and compressing complete blocks. The context accumulates state. Callers may call update as many times as needed with chunks of any size — this is what makes the API composable with streaming data sources (network sockets, file reads, database queries).			
Finalize (called exactly once): Apply any finalization logic (padding for hashes, authentication tag generation for AEAD modes, key derivation output for KDFs), write the result, and invalidate or clean the context. After finalize, the context must not be used for further updates.			

Why is this pattern universal? Because it perfectly matches how data arrives in real systems. Files are read in chunks. Network packets arrive sequentially. Database results come row by row. A cryptographic library that required the entire input to be in memory simultaneously would be unusable for hashing a 10 GB file or computing an HMAC over a streaming network connection. The init/update/finalize pattern decouples the *arrival* of data from the *consumption* of data, which is the fundamental engineering challenge in any streaming computation. When you call `git log --oneline` and Git computes the SHA-1 hash of a commit object, it uses the streaming API internally — it hashes the header, then the content, then the parent references, one piece at a time. When your browser downloads a file and verifies its SHA-256 hash,

it uses the streaming API — it hashes each network packet as it arrives, never needing the whole file in memory. The pattern you built today is the same one powering those systems.

IV Reuse: When Forgetting to Reset Destroys Security

The `sha256_init()` call that resets H[0]..H[7] to the FIPS constants might feel like boilerplate. It's not. Forgetting it — or forgetting to call it between hash computations — is a real vulnerability pattern with a name: **initialization vector (IV) reuse**. In SHA-256 hashing, the initial hash values are the IV — the starting state of the computation. If two messages are hashed using the same modified starting state (because H was not reset), the results are:

- Not SHA-256 of either message (incompatible with all other implementations)
- Potentially correlated in predictable ways (the second hash's output depends on the first message's content in a complex, non-cryptographic way) For a plain hash function, IV reuse produces wrong-but-deterministic results that are immediately caught by test vectors. But the same pattern in symmetric encryption is catastrophically exploitable. **AES-GCM IV reuse**: AES-GCM (Galois/Counter Mode) is the dominant AEAD (Authenticated Encryption with Associated Data) mode used in TLS 1.3, QUIC, and most modern protocols. It requires a unique 96-bit nonce (IV) for every encryption operation with the same key. If you encrypt two different messages with the same key and the same nonce:
- An attacker who observes both ciphertexts can XOR them together to obtain `plaintext1 XOR plaintext2` — eliminating the encryption of both messages simultaneously
- The authentication tag is completely broken — the attacker can forge arbitrary ciphertexts This is not theoretical. The WEP (Wired Equivalent Privacy) Wi-Fi security protocol reused IVs after only 2^24 ≈ 16 million packets, and because the protocol used RC4 with IVs as part of the keystream, analysts could recover the WEP key from passively captured traffic. WEP was deprecated in 2003 after this and related vulnerabilities were demonstrated in practice. The `sha256_init()` function you wrote — four lines that reset H, clear the buffer, and zero the counters — is the equivalent of "generate a fresh IV." It's not ceremony. It's the line between security and brokenness. Your `sha256_finalize()` goes one step further by calling `memset(ctx->buf, 0, SHA256_BLOCK_SIZE)` after producing the digest. This zeros any partial message bytes that might still be in the buffer. On a system without memory encryption, those bytes can be read by another process through memory introspection tools, by an attacker who exploits a vulnerability like Heartbleed (the 2014 OpenSSL buffer over-read that leaked memory contents from live TLS connections), or by forensic analysis of a memory dump. Zeroing the buffer after use limits the window in which sensitive data is exposed. This is the entry point to a broader discipline called **constant-time programming** and **secret erasure**, which underpins the security of every HSM (Hardware Security Module) and secure enclave in production use.

Known Answer Tests: The Standard for Cryptographic Validation

The three NIST test vectors you just validated against are examples of **Known Answer Tests** (KATs) — the standard methodology for verifying cryptographic implementations. Understanding KAT methodology transfers directly to validating every other cryptographic primitive you'll implement or use. The principle: for a correctly implemented algorithm, specific inputs must always produce specific outputs. These input-output pairs are computed by a reference implementation (typically NIST's own reference code or a formally verified implementation), published in a standard document, and used as acceptance tests by all subsequent implementations. If your implementation's output matches all published KATs, it is almost certainly correct — because producing the correct output for multiple distinct inputs via a wrong algorithm is computationally infeasible. (A broken implementation that happens to pass all KATs would essentially be a collision in the space of algorithms, which is vanishingly unlikely and would itself be a major research finding.) NIST publishes KAT suites for every standardized algorithm:

- **FIPS 180-4**: SHA-256 test vectors (what you just used)
- **FIPS 197**: AES test vectors (for all key sizes and modes)
- **FIPS 186-5**: ECDSA and EdDSA test vectors
- **FIPS 140-3**: The standard governing cryptographic module testing; all FIPS-compliant libraries must pass KATs at startup The habit of validating against KATs before trusting an implementation is non-negotiable in security engineering. When the Heartbleed vulnerability in OpenSSL was discovered, the heartbeat code that was leaking memory had never been subjected to boundary-condition testing that a systematic KAT methodology would have caught. Test vectors don't catch all bugs — but they catch any implementation that produces wrong output for known inputs, which is the most fundamental class of cryptographic failure. For your SHA-256 implementation, the three test vectors cover:
- Zero-length input (tests the empty-input padding edge case)
- Short input that fits in one block (the common case)
- An input that requires exactly two blocks (tests the boundary that traps most beginners) A more thorough test suite would also include:
- An input of exactly 55 bytes (the maximum single-block message)
- An input of exactly 55 bytes fed one byte at a time (streaming with boundary alignment)
- A 1 MB input to verify performance and multi-block correctness
- The NIST Million-'a' vector: SHA-256 of "aaa...a" repeated 1,000,000 times = `cdc76e5c9914fb9281a1c7e284d73e67f1809a48a497200e046d39ccc7112cd0`

Incremental Hashing in Merkle Trees: Git, IPFS, and Blockchain

The streaming API you built is the foundation of a powerful data structure: the **Merkle tree** (named after cryptographer Ralph Merkle, who described it in 1979). A Merkle tree is a binary tree where every leaf node contains the hash of a data block, and every internal node contains the hash of its children's hashes. The root hash (the "Merkle root") is a single cryptographic fingerprint of the entire dataset.

```
          Root = H(H(H01) || H(H23))
                    /              \
             H01 = H(H0||H1)    H23 = H(H2||H3)
                /        \         /        \
        H0=H(B0)  H1=H(B1)  H2=H(B2)  H3=H(B3)
           |         |         |         |
         Block0    Block1    Block2    Block3
```

In this notation, H is SHA-256 and `||` means concatenation. Git uses a Merkle-like structure for its object model. Every file (blob), directory (tree), and commit is stored as an object whose SHA-1 hash (Git is transitioning to SHA-256) is its identifier. A commit object hashes the commit metadata plus the hash of the root tree object. That tree object hashes directory entries, each of which is the hash of a blob or subtree. If any byte anywhere in the repository changes, every hash on the path from that change to the root changes. This is how `git diff` and `git status` detect changes efficiently: compare hashes, not file contents. Your `sha256_update()` function is precisely how Git computes these hashes — it feeds the object type header, then the object content, through the streaming API. The streaming is necessary because a large file might be hundreds of megabytes; you can't require it all in memory at once. IPFS (InterPlanetary File System) takes the same principle to a distributed system. Files are chunked, each chunk is hashed, and the chunks are arranged into a Merkle DAG (Directed Acyclic Graph — a generalization of a tree). The root hash is a content identifier (CID) that uniquely and verifiably identifies the data. Two nodes on the IPFS network can verify they have the same file by comparing CIDs, and can verify the integrity of individual chunks by checking their hashes against the parent node's hash. Your SHA-256 implementation, called incrementally via the streaming API, is the mechanism that makes this possible. Bitcoin uses a Merkle tree of transaction hashes to construct the block header. When a Bitcoin block is mined, the Merkle root of all transactions in the block is included in the header, and the entire header is double-SHA-256 hashed (SHA-256 applied twice) as the proof-of-work target. If any transaction in the block is modified, the Merkle root changes, the header changes, the hash changes, and the proof of work must be redone — securing the blockchain's immutability. Understanding the streaming SHA-256 API is the prerequisite for understanding how all of these systems compute and verify content hashes. The `sha256_update()` function you built today is the engine underneath Git's entire object model, IPFS's content addressing, and Bitcoin's transaction integrity guarantees.

Security: Clearing Sensitive Data from Memory

Your `sha256_finalize()` calls `memset(ctx->buf, 0, SHA256_BLOCK_SIZE)` after the hash is produced. This is the beginning of a discipline called **sensitive data erasure** that every cryptographic library takes seriously. The problem: after `sha256_finalize()` returns, the `SHA256_CTX` struct might remain in memory — either on the stack (waiting to be overwritten by the next function call) or on the heap (waiting for `free()`). If the message being hashed was sensitive (a password, a private key, a secret token), its bytes are now lingering in `ctx->buf`. Any of the following can expose them:

- A memory corruption vulnerability in your program (a use-after-free or buffer overflow) that lets an attacker read unrelated memory
- A core dump if the program crashes
- A memory analysis tool run by another process (not normally possible due to OS isolation, but possible if the process has debug privileges or if the system is compromised)
- Swap file analysis (if the OS wrote the memory page containing the context to disk) The defense is simple: zero the memory as soon as you no longer need it. Your finalize function already does this for `ctx->buf` and the local `W` array. There is one subtle complication: compilers are smart enough to optimize away

`memset` calls that they determine are "dead writes" — writes to memory that is never read again before the program ends or the variable goes out of scope. Since the entire point of the `memset` is security (not functional correctness), the compiler may legally remove it, leaving sensitive data in memory. The solution used in production cryptographic libraries (OpenSSL, libsodium, BoringSSL) is to use a `memset` variant that the compiler cannot optimize away. The most portable approach in C is:

```c
/*
 * secure_memzero — Zero memory in a way the compiler cannot optimize away.
 *
 * Unlike memset(), this function is guaranteed to execute even if the
 * compiler determines the memory is never read again.
 */
static void secure_memzero(void *ptr, size_t len) {
    volatile uint8_t *p = (volatile uint8_t *)ptr;
    while (len--) {
        *p++ = 0;
    }
}
```

The `volatile` qualifier tells the compiler "assume this memory might be read by something outside your analysis — do not optimize away writes to it." This forces the compiler to actually emit the zeroing instructions. For a beginner implementation, `memset` is acceptable. For production code, use `explicit_bzero()` (available on modern Linux/macOS/BSD) or `SecureZeroMemory()` (Windows), or your own `secure_memzero` as above. This is the entry point to constant-time programming: the discipline of writing cryptographic code that provides no useful information through timing, branch patterns, cache access patterns, or memory residue. Your SHA-256 implementation is already constant-time in the cryptographic sense (no data-dependent branches in the hash computation), and the `memset` closes the simplest information leakage channel: memory residue.

Three-Level View: The Complete SHA-256 Implementation

Level 1 — API (What callers see) Four functions: `sha256_init()`, `sha256_update()`, `sha256_finalize()`, and the convenience wrapper `sha256()`. Feed data in any chunk size; always get the same 64-character hex digest. Reuse the context struct by calling `sha256_init()` again. Level 2 — Engine (What the streaming API does) Three phases per computation: initialization loads FIPS constants into the context; update accumulates data by filling a 64-byte buffer and compressing complete blocks as they fill; finalize pads the remaining buffer with the 0x80 byte, zero-fill, and 64-bit message length, then compresses the final one or two padding blocks, then encodes H[0]..H[7] as big-endian bytes. The context struct holds all mutable state — hash state, partial block buffer, buffer size, total message length — enabling true streaming with no size limits. Level 3 — Hardware (What the CPU sees) Each `sha256_compress()` call executes approximately 2,000 instructions: 64 schedule words × ~20 instructions + 64 compression rounds × ~35 instructions. On a 3 GHz CPU doing 3 × 10^9 instructions per second, a single block compresses in roughly 650–700 ns. At 64 bytes per block, this gives throughput around 90–100 MB/s for pure software SHA-256 on a single core. With Intel SHA-NI extensions, the same computation executes in ~160 instructions per block (the

dedicated SHA256RNDS2 instruction processes 2 rounds in a single cycle), reaching 600–700 MB/s. For reference, OpenSSL SHA-256 with SHA-NI on a modern Xeon achieves approximately 10 GB/s using AVX-512 SIMD to process multiple independent hash streams in parallel.

Knowledge Cascade: What This Milestone Unlocks

1. HMAC: Building a MAC from Your Hash

Now that you have a working SHA-256 streaming implementation, you can build HMAC-SHA-256 (Hash-based Message Authentication Code) in about 30 lines of code. HMAC addresses the length extension vulnerability inherent in Merkle-Damgård hashes by using a specific double-hash construction:

```
HMAC(key, message) = SHA-256( (key XOR opad) || SHA-256( (key XOR ipad) || message ) )
```

Where `opad = 0x5C5C5C...` and `ipad = 0x3636...` are fixed padding constants. The inner hash processes the key-padded-with-ipad followed by the message. The outer hash processes the key-padded-with-opad followed by the inner hash result. Because the outer hash includes a key-derived prefix, an attacker who knows HMAC(key, message) cannot compute HMAC(key, message || extension) without knowing the key — the length extension attack is completely neutralized. Every TLS handshake, every HTTPS request, every JWT token uses HMAC as its integrity mechanism. The streaming API you built is exactly the API HMAC uses internally: two `sha256_init` / `sha256_update` / `sha256_finalize` sequences, one for the inner hash and one for the outer.

2. Password Hashing: Why SHA-256 Is Wrong for Passwords

Your SHA-256 implementation is deliberately fast — it's designed to process data at tens of megabytes per second. For most cryptographic uses (data integrity, message authentication, digital signatures), this speed is a virtue. For password hashing, it's a critical flaw. An attacker with a GPU cluster can compute billions of SHA-256 hashes per second. A database of leaked password hashes protected only by SHA-256 can be cracked in hours using a dictionary of common passwords. This is why storing passwords as SHA-256(password) — or even SHA-256(salt || password) — is never acceptable in production. Password hashing algorithms like **bcrypt**, **scrypt**, and **Argon2** are deliberately slow and memory-intensive, designed so that even specialized hardware can only attempt thousands (not billions) of passwords per second. They achieve this by iterating a hash function thousands to millions of times (bcrypt), or by requiring large amounts of RAM that cannot be parallelized (scrypt, Argon2). The underlying primitive is often SHA-256 or another cryptographic hash — but wrapped in an iteration and memory-hardening layer that changes the cost profile entirely. Understanding why your fast SHA-256 is wrong for passwords — and what properties a password hash needs instead — is one of the most practically important cryptographic concepts for any developer building authentication systems.

3. Digital Signatures: SHA-256 as the Hash in RSA-SHA256 and ECDSA

RSA and ECDSA (Elliptic Curve Digital Signature Algorithm) both require hashing the message before signing, because the mathematical operations (modular exponentiation for RSA, scalar multiplication on an elliptic curve for ECDSA) can only operate on fixed-size inputs. SHA-256 is the hash function in RSA-SHA256 (the signature scheme used in TLS certificates, code signing, and PGP) and ECDSA-SHA256 (used in TLS 1.3, Bitcoin transaction signing, and SSH). When you call `openssl verify`, `git verify-commit`, or when your browser validates a TLS certificate chain, it is computing SHA-256 of the signed data using exactly the algorithm you implemented — the streaming API, the same compression function, the same final hash. The signature verification then checks a mathematical relationship between the hash output and the signature bytes. Your SHA-256 is literally the first step in the cryptographic chain of trust that secures HTTPS.

4. Test Vector Methodology: The Skill That Transfers Everywhere

The KAT validation you performed in this milestone is not specific to SHA-256. It is the standard practice for validating *any* cryptographic implementation. When you implement AES, you validate against FIPS 197 Appendix B. When you implement ECDSA, you validate against FIPS 186-5 test vectors. When you implement X25519 key exchange, you validate against RFC 7748 Section 6. The discipline you've practiced here — write the implementation, run the known-answer tests, treat any failure as a bug — transfers with zero modification to every future cryptographic implementation. NIST's Cryptographic Algorithm Validation Program (CAVP) tests every FIPS-compliant library against thousands of generated test vectors. Libraries that pass CAVP testing are trusted for use in government and financial systems. The methodology is exactly what you did in this milestone, just at industrial scale.

> 🔭 **Deep Dive**: For a rigorous treatment of how cryptographic implementations are validated, tested, and certified, see NIST SP 800-140 (the FIPS 140-3 implementation guidance) and the CAVP test vector files at csrc.nist.gov/projects/cryptographic-algorithm-validation-program. The SHA-256 test vectors used in this milestone come from the SHA-256 portion of the CAVP test suite.

Summary: What Your Complete SHA-256 Implementation Guarantees

After completing this milestone, your full SHA-256 implementation enforces these invariants:

1. **H[0]..H[7] are initialized from the FIPS 180-4 initial values** (0x6A09E667, ..., 0x5BE0CD19) by every `sha256_init()` call — not left over from any previous computation.

2. `sha256_update()` **maintains a partial block buffer** of 0–63 bytes; complete blocks are compressed immediately; partial blocks accumulate until a future call fills them.
3. **Total message length is tracked in bits across all** `update()` **calls** in `ctx->msg_len_bits` and written to the final padding block.
4. `sha256_finalize()` **pads correctly** for both cases: `buf_len ≤ 55` bytes remaining (one padding block) and `buf_len ≥ 56` bytes remaining (two padding blocks).
5. **Final digest is H[0]..H[7] encoded as big-endian bytes**, producing a 32-byte binary digest or a 64-character lowercase hex string.
6. **SHA-256("") =** `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855` ✓
7. **SHA-256("abc") =** `ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad` ✓
8. **SHA-256("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq") =** `248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1` ✓
9. **Multiple independent invocations produce identical results** for identical input — `sha256_init()` fully resets all state.
10. **Streaming results match single-call results** for all three test vectors, regardless of chunk size.
11. `sha256_finalize()` **zeros the internal buffer** after producing the digest, preventing sensitive data from lingering in memory. Your SHA-256 implementation is complete. You've translated the FIPS 180-4 specification into a working, verified, streaming cryptographic hash function — from the padding boundary cases through the compression function to the output encoding. Every test vector passes. The same algorithm protects HTTPS connections, signs Git commits, secures Bitcoin transactions, and validates software downloads worldwide. You've built it from scratch.

TDD

A bite-sized but specification-exact implementation of FIPS 180-4 SHA-256, decomposed into four modules that map 1:1 to milestones. Every data structure, bit-level operation, and state transition is visualized. The design enforces: (1) byte-level correctness validated against NIST KATs at each layer boundary, (2) constant-time arithmetic throughout, (3) streaming API with safe state management. Each module builds on the previous, and correctness is checkpointed before advancing.

MODULE SPECIFICATION: Message Preprocessing and Padding

Module ID: `hash-impl-m1`
Domain: Security & Cryptography
Difficulty: Intermediate
Primary Language: C

---

1. Module Charter

This module is responsible for the initial stage of the SHA-256 pipeline: transforming raw, arbitrary-length input bytes into a sequence of perfectly aligned 512-bit (64-byte) blocks. It implements the padding scheme defined in **FIPS 180-4 Section 5.1.1**.

The module does **not** maintain internal hash state, generate message schedules, or perform compression; its scope is strictly limited to data formatting and length encoding. Its primary output is a heap-allocated array of blocks that satisfy the Merkle-Damgård requirement: every block must be exactly 512 bits.

Key Invariants:

- The resulting total bit length is always a multiple of 512.
- The first bit after the original message is always `1` (byte `0x80`).
- The last 64 bits (8 bytes) of the final block always contain the original message length in bits, encoded as a big-endian integer.
- Padding is unambiguous; even an empty input produces exactly one 512-bit block.
- Endianness-aware: The length field is always big-endian, regardless of the host CPU architecture.

---

2. File Structure

The implementation follows a clean separation of data structures and logic. The files should be created in the following order:

1. `sha256_types.h`: Definition of the `SHA256_Block` structure and fixed-width types.
2. `sha256_padding.h`: Public interface for the padding function.
3. `sha256_padding.c`: Logic for `sha256_pad` and internal endianness helpers.

---

3. Complete Data Model

3.1 SHA256_Block Structure

Since SHA-256 operates on 512-bit blocks, we define a specialized struct to represent this unit of work. This ensures type safety and alignment when passing data to the message schedule.

Field	Type	Offset	Description
`bytes`	`uint8_t[64]`	`0x00`	Raw byte array representing 512 bits.

```c
#include <stdint.h>

/**
 * @struct SHA256_Block
 * @brief Represents a single 512-bit (64-byte) block.
 *
 * FIPS 180-4 defines the message blocks as M(1), M(2), ..., M(N).
 * Each block M(i) is exactly 512 bits.
 */
typedef struct {
    uint8_t bytes[64];
} SHA256_Block;
```

3.2 Internal Bit-Level Layout (Final Block)

The final block in the sequence has a mandatory structure. If the message is `L` bytes long, and the block index is `N` (the last block):

Byte Range	Content	Constraint
`0` to `buf_len-1`	Message Remainder	Bytes from original input
`buf_len`	`0x80`	The '1' bit and seven '0' bits
`buf_len+1` to `55`	`0x00`	Zero-padding (variable length)
`56` to `63`	`uint64_t`	Message length in **bits**, Big-Endian

{{DIAGRAM:tdd-diag-1|Padding Layout|Visualizing the 64-byte boundary and the position of 0x80 vs Length}}

4. Interface Contracts

4.1 write_u64_be (Internal Helper)

Signature: `static void write_u64_be(uint8_t *dst, uint64_t value);`

- **Description**: Writes a 64-bit integer into a destination buffer in Big-Endian order.
- **Pre-conditions**: `dst` must point to at least 8 bytes of writable memory.
- **Post-conditions**: `dst[0]` contains the MSB, `dst[7]` contains the LSB.

4.2 sha256_pad (Public API)

Signature: `SHA256_Block* sha256_pad(const uint8_t *input, size_t input_len, size_t *num_blocks);`

- **Parameters**:
  - `input` : Pointer to the raw message bytes. Can be `NULL` if `input_len` is 0.
  - `input_len` : Size of the message in bytes.
  - `num_blocks` : [Output] Pointer to a `size_t` that will receive the count of blocks generated.

- **Return**: Pointer to the first element of a heap-allocated `SHA256_Block` array. Returns `NULL` on allocation failure or if `num_blocks` is `NULL`.
- **Memory Contract**: The caller is responsible for calling `free()` on the returned pointer.
- **Edge Cases**:
  - `input_len = 0`: Returns 1 block containing `0x80` and 63 trailing zero/length bytes.
  - `input_len = 56`: Returns 2 blocks (insufficient space for length in block 1).
  - `input_len = 120`: Returns 2 blocks (message fills block 1 and part of block 2).

---

5. Algorithm Specification: `sha256_pad`

The algorithm must determine exactly how many blocks are required and perform a single allocation to minimize fragmentation.

Step 1: Calculate Required Blocks

1. Let `L` be `input_len`.
2. Total bytes needed: `L + 1 (for 0x80) + 8 (for length)`.
3. Number of zero-padding bytes `k`: `k = (64 - ((L + 9) % 64)) % 64`
4. Total Padded Bytes `P = L + 1 + k + 8`.
5. `num_blocks = P / 64`.

Step 2: Allocation

1. Allocate `P` bytes using `calloc(num_blocks, sizeof(SHA256_Block))`.
2. Using `calloc` is mandatory to ensure all padding bytes are initialized to `0x00` without explicit loops.
3. If allocation fails, set `*num_blocks = 0` and return `NULL`.

Step 3: Fill Message and Separator

1. `memcpy` the first `L` bytes from `input` into the start of the buffer.
2. Set `buffer[L] = 0x80`.

Step 4: Encode Length

1. Calculate `bit_length = (uint64_t)L * 8`.
   - **Warning**: Cast `L` to `uint64_t` before the multiplication to prevent 32-bit wrap-around on 32-bit systems.
2. Locate the final 8 bytes of the buffer: `target = buffer + (num_blocks * 64) - 8`.
3. Call `write_u64_be(target, bit_length)`.

Step 5: Return

1. Return the pointer cast to `(SHA256_Block *)`.

{{DIAGRAM:tdd-diag-2|Padding Sequence Flow|Tracing the movement of bytes for a 56-byte message}}

---

6. Adversary Soul: Threat Model & Pitfalls

6.1 Padding Ambiguity

If the `0x80` separator (the '1' bit) were omitted, a message ending in zeros could be indistinguishable from a shorter message with more padding.

- **Example**: `SHA("abc")` vs `SHA("abc\0")`.
- **Defense**: The mandatory `0x80` byte ensures that even if a message ends with `0x00`, the first byte of padding is `0x80`, acting as a strict domain separator.

6.2 Side-Channel: Memory Residue

Padding functions often handle sensitive data (passwords, keys).

- **Pitfall**: If `sha256_pad` uses `malloc` and doesn't clear the k-padding area, it might leak "garbage" data from previous heap allocations into the final block, making the hash non-deterministic or leaking secrets.
- **Defense**: Use `calloc` or `memset` the entire block array to zero before processing.

6.3 32-bit Integer Overflow

On 32-bit systems, `size_t` is 32 bits.

- **Attack/Bug**: If a message is 600MB, `input_len` is ~600M. `input_len * 8` is ~4.8 billion. This overflows a 32-bit `size_t` (max ~4.2B).
- **Defense**: Explicitly cast `input_len` to `uint64_t` before multiplying by 8.

---

7. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Allocation Failure	`calloc` returns `NULL`	Return `NULL` to caller, set `*num_blocks = 0`.	Yes (via NULL return)
Null Output Param	Check `if (!num_blocks)`	Return `NULL` immediately.	No (Developer Error)
Bit Length Overflow	Manual check (optional)	SHA-256 supports up to 2^64-1 bits. If `input_len` is `size_t`, it's impossible to exceed this on current systems.	N/A
Integer Wrap	Static Analysis / Peer Review	Use `uint64_t` for bit-length calculations.	No

---

8. Implementation Sequence with Checkpoints

Phase 1: Foundation (0.5 Hours)

1. Define `SHA256_Block` in `sha256_types.h`.
2. Implement `static void write_u64_be` in `sha256_padding.c`.
3. **Checkpoint**: Create a small test to verify `write_u64_be` writes `0x0123456789ABCDEF` correctly on an x86 (little-endian) machine.

Phase 2: Padding Logic (1 Hour)

1. Implement the block count calculation logic.
2. Implement the `calloc` and message copy.
3. Implement the separator and length encoding.
4. **Checkpoint**: Call `sha256_pad("abc", 3, &n)`. Verify `n == 1`. Verify `buffer[63] == 0x18` (24 bits).

Phase 3: Boundary Validation (1 Hour)

1. Implement tests for the 55-byte and 56-byte boundaries.
2. Verify memory cleanup (run `valgrind`).
3. **Checkpoint**: `sha256_pad` with 56 bytes returns 2 blocks. `valgrind --leak-check=full` reports 0 leaks.

---

9. Test Specification

9.1 Functional Tests (Happy Path)

- **Test Empty**: `input_len = 0`.
  - *Expect*: `num_blocks = 1`, `bytes[0] = 0x80`, `bytes[63] = 0x00`.
- **Test Standard**: `input = "abc"`, `input_len = 3`.
  - *Expect*: `num_blocks = 1`, `bytes[0..2] = "abc"`, `bytes[3] = 0x80`, `bytes[63] = 0x18`.

9.2 Boundary Tests

- **Test 55-Byte**: `input_len = 55`.
  - *Expect*: `num_blocks = 1`. The `0x80` is at `bytes[55]`. Length fits in the last 8 bytes.
- **Test 56-Byte**: `input_len = 56`.
  - *Expect*: `num_blocks = 2`. The `0x80` is at `bytes[56]` (start of Block 2). Length is in Block 2 `bytes[120..127]`.

9.3 Security Tests

- **Stress Allocation**: Call with `input_len = (size_t)-1` (or a very large value).
  - *Expect*: Graceful `NULL` return, not a crash.

---

10. Performance Targets

Operation	Target	How to Measure
Padding Speed	> 500 MB/s	`gettimeofday` before/after padding a 100MB buffer.
Memory Overhead	`input_len + 128` bytes max	Monitor `heap_usage` via `valgrind`.
Initialization	Constant Time O(1)	Ensure the `zero_pad` calculation does not use a linear search loop.

---

11. Wire Format: SHA-256 Padded Block

For an input of 3 bytes `"abc"`:

```
Offset: 00 01 02 03 04 05 ... 37 38 39 3A 3B 3C 3D 3E 3F
Data:   61 62 63 80 00 00 ... 00 00 00 00 00 00 00 00 18
        |--Msg--|  |--Padding--|              |--Length--|
```

`18` hex = 24 decimal. (3 bytes * 8 bits/byte = 24 bits).

MODULE SPECIFICATION: Message Schedule Generation

Module ID: `hash-impl-m2`
Domain: Security & Cryptography
Difficulty: Intermediate
Primary Language: C

---

1. Module Charter

This module implements the **Message Schedule Expansion** for SHA-256 as defined in **FIPS 180-4 Section 6.2**. Its purpose is to transform a single 512-bit (64-byte) message block, produced by the padding module (M1), into a 64-word sequence of 32-bit integers (W_0 through W_{63}).

This process is the primary engine of **diffusion** in SHA-256. While the first 16 words are simply a re-interpretation of the input block, the subsequent 48 words are generated through a recursive bit-mixing process. This ensures that a single-bit change in the input block propagates and scrambles across all 64 rounds of the compression function.

Key Responsibilities:

- Big-endian parsing of raw bytes into 32-bit words.
- Implementation of the bitwise "mixing" primitives σ_0 and σ_1.
- Execution of the message schedule recurrence relation.
- Execution in constant-time (no data-dependent branching).

What this module does NOT do:

- It does not modify the global hash state ($H_0 \dots H_7$).
- It does not implement the compression rounds (Milestone 3).
- It does not handle message padding (Milestone 1).

---

2. File Structure

The implementation follows a strict creation order to ensure primitives are tested before the integration logic is written.

1. `sha256_schedule.h`: Public interface and function prototypes.

2. `sha256_schedule.c` : Implementation of bitwise primitives, parsing, and expansion.
3. `test_schedule.c` : Unit tests using NIST Appendix B.1 intermediate values.

---

3. Complete Data Model

3.1 Input: SHA256_Block (From M1)

The module accepts a pointer to a `SHA256_Block` . This is a 64-byte aligned structure.

Offset	Content	Interpretation
`0x00` - `0x03`	4 bytes	Becomes W_0 (Big-Endian)
`0x04 - 0x07`	4 bytes	Becomes W_1 (Big-Endian)
...	...	...
`0x3C - 0x3F`	4 bytes	Becomes W_{15} (Big-Endian)

3.2 Output: The W Array

The output is a caller-allocated array of 64 unsigned 32-bit integers.

Word Index	Origin	Constraint
`W[0..15]`	Direct Parse	Must be big-endian converted.
`W[16..63]`	Expansion	Generated via σ functions and addition mod 2^{32}.

```c
#include <stdint.h>
#include "sha256_types.h" // Defines SHA256_Block

/**
 * @brief The message schedule workspace.
 * Size: 64 * 4 bytes = 256 bytes.
 * Usually allocated on the stack within the compression function.
 */
typedef uint32_t SHA256_Schedule[64];
```

{{DIAGRAM:tdd-diag-7|Word Parsing Logic|Visualizing the extraction of four 8-bit bytes into one 32-bit word using Big-Endian order}}

---

4. Interface Contracts

4.1 rotr32 (Inline Primitive)

Signature: `static inline uint32_t rotr32(uint32_t x, uint32_t n);`

- **Description**: Performs a bitwise right-rotation. Bits shifted off the right wrap to the left.
- **Invariant**: No information is lost (unlike `SHR`).
- **Constraint**: `n` must be in the range `[1, 31]` . If `n=0` or `n=32` , behavior is undefined in C.

4.2 load_u32_be (Inline Primitive)

Signature: `static inline uint32_t load_u32_be(const uint8_t *p);`

- **Description**: Reconstructs a 32-bit word from 4 bytes in Big-Endian order.
- **Implementation Path**: Must use bitwise OR and shifts. `(p[0] << 24) | (p[1] << 16) | (p[2] << 8) | p[3]` .
- **Rationale**: Direct pointer casting `*(uint32_t*)p` is an **Endianness Bug** on x86/ARM systems.

4.3 sigma0 & sigma1 (Internal Helpers)

Signatures:

- `static uint32_t sigma0(uint32_t x);`
- `static uint32_t sigma1(uint32_t x);`
- **σ_0 Constants**: `ROTR(x, 7) ^ ROTR(x, 18) ^ SHR(x, 3)`
- **σ_1 Constants**: `ROTR(x, 17) ^ ROTR(x, 19) ^ SHR(x, 10)`

4.4 sha256_message_schedule (Public API)

Signature: `void sha256_message_schedule(const SHA256_Block *block, uint32_t W[64]);`

- **Input**: A single 64-byte block.
- **Output**: 64 words of expanded schedule.
- **Side Effects**: None. This is a pure transformation function.

---

5. Algorithm Specification

5.1 Stage 1: Initial Parsing (t = 0 to 15)

1. Iterate `t` from 0 to 15.
2. Calculate byte offset: `offset = t * 4`.
3. Load word: `W[t] = load_u32_be(&block->bytes[offset])`.

5.2 Stage 2: Schedule Expansion (t = 16 to 63)

For each iteration `t` from 16 to 63, compute the word using the following recurrence: $$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$$

1. **Mixing**: Apply `sigma1` to the word two steps back.
2. **Mixing**: Apply `sigma0` to the word fifteen steps back.
3. **Summation**: Add results to W_{t-7} and W_{t-16}.
4. **Modular Arithmetic**: The addition is performed modulo 2^{32}. In C, using `uint32_t` ensures this behavior automatically upon overflow. No masking is required.

{{DIAGRAM:tdd-diag-8|Recurrence Window|Visualizing the 'sliding window' of the 4 words used to generate the current W[t]}}

---

6. Adversary Soul: Threat Model & Pitfalls

6.1 The "Silent Failure" of SHR vs ROTR

The most common implementation error is using the right-shift operator `>>` (SHR) where the specification mandates `ROTR`.

- **Attack/Bug**: If `sigma0` uses `(x >> 7) ^ (x >> 18) ^ (x >> 3)`, it permanently destroys bit information that should have wrapped around.
- **Impact**: The hash function will produce 100% incorrect results that look valid but fail all security properties.
- **Defense**: Implement `rotr32` as a dedicated, tested macro or inline function: `(x >> n) | (x << (32 - n))`.

6.2 Endianness Vulnerability

SHA-256 is defined on Big-Endian words. Most modern development occurs on Little-Endian (LE) CPUs.

- **Pitfall**: A developer uses `memcpy(W, block->bytes, 64)`. On x86, W_0 will have its bytes swapped, resulting in a completely different schedule.
- **Defense**: Mandatory use of the `load_u32_be` helper which explicitly assigns byte positions.

6.3 σ vs Σ Confusion

FIPS 180-4 uses lowercase σ for the schedule and uppercase Σ for the compression rounds. They use different rotation constants.

- **Pitfall**: Swapping constants between these two groups.
- **Defense**: Use strict naming conventions: `sigma0/sigma1` for M2, and `SIGMA0/SIGMA1` for M3.

---

7. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Buffer Overflow	Static analysis / Boundary checks	The loop is fixed at `t < 64`. Ensure `W` is exactly `uint32_t[64]`.	No
Null Pointer	Runtime check: `if (!block		!W)`
Undefined Shift	Code review	Ensure `rotr32` is only called with NIST constants (7, 18, 17, 19).	No
Integer Overflow	Native hardware behavior	Addition mod 2^{32} is the desired behavior; no recovery needed.	No

8. Implementation Sequence with Checkpoints

Phase 1: Primitives (0.5 Hours)

1. Implement `rotr32` in `sha256_schedule.c`.
2. Implement `load_u32_be`.
3. **Checkpoint**: Write a test that takes `0x01020304` (bytes) and ensures `load_u32_be` returns `0x01020304` as a `uint32_t`, regardless of CPU endianness.

Phase 2: Mixing Functions (1 Hour)

1. Implement `sigma0` and `sigma1` using `rotr32` and `>>`.
2. **Checkpoint**: Use NIST Appendix B.1 values. For input `W[1] = 0x00000000`, `sigma0(0)` must be `0`. For `W[15] = 0x00000018`, compute `sigma1(0x18)` and verify it matches `0x000F0000`.

Phase 3: Expansion Logic (1 Hour)

1. Implement `sha256_message_schedule` logic (init loop + expand loop).
2. **Checkpoint**: Feed the "abc" padded block from M1. Verify:
   - `W[0] = 0x61626380`
   - `W[15] = 0x00000018`
   - `W[16] = 0x61626380`
   - `W[17] = 0x000F0000`

9. Test Specification

9.1 Primitive Validation

- `rotr32(0x80000000, 1)` -> `0x40000000`
- `rotr32(0x00000001, 1)` -> `0x80000000`
- `load_u32_be({0x12, 0x34, 0x56, 0x78})` -> `0x12345678`

9.2 Sigma Function KAT (Known Answer Test)

Input (Hex)	σ_0 Output	σ_1 Output
`00000000`	`00000000`	`00000000`
`00000018`	`00030003` (verify)	`000F0000`
`61626380`	`?` (Check NIST)	`?` (Check NIST)

9.3 Integration Test (The "abc" Block)

Message: `"abc"` Padded Block: `61 62 63 80 00...00 18` **Required Assertions:**

- `W[0] == 0x61626380`
- `W[15] == 0x00000018`
- `W[16] == sigma1(W[14]) + W[9] + sigma0(W[1]) + W[0]`
- Verify against NIST Appendix B.1 for all words up to W_{63}.

10. Performance Targets

Operation	Target	How to Measure
Expansion Speed	< 500 cycles / block	Use `RDTSC` instruction on x86 to measure `sha256_message_schedule`.
Instruction Count	~1200 instructions	Inspect disassembly with `objdump -S`. Look for `ROR` instructions.
Stack Usage	Exactly 256 bytes	Monitor stack pointer or use static analysis tools.

11. Mathematical Intuition: The Diffusion Matrix

The message schedule expansion can be viewed as a linear transformation over the field $GF(2^{32})$. Because the σ functions incorporate rotations, they ensure that every bit of $W_0 \dots W_{15}$ eventually "touches" every bit of the state.

{{DIAGRAM:tdd-diag-9|Diffusion Cascade|Showing how a single bit flip in W[0] affects multiple bits in W[16], W[17], and eventually saturates the entire schedule by word 48}}

By the time the compression function reaches round 48, the avalanche effect is nearly complete. This is why SHA-256 uses 64 rounds—it provides a significant safety margin beyond the point of full bit saturation.

12. Constant-Time Analysis

In the context of side-channel attacks, the message schedule is inherently safe because:

1. It contains no `if/else` branches that depend on the value of W_t.
2. It contains no array indexing that depends on the value of W_t (preventing cache-timing attacks).
3. The bitwise operations (`XOR`, `SHR`, `ROTR`) and `ADD` take the same number of cycles on modern CPUs regardless of the operands.

MANDATORY: Ensure no compiler optimizations (like "Short Circuiting" a sequence of additions) introduce branching. (Highly unlikely with `uint32_t` but worth noting for security audits).

MODULE SPECIFICATION: Compression Function

Module ID: `hash-impl-m3`
Domain: Security & Cryptography
Difficulty: Intermediate
Primary Language: C

1. Module Charter

This module implements the core cryptographic engine of SHA-256: the **Compression Function**. Following the **Merkle-Damgård construction**, this module consumes exactly one 512-bit message block and updates the internal 256-bit hash state. It is the "churn" of the algorithm, responsible for mapping two inputs (the current hash state and the message block) into a single output (the updated hash state) such that the mapping is computationally one-way and collision-resistant.

The compression function achieves its security properties through 64 rounds of transformation using bitwise logical functions (**Ch**, **Maj**, **Σ0**, **Σ1**), modular addition, and a set of 64 round constants (**K**).

Key Responsibilities:

- Initialization of eight working variables (`a` through `h`) from the current hash state.
- Execution of 64 iterative rounds of bit-scrambling and injection.
- Implementation of the non-linear "Choice" (Ch) and "Majority" (Maj) functions.
- Implementation of the high-diffusion "Uppercase Sigma" (Σ) functions.
- Final integration of the working variables back into the hash state using modular addition (the Davies-Meyer feed-forward).

What this module does NOT do:

- It does not handle padding or bit-length tracking (handled in M1/M4).

- It does not manage the streaming of multiple blocks (handled in M4).
- It does not perform hexadecimal formatting for output (handled in M4).

---

2. File Structure

The compression function is the bridge between the message schedule and the final output. The files must be implemented and tested in this order:

1. `sha256_op_primitives.h`: Inline implementations of `ch`, `maj`, `Sigma0`, and `Sigma1`.
2. `sha256_constants.c`: The 64-word `K` constant array and the 8-word `H_INIT` array.
3. `sha256_compress.c`: The primary `sha256_compress` function logic.
4. `test_compress_kat.c`: Known Answer Tests (KAT) for single-block compression.

---

3. Complete Data Model

3.1 Hash State (H)

The hash state consists of eight 32-bit words, totaling 256 bits. This state is updated in-place by the compression function.

State Index	FIPS Name	Initial Value (Fractional Part of sqrt(prime))
`H[0]`	H_0	`0x6A09E667` (sqrt(2))
`H[1]`	H_1	`0xBB67AE85` (sqrt(3))
`H[2]`	H_2	`0x3C6EF372` (sqrt(5))
`H[3]`	H_3	`0xA54FF53A` (sqrt(7))
`H[4]`	H_4	`0x510E527F` (sqrt(11))
`H[5]`	H_5	`0x9B05688C` (sqrt(13))
`H[6]`	H_6	`0x1F83D9AB` (sqrt(17))
`H[7]`	H_7	`0x5BE0CD19` (sqrt(19))

3.2 Working Variables (a-h)

During the 64 rounds, the state is held in eight local `uint32_t` variables. These variables are mutated in every round.

{{DIAGRAM:tdd-diag-12|Working Variable Stack|Visualizing a..h as a 256-bit vertical register stack that "shifts" and "injects" data}}

3.3 Round Constants (K)

The 64 constants K_t are used to break symmetry between rounds and ensure that different rounds contribute uniquely to the final diffusion.

```c
static const uint32_t K[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
};
```

---

4. Interface Contracts

4.1 Bitwise Transformation Primitives (Internal)

All functions below must be implemented as `static inline` for maximum performance and to avoid the overhead of function calls inside the 64-round loop.

Function	Logic	Purpose
`ch(x, y, z)`	`(x & y) ^ (~x & z)`	Choice: bits from y if x is 1, else bits from z.
`maj(x, y, z)`	`(x & y) ^ (x & z) ^ (y & z)`	Majority: bit is 1 if 2 or more inputs are 1.
`Sigma0(x)`	`ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22)`	Diffusion mixing for the "a-group" variables.
`Sigma1(x)`	`ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25)`	Diffusion mixing for the "e-group" variables.

4.2 sha256_compress (Public API)

Signature: `void sha256_compress(uint32_t H[8], const SHA256_Block *block, uint32_t W[64]);`

- **Parameters**:
  - `H`: Array of 8 hash words. Updated in-place (Davies-Meyer addition).
  - `block`: The 64-byte padded message block to process.
  - `W`: Caller-provided workspace for the message schedule (prevents stack/heap thrashing).
- **Pre-conditions**:
  - `H` must be initialized (either to `H_INIT` or previous block output).
  - `block` must be non-NULL and 512-bit aligned.
- **Post-conditions**:
  - `H` contains the updated state ($H_i = H_i + \text{working_var}_i \pmod{2^{32}}$).
  - `W` will contain the message schedule for the processed block.

---

5. Algorithm Specification: `sha256_compress`

The algorithm consists of four distinct phases: Schedule Prep, Initialization, Round Execution, and State Update.

Step 1: Message Schedule Preparation

1. Call `sha256_message_schedule(block, W)` (from M2).
2. This populates $W_{0 \dots 63}$ for use in the rounds.

Step 2: Working Variable Initialization

Load the current hash state into working variables a through h:

```c
uint32_t a = H[0], b = H[1], c = H[2], d = H[3],
         e = H[4], f = H[5], g = H[6], h = H[7];
```

Step 3: The 64 Rounds

Iterate t from 0 to 63. In each round:

1. **Calculate T_1 (Temp 1)**: $T_1 = h + \Sigma_1(e) + \text{ch}(e, f, g) + K_t + W_t$
2. **Calculate T_2 (Temp 2)**: $T_2 = \Sigma_0(a) + \text{maj}(a, b, c)$
3. **Rotate Variables**:
   - $h = g$
   - $g = f$
   - $f = e$
   - $e = d + T_1$ (**Crucial: e gets the message injection**)
   - $d = c$
   - $c = b$
   - $b = a$

- $a = T_1 + T_2$ (**New a is the sum of both temp variables**)

{{DIAGRAM:tdd-diag-13|Round Transformation Flow|Arrows showing how d and a are updated while others shift down}}

Step 4: Final State Update (Feed-Forward)

Add the values of the working variables back into the original hash state:

```c
H[0] += a; H[1] += b; H[2] += c; H[3] += d;

H[4] += e; H[5] += f; H[6] += g; H[7] += h;
```

Constraint: All additions are implicitly modulo 2^{32}.

---

6. Adversary Soul: Implementation Pitfalls

6.1 The "Update Order" Bug

A common beginner error is to update $a, b, c \dots$ before calculating T_1 and T_2.

- **Risk**: If you set `h = g` before using `h` in the T_1 calculation, the logic breaks.
- **Defense**: Calculate T_1 and T_2 into local variables **first**, then perform the shifting of $a \dots h$.

6.2 The "e-Update" Bug

The FIPS spec defines $e = d + T_1$ and $a = T_1 + T_2$.

- **Risk**: Some implementations accidentally write $e = d + T_1 + T_2$ or $a = T_1$.
- **Impact**: Subtle failure that only manifests as an incorrect final hash.
- **Defense**: Meticulous verification of the round logic against the NIST pseudocode.

6.3 Lowercase vs. Uppercase Sigma Constants

The schedule (M2) uses lowercase σ with constants (7, 18, 3) and (17, 19, 10). The rounds (M3) use uppercase Σ with constants (2, 13, 22) and (6, 11, 25).

- **Risk**: Mixing these up.
- **Defense**: Use distinctive names in code (`SIGMA_UPPER_0` vs `sigma_lower_0`).

6.4 The "OR instead of XOR" in Ch Function

The Choice function is defined as `(x & y) ^ (~x & z)`.

- **Risk**: Using `(x & y) | (~x & z)`. While mathematically equivalent for non-overlapping bits, bitwise XOR is standard practice in cryptographic implementations to prevent potential carry-related side channels or logic errors in non-standard hardware.
- **Defense**: Use the XOR (`^`) operator as specified.

---

7. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Null Workspace	`if (!W)` check	Return or assert; the function requires workspace to operate safely.	No
K Constant Corrupt	Unit Test (Phase 3)	Re-verify constants against FIPS 180-4 Table 1.	No
State Drift	KAT Failure (Phase 5)	Trace round-by-round intermediate values against NIST Appendix B.	No

---

8. Implementation Sequence with Checkpoints

Phase 1: Logical Primitives (0.5 Hours)

1. Implement `ch`, `maj`, `Sigma0`, `Sigma1` as macros or static inline functions.

2. **Checkpoint**: Test `ch(0xFFFFFFFF, 0xAAAAAAAA, 0x55555555)` returns `0xAAAAAAAA`. Test `maj(0xAAAAAAAA, 0x55555555, 0xAAAAAAAA)` returns `0xAAAAAAAA`.

Phase 2: Constant Verification (0.5 Hours)

1. Define the `K` array.
2. **Checkpoint**: Assert `K[0] == 0x428a2f98` and `K[63] == 0xc67178f2`.

Phase 3: The Compression Core (1 Hour)

1. Implement `sha256_compress` round loop.
2. Implement the Davies-Meyer feed-forward (`H += update`).
3. **Checkpoint**: Verify using a "zero-block" (a 64-byte block of all zeros). Compare the state after 1 round to a manual calculation.

Phase 4: Intermediate Value Validation (1 Hour)

1. Instrument the code to print `a..h` values after each round.
2. **Checkpoint**: Process the "abc" block. Compare your Round 0 output to NIST Appendix B:
   - `a = 5d6aebb1`
   - `b = 6a09e667`
   - ...
   - `h = 1f83d9ab`

---

9. Test Specification

9.1 Logic Gates Truth Tables

- Verify `maj` output for all 8 combinations of 1-bit inputs (000, 001, ... 111).
- Verify `ch` output for all 8 combinations.

9.2 Rotation Verification

- `Sigma0(0x00000001)` : Verify bits 2, 13, and 22 are set in the result.
- `Sigma1(0x00000001)` : Verify bits 6, 11, and 25 are set.

9.3 Single-Block KAT ("abc")

- **Input**: `H = H_INIT` , `block = padded "abc"` .
- **Expect**: `H[0] = 0xBA7816BF` , `H[1] = 0x8F01CFEA` , `H[2] = 0x414140DE` , `H[3] = 0x5DAE2223` , `H[4] = 0xB00361A3` , `H[5] = 0x96177A9C` , `H[6] = 0xB410FF61` , `H[7] = 0xF20015AD` .

---

10. Performance Targets

Operation	Target	Measurement
Cycle Count	< 2500 cycles/block	Use `perf` or `valgrind --tool=callgrind` .
Instruction Throughput	> 0.5 IPC	Check via hardware performance counters.
Round Latency	< 10ns per round	Benchmark 1 million iterations of a single round.

---

11. Threat Model: Side-Channel Resistance

11.1 Constant-Time Execution

The compression function must execute in the same number of CPU cycles regardless of the input data.

- **Vulnerability**: If `ch` or `maj` used `if` statements, an attacker could timing-attack the data.
- **Defense**: Every operation in this module (XOR, AND, NOT, ADD, ROTR) is a constant-time instruction on all modern CPUs (x86, ARM). **NEVER** introduce a branch based on `a..h` or `W_t` .

11.2 Register Spilling

The working variables `a..h` are sensitive. If the compiler spills these variables to the stack due to register pressure, they might linger in memory.

- **Defense**: (Advanced) Use `volatile` carefully or rely on `memset` in the finalizer (M4) to clear the stack if the environment is high-security. For this intermediate project, ensure variables are local to the function scope.

---

12. Hardware Acceleration Note (SHA-NI)

On modern Intel (Goldmont+) and ARM (v8 Crypto) CPUs, the rounds can be accelerated using dedicated instructions like `SHA256RNDS2`.

- **Standard Implementation**: 64 round iterations in C.
- **Accelerated Implementation**: Replaces the round loop with 32 calls to `SHA256RNDS2`, processing 2 rounds per instruction. This is out of scope for Milestone 3 but is the "expert" path for performance.

MODULE SPECIFICATION: Final Hash Output and Validation

Module ID: `hash-impl-m4` **Domain**: Security & Cryptography **Difficulty**: Intermediate **Primary Language**: C

---

1. Module Charter

This module serves as the orchestrator for the entire SHA-256 pipeline. It integrates the preprocessing (M1), message schedule (M2), and compression logic (M3) into a cohesive, production-grade streaming API. The module provides a stateful context (`SHA256_CTX`) that allows users to process data of arbitrary size in non-contiguous chunks—essential for hashing large files or network streams.

Key Responsibilities:

- Management of the `SHA256_CTX` lifecycle (init, update, finalize).
- Implementation of a 3-phase buffer management strategy in `sha256_update` to optimize throughput by bypassing the internal buffer for full blocks.
- Execution of the final padding logic in `sha256_finalize`, correctly handling the 1-block vs. 2-block padding boundary.
- Transformation of internal 32-bit state words into a big-endian binary digest and a lowercase hexadecimal string.
- Verification of cryptographic correctness against NIST FIPS 180-4 Known Answer Tests (KATs).

What this module does NOT do:

- It does not implement the bitwise `Sigma` , `Ch` , or `Maj` functions (delegated to M2/M3).
- It does not perform heap allocations; all state management is caller-allocated.

---

2. File Structure

The files should be implemented in the following sequence to allow for incremental integration testing:

1. `sha256.h` : The public API and `SHA256_CTX` definition.
2. `sha256.c` : The core logic for the streaming state machine and convenience wrappers.
3. `sha256_test.c` : The validation suite containing NIST test vectors and streaming chunk-size edge cases.

---

3. Complete Data Model

3.1 SHA256_CTX (Context Object)

The `SHA256_CTX` holds the entire state of an in-progress hash. It must be treated as an opaque object by the user.

Field	Type	Offset	Description
`h`	`uint32_t[8]`	`0x00`	The intermediate 256-bit hash state (H0..H7).
`buf`	`uint8_t[64]`	`0x20`	Internal buffer for data that hasn't reached a 512-bit boundary.
`buf_len`	`uint32_t`	`0x60`	Current number of bytes stored in `buf` (0-63).
`msg_len`	`uint64_t`	`0x68`	Cumulative message length in **bits**. Used for final padding.

Byte-Level Layout (112 Bytes Total):

```
[ H0...H7 (32B) ] [ buf (64B) ] [ buf_len (4B) ] [ PADDING (4B) ] [ msg_len (8B) ]
0x00              0x20          0x60             0x64             0x68
```

Note: The 4-byte padding at 0x64 is added by the compiler on 64-bit systems to align the uint64_t msg_len to an 8-byte boundary.

{{DIAGRAM:tdd-diag-20|Context Memory Layout|Visualizing the state retention across update calls}}

---

4. Interface Contracts

4.1 sha256_init

Signature: `void sha256_init(SHA256_CTX *ctx);`

- **Description**: Resets the context to the starting state defined by FIPS 180-4.
- **Invariants**: `h` is set to the square roots of the first 8 primes. `buf_len` and `msg_len` are zeroed.
- **Security**: Must be called before every new message to prevent state leakage from previous computations.

4.2 sha256_update

Signature: `void sha256_update(SHA256_CTX *ctx, const uint8_t *data, size_t len);`

- **Description**: Absorbs `len` bytes of `data` into the hash state.
- **Complexity**: O(N) where N is message length.
- **Constraints**: Can be called with any `len` (0 to `SIZE_MAX`).

4.3 sha256_finalize

Signature: `void sha256_finalize(SHA256_CTX *ctx, uint8_t digest[32]);`

- **Description**: Performs padding, processes the final blocks, and writes the 32-byte result.
- **Output**: `digest` is written as 32 big-endian bytes (8 words x 4 bytes).
- **Security**: Calls `secure_memzero` on the context buffer and schedule workspace before returning.

4.4 sha256_hex

Signature: `void sha256_hex(const uint8_t digest[32], char out[65]);`

- **Description**: Converts binary digest to lowercase hex.
- **Output**: `out` is null-terminated.

---

5. Algorithm Specification

5.1 The 3-Phase Update Logic (`sha256_update`)

To maximize performance, the update function avoids unnecessary copying by streaming full blocks directly from the source buffer.

Phase 1: Handle Buffered Data If `ctx->buf_len > 0`:

1. Calculate `remaining_space = 64 - ctx->buf_len`.
2. Let `fill_amt = min(len, remaining_space)`.
3. `memcpy` `fill_amt` bytes from `data` to `ctx->buf + ctx->buf_len`.
4. Update `ctx->buf_len`.
5. If `ctx->buf_len == 64`:

- Call `sha256_compress(ctx->h, ctx->buf, workspace)`.
- Reset `ctx->buf_len = 0`.

6. Advance `data` by `fill_amt` and decrement `len` by `fill_amt`.

Phase 2: Bulk Processing (Fast Path) While `len >= 64`:

1. Call `sha256_compress(ctx->h, data, workspace)`.
2. Advance `data` by 64 and decrement `len` by 64. *This bypasses the internal context buffer entirely.*

Phase 3: Buffer Remainder If `len > 0`:

1. `memcpy` remaining `len` bytes from `data` to `ctx->buf`.
2. Set `ctx->buf_len = len`.

{{DIAGRAM:tdd-diag-21|Update Phase Sequence|Flowchart showing the transition between buffering and bulk compression}}

5.2 The Finalization Logic (`sha256_finalize`)

The most sensitive part of the implementation is the transition from raw message to the 64-bit length suffix.

1. **Length Capture**: Store `total_bits = ctx->msg_len` into a local `uint64_t`.
2. **Append '1' bit**: Write `0x80` to `ctx->buf[ctx->buf_len]`. Increment `ctx->buf_len`.
3. **Branch Decision**:
   - **Single-Block Path**: If `ctx->buf_len <= 56`:
     - `memset` `ctx->buf + ctx->buf_len` to `0` up to index 55.
   - **Double-Block Path**: If `ctx->buf_len > 56`:
     - `memset` remainder of `ctx->buf` to `0`.
     - Compress current `ctx->buf`.
     - `memset` entire `ctx->buf` to `0` up to index 55.
4. **Append Length**: Write `total_bits` as 8 Big-Endian bytes at `ctx->buf[56..63]`.
5. **Final Compression**: Call `sha256_compress` on the final block.
6. **Digest Extraction**: Loop 8 times; for each word `ctx->h[i]`, write 4 bytes using: `out[i*4 + 0] = (h[i] >> 24) & 0xFF` ... `out[i*4 + 3] = h[i] & 0xFF`.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
State Contamination	Developer Inspection / KATs	Ensure `sha256_init` zeroes `buf_len` and `msg_len`.	No
Bit-Length Overflow	Bit-count calculation	SHA-256 supports 2^64 bits. Check if `len > (SIZE_MAX / 8)` before multiplication.	No
Buffer Overrun	`update` logic bounds check	Ensure Phase 1 never copies more than `64 - buf_len`.	No
Invalid Digest Pointer	Runtime Check	`if (!digest) return;`	No
Endianness Reversal	KAT Failure	Explicitly use bit-shifts in finalization (avoid `memcpy` for words).	No

---

7. Implementation Sequence with Checkpoints

Phase 1: Context & Init (0.5 Hours)

1. Define `SHA256_CTX` in `sha256.h`.
2. Implement `sha256_init` with the prime-root constants.
3. **Checkpoint**: Create a context and verify that all 8 state words are correctly initialized.

Phase 2: Update & Fast Path (1.5 Hours)

1. Implement the 3-phase `sha256_update`.
2. Integration: Link to M3 `sha256_compress`.
3. **Checkpoint**: Feed 128 bytes (2 blocks). Verify `sha256_compress` is called exactly twice and `buf_len` is 0.

Phase 3: Finalization & Padding (1 Hour)

1. Implement the length-padding logic in `sha256_finalize`.
2. Implement big-endian extraction.
3. **Checkpoint**: Run the "abc" test vector. Output must be `ba7816bf...`.

Phase 4: Formatting & Convenience (0.5 Hours)

1. Implement `sha256_hex` and the `sha256()` one-call function.
2. **Checkpoint**: `sha256_hex` produces the standard 64-char string.

---

8. Test Specification

8.1 NIST KAT (Known Answer Tests)

Validate against FIPS 180-4 Appendix B:

- **Empty**: `""` -> `e3b0c442...b855`
- **Small**: `"abc"` -> `ba7816bf...15ad`
- **Boundary**: 56 characters `"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"` -> `248d6a61...06c1`

8.2 Streaming Integrity Tests

- **Byte-by-Byte**: Feed `"abc"` in three 1-byte calls.
- **Misaligned**: Feed a 128-byte message in 7-byte chunks.
- **All-at-once**: Feed 1MB in one call. *Result must be identical across all three patterns.*

---

9. Performance Targets

Operation	Target	Measurement
Fast Path Throughput	> 100 MB/s	Time `sha256_update` with a 100MB buffer on a single 3GHz core.
Memory footprint	< 128 Bytes	Total size of `SHA256_CTX`.
Finalization Latency	< 2000 cycles	Measured from `finalize` call to binary output.

---

10. Adversary Soul: The Cryptographic "Final" View

10.1 Length Extension Vulnerability

Note that SHA-256 (like all Merkle-Damgård hashes) is vulnerable to length extension. If an attacker knows `SHA256(Secret || Message)`, they can compute `SHA256(Secret || Message || Padding || ExtraMessage)` by initializing their `SHA256_CTX` with your output hash and calling `update` with `ExtraMessage`.

- **Mitigation**: While not solvable within the hash function itself, this is why users should use **HMAC-SHA256** for authentication.

10.2 Constant-Time and Side Channels

The `sha256_update` loop contains branches (`if (len >= 64)`), but these branches depend on the **length** of the message, which is typically public, not the **content**.

- **Requirement**: Ensure the `sha256_compress` function (M3) is used as the only way to process data to maintain its constant-time property regarding data content.

10.3 Secret Erasure

In security contexts, message fragments in the context buffer can be leaked via memory dumps or use-after-free vulnerabilities.

- **Requirement**: `sha256_finalize` MUST call `memset` on the buffer and any stack-allocated workspaces before returning.

```c
// Implementation of secure_memzero to prevent compiler optimization

static void secure_memzero(void *p, size_t len) {

    volatile uint8_t *ptr = (volatile uint8_t *)p;

    while (len--) *ptr++ = 0;

}
```

{{DIAGRAM:tdd-diag-22|The Final Pipeline|A sequence diagram showing data flowing from raw input to the final 64-character hex string}}

Project Structure: SHA-256 Hash Function

Directory Tree

```text
sha256-impl/
├── include/                 # Header files (Public & Internal)
│   ├── sha256_types.h      # Core data structures (M1: Block definitions)
│   ├── sha256_padding.h    # Padding interface (M1: Buffer formatting)
│   ├── sha256_schedule.h   # Schedule expansion interface (M2: Diffusion)
│   ├── sha256_op_primitives.h # Logic gates & Σ functions (M3: Ch/Maj/Sigma)
│   └── sha256.h            # Main streaming API (M4: Context & Orchestration)
├── src/                     # Implementation files
│   ├── sha256_padding.c    # Padding & endianness logic (M1)
│   ├── sha256_schedule.c   # Message expansion & bit-mixing (M2)
│   ├── sha256_constants.c  # K-constants & H_INIT values (M3)
│   ├── sha256_compress.c   # 64-round compression engine (M3)
│   └── sha256.c            # Streaming state machine (M4)
├── tests/                   # Validation suites
│   ├── test_padding.c      # Boundary tests for 55/56 bytes (M1)
│   ├── test_schedule.c     # NIST Appendix B.1 intermediate values (M2)
│   ├── test_compress_kat.c # Single-block known answer tests (M3)
│   └── sha256_test.c       # Full NIST validation & streaming tests (M4)
├── build/                   # Compiled artifacts (Created by Makefile)
├── Makefile                 # Build system (Compiler flags & linking)
├── .gitignore               # Ignores build/ and binary outputs
└── README.md               # Implementation notes & FIPS 180-4 references
```

Creation Order

1. **Foundation & Types** (M1)
   - Create `include/sha256_types.h` to define `SHA256_Block`.
   - Implement `src/sha256_padding.c` with big-endian helpers (`write_u64_be`).
2. **Message Expansion** (M2)
   - Implement `rotr32` and `load_u32_be` in `src/sha256_schedule.c`.
   - Define lowercase σ_0 and σ_1 mixing functions.
   - Implement the 16-to-64 word expansion recurrence.
3. **The Cryptographic Core** (M3)
   - Define `include/sha256_op_primitives.h` (Ch, Maj, Σ_0, Σ_1).
   - Populate `src/sha256_constants.c` with prime-derived K and H_{INIT} values.
   - Implement the 64-round loop in `src/sha256_compress.c`.
4. **Streaming API & Orchestration** (M4)
   - Define `SHA256_CTX` in `include/sha256.h`.
   - Implement the 3-phase `update` logic (buffering vs. bulk) in `src/sha256.c`.
   - Implement `finalize` with the length-padding logic.
5. **Validation** (Integration)
   - Assemble `tests/sha256_test.c`.
   - Verify against the "abc" and 1,000,000 'a' NIST test vectors.

File Count Summary

- **Total files**: 14
- **Directories**: 4
- **Estimated lines of code**: ~950 lines
- **Critical Components**: 64-round compression loop, Big-endian word loading, Merkle-Damgård additive chaining.

📚 Beyond the Atlas: Further Reading

🏛️ The Formal Specification

Spec: NIST FIPS 180-4: Secure Hash Standard (SHS)
Why: This is the ultimate source of truth. Every constant, rotation count, and padding rule originates here.
Pedagogical Timing: **Read before Milestone 1.** You should keep Section 5 (Preprocessing) and Section 6 (SHA-256) open as a constant reference while coding.

🏗️ Architectural Foundations

Paper: A Certified Digital Signature by Ralph Merkle (1979)
Why: The "Merkle" in Merkle-Damgård. This thesis established the foundation for tree-based hashing and iterative compression.
Pedagogical Timing: **Read before Milestone 1.** It provides the theoretical "why" behind the block-based processing you are about to implement.

Best Explanation: Chapter 2: Designing Hash Functions in *Serious Cryptography* by Jean-Philippe Aumasson.
Why: Aumasson provides the clearest modern explanation of why SHA-256 uses specific bitwise functions (Ch, Maj) to achieve "confusion and diffusion."
Pedagogical Timing: **Read before Milestone 3.** It will help you see the compression function as a logical machine rather than a wall of magic math.

🔢 Data Alignment & Endianness

Best Explanation: On Holy Wars and a Plea for Peace by Danny Cohen (1980)
Why: The classic IEN 137 paper that coined "Big-Endian" and "Little-Endian." Essential for understanding why you must manually swap bytes in C.
Pedagogical Timing: **Read during Milestone 1.** Read it when you encounter the Big-Endian length field requirement to understand the historical context of the "byte order" problem.

🛠️ Production-Grade Implementation

Code: OpenSSL: crypto/sha/sha256.c
Why: The industry standard. Note how they handle "unrolling" the 64 rounds for performance and how they manage the internal buffer state.
Pedagogical Timing: **Read after Milestone 4.** Comparing your "clean" implementation to OpenSSL's highly optimized (and sometimes obscure) C will show you how production libraries squeeze out every CPU cycle.

Code: BearSSL: src/hash/sha2.c
Why: Thomas Pornin's BearSSL is famous for being "painfully readable" and secure. It is the best example of "Academic-quality C."
Pedagogical Timing: **Read during Milestone 3.** Use this as a reference if you get stuck on the rotation logic; the code is much easier to follow than OpenSSL's.

🛡️ Vulnerabilities & Extensions

Best Explanation: Everything you need to know about Hash Length Extension Attacks by Ron Bowes.
Why: This is the "Adversary's View." It explains why the Merkle-Damgård construction allows an attacker to add data to a hash without knowing the original message.
Pedagogical Timing: **Read after Milestone 4.** This will explain why you should never use `SHA256(key + message)` for authentication.

Spec: RFC 2104: HMAC: Keyed-Hashing for Message Authentication
Why: The solution to the length extension attack. It defines how to wrap your SHA-256 function to create a secure Message Authentication Code.
Pedagogical Timing: **Read after Milestone 4.** This is the natural "Next Project" after finishing your base hash implementation.

₿ Real-World Application: Bitcoin

Paper: Bitcoin: A Peer-to-Peer Electronic Cash System by Satoshi Nakamoto (2008)
Why: SHA-256's most famous application. Specifically, Section 4 (Proof-of-Work) describes the "Double-SHA256" mechanism.
Pedagogical Timing: **Read after Milestone 4.** It transforms your code from an abstract utility into the engine that secures a trillion-dollar asset class.

Best Explanation: Chapter 8: Mining and Consensus in *Mastering Bitcoin* by Andreas Antonopoulos.
Why: Visualizes how individual transaction hashes are combined into a "Merkle Tree" to produce a single block header hash.
Pedagogical Timing: **Read after Milestone 4.** It provides a grand-scale view of how your Milestone 4 `sha256_update` calls are used to build massive data structures.