src\emd.py

```python
import numpy as np
from bitarray import bitarray
from src.arithmetic_coder import arithmetic_encode, arithmetic_decode
from src.hamming import encode_bytes, decode_bits_to_bytes

BLOCK_SIZE = 16
RESERVED_PIXELS = 6

def divide_into_blocks(image):
    h, w = image.shape
    assert h % BLOCK_SIZE == 0 and w % BLOCK_SIZE == 0, "Image dimensions must be divisible by 16"
    blocks = image.reshape(h // BLOCK_SIZE, BLOCK_SIZE, w // BLOCK_SIZE, BLOCK_SIZE)
    blocks = blocks.swapaxes(1, 2).reshape(-1, BLOCK_SIZE, BLOCK_SIZE)
    return blocks

def merge_blocks(blocks, h, w):
    merged = blocks.reshape(h // BLOCK_SIZE, w // BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE)
    merged = merged.swapaxes(1, 2).reshape(h, w)
    return merged.astype(np.uint8)

def generate_shadows(cover):
    return cover.copy(), cover.copy()

def embed(cover1, cover2, payload_bytes):
    h, w = cover1.shape
    blocks1 = divide_into_blocks(cover1)
    blocks2 = divide_into_blocks(cover2)

    # Hamming encode
    encoded_bits = encode_bytes(payload_bytes)

    # Group bits into symbols (3 bits → 0-7)
    symbols = [(encoded_bits[i] << 2) | (encoded_bits[i+1] << 1) | encoded_bits[i+2]
               for i in range(0, len(encoded_bits) - 2, 3)]

    # Arithmetic encode
    compressed = arithmetic_encode(symbols)
    compressed_bits = bitarray()
    compressed_bits.frombytes(compressed)
    bitstream = list(compressed_bits)

    idx = 0
    for i in range(len(blocks1)):
        flat1 = blocks1[i].flatten()
        flat2 = blocks2[i].flatten()
        embedded = []

        for j in range(BLOCK_SIZE * BLOCK_SIZE - RESERVED_PIXELS):
            if idx >= len(bitstream): break
            f = (int(flat1[j]) + int(flat2[j])) % 5
            pos = (bitstream[idx] - f) % 5
            if pos == 1: flat1[j] += 1
            elif pos == 2: flat1[j] -= 1
            elif pos == 3: flat2[j] += 1
            elif pos == 4: flat2[j] -= 1
            embedded.append(bitstream[idx])
            idx += 1

        # Store length L in last 6 pixels
        L = len(embedded)
        for k in range(RESERVED_PIXELS):
            flat1[-(k+1)] = (flat1[-(k+1)] & 0xF0) | ((L >> (k * 4)) & 0x0F)

        blocks1[i] = flat1.reshape(BLOCK_SIZE, BLOCK_SIZE)
        blocks2[i] = flat2.reshape(BLOCK_SIZE, BLOCK_SIZE)

    stego1 = merge_blocks(blocks1, h, w)
    stego2 = merge_blocks(blocks2, h, w)
    return stego1, stego2

def extract(stego1, stego2):
    h, w = stego1.shape
    blocks1 = divide_into_blocks(stego1)
    blocks2 = divide_into_blocks(stego2)

    recovered_bits = []
    for i in range(len(blocks1)):
        flat1 = blocks1[i].flatten()
        flat2 = blocks2[i].flatten()

        # Read length L from last 6 pixels
        L = sum((flat1[-(k+1)] & 0x0F) << (k * 4) for k in range(RESERVED_PIXELS))

        for j in range(min(L, BLOCK_SIZE * BLOCK_SIZE - RESERVED_PIXELS)):
            f = (int(flat1[j]) + int(flat2[j])) % 5
            recovered_bits.append(f)

    # Convert recovered bits to bytes
    recovered_bytes = bitarray(recovered_bits).tobytes()

    # Arithmetic decode
    decoded_symbols = arithmetic_decode(recovered_bytes)

    # Convert symbols (0-7) back to bits
    bitstream = []
    for val in decoded_symbols:
        bitstream.extend([(val >> 2) & 1, (val >> 1) & 1, val & 1])

    # Hamming decode
    final_bytes, corrected = decode_bits_to_bytes(bitstream)
    return final_bytes
```