UNIT 8 - Final Code – SHA 256

PAVAN POLISETTY

F00572483

function SHA256(message):

# Pad and break message

padded\_message = pad\_message(message)

blocks = break\_message(padded\_message)

# Initialize hash values

H[0] = ... (Hexadecimal constant)

... (Initialize remaining H values)

# Round function for each block

for i in range(len(blocks)):

m\_i = blocks[i]

W\_t = calculate\_W\_t(m\_i)

for t in range(64):

K\_t = round\_constant\_table[t]

a = H[7]

b = H[6]

... (Series of modular addition and bitwise operations)

H = update\_hash\_values(H, a, b, ..., K\_t, W\_t)

# Final hash

return concatenate(H)

function pad\_message(message):

... (Pad message with "1", zeros, and message length)

function break\_message(padded\_message):

... (Break message into 512-bit blocks)

function calculate\_W\_t(m\_i):

... (Calculate W\_t from m\_i)

function update\_hash\_values(H, a, b, ..., K\_t, W\_t):

... (Update H values using calculated results)

<https://github.com/chrisveness/crypto/blob/master/sha256.js>

#include <stdlib.h>

#include <memory.h>

#include "sha256.h"

#define ROTLEFT(a,b) (((a) << (b)) | ((a) >> (32-(b))))

#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))

#define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))

#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

#define EP0(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))

#define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))

#define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))

#define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))

static const WORD 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

};

void sha256\_transform(SHA256\_CTX \*ctx, const BYTE data[])

{

WORD a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];

for (i = 0, j = 0; i < 16; ++i, j += 4)

m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);

for ( ; i < 64; ++i)

m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

a = ctx->state[0];

b = ctx->state[1];

c = ctx->state[2];

d = ctx->state[3];

e = ctx->state[4];

f = ctx->state[5];

g = ctx->state[6];

h = ctx->state[7];

for (i = 0; i < 64; ++i) {

t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];

t2 = EP0(a) + MAJ(a,b,c);

h = g;

g = f;

f = e;

e = d + t1;

d = c;

c = b;

b = a;

a = t1 + t2;

}

ctx->state[0] += a;

ctx->state[1] += b;

ctx->state[2] += c;

ctx->state[3] += d;

ctx->state[4] += e;

ctx->state[5] += f;

ctx->state[6] += g;

ctx->state[7] += h;

}

void sha256\_init(SHA256\_CTX \*ctx)

{

ctx->datalen = 0;

ctx->bitlen = 0;

ctx->state[0] = 0x6a09e667;

ctx->state[1] = 0xbb67ae85;

ctx->state[2] = 0x3c6ef372;

ctx->state[3] = 0xa54ff53a;

ctx->state[4] = 0x510e527f;

ctx->state[5] = 0x9b05688c;

ctx->state[6] = 0x1f83d9ab;

ctx->state[7] = 0x5be0cd19;

}

void sha256\_update(SHA256\_CTX \*ctx, const BYTE data[], size\_t len)

{

WORD i;

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

ctx->data[ctx->datalen] = data[i];

ctx->datalen++;

if (ctx->datalen == 64) {

sha256\_transform(ctx, ctx->data);

ctx->bitlen += 512;

ctx->datalen = 0;

}

}

}

void sha256\_final(SHA256\_CTX \*ctx, BYTE hash[])

{

WORD i;

i = ctx->datalen;

// Pad whatever data is left in the buffer.

if (ctx->datalen < 56) {

ctx->data[i++] = 0x80;

while (i < 56)

ctx->data[i++] = 0x00;

}

else {

ctx->data[i++] = 0x80;

while (i < 64)

ctx->data[i++] = 0x00;

sha256\_transform(ctx, ctx->data);

memset(ctx->data, 0, 56);

}

// Append to the padding the total message's length in bits and transform.

ctx->bitlen += ctx->datalen \* 8;

ctx->data[63] = ctx->bitlen;

ctx->data[62] = ctx->bitlen >> 8;

ctx->data[61] = ctx->bitlen >> 16;

ctx->data[60] = ctx->bitlen >> 24;

ctx->data[59] = ctx->bitlen >> 32;

ctx->data[58] = ctx->bitlen >> 40;

ctx->data[57] = ctx->bitlen >> 48;

ctx->data[56] = ctx->bitlen >> 56;

sha256\_transform(ctx, ctx->data);

// Since this implementation uses little endian byte ordering and SHA uses big endian,

// reverse all the bytes when copying the final state to the output hash.

for (i = 0; i < 4; ++i) {

hash[i] = (ctx->state[0] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 4] = (ctx->state[1] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 8] = (ctx->state[2] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 12] = (ctx->state[3] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 16] = (ctx->state[4] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 20] = (ctx->state[5] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 24] = (ctx->state[6] >> (24 - i \* 8)) & 0x000000ff;

hash[i + 28] = (ctx->state[7] >> (24 - i \* 8)) & 0x000000ff;

}

}

import hashlib

def SHA256(message):

  """

  Calculates the SHA-256 hash of a message.

  Args:

    message: The message to be hashed.

  Returns:

    The SHA-256 hash of the message.

  """

  # 1. Convert the message to bytes

  message\_bytes = bytes(message, encoding='utf-8')

  # 2. Create a SHA-256 hash object

  hasher = hashlib.sha256()

  # 3. Update the hash object with the message

  hasher.update(message\_bytes)

  # 4. Get the SHA-256 digest

  digest = hasher.digest()

  # 5. Convert the digest to a hex string

  hash\_hex = digest.hex()

  return hash\_hex

# Download the book of Mark

book\_url = "https://quod.lib.umich.edu/cgi/r/rsv/rsv-idx?type=DIV1&byte=4697892"

response = requests.get(book\_url)

book\_text = response.text

# 6. Calculate the SHA-256 hash of the book text

book\_hash = SHA256(book\_text)

# 7. Print the hash

print("SHA-256 hash of the book of Mark:", book\_hash)

HASH 93674c3b338bf0aa7819f16219339a3b93a710acbc88c7fd49f8a700f96c9825