Sonic Security: Real-Time Audio Encryption Design Document

Keep your sound under wraps 🎁

Sonic Security Experts:

Jaewon Lee (jl6367), Tyler Chang (tc3407), Joshua Mathew (jm5915)

1) Introduction

Sonic Security is a hardware-accelerated solution designed to provide real-time audio encryption using the Terasic DE1-SoC platform. Our goal is to take clear, high-quality WAV files and transform them into secure, encrypted data—ensuring that only authorized users with the correct key can decode the original audio. The user will also have the option of recording live audio through an INMP441 microphone module, which will then be processed in real-time to be encrypted. By leveraging the parallel processing capability of our FPGA, we have put forward a design that minimizes latency while delivering robust encryption performance using an AES-128 core.

2) Algorithms

1) Brief Intro to AES-128:

The Advanced Encryption Standard (AES) is a symmetric block cipher formally adopted as a U.S. federal standard in 2001. It was the result of a multi-year NIST competition to find a successor to the older DES cipher, which had become insecure due to its short 56-bit key. AES as standardized has a fixed block size of 128 bits and supports key sizes of 128, 192, or 256. We focus on AES-128, the variant with a 128-bit (16-byte) key. Despite its shorter key, AES-128 is still considered highly secure.

```python
# Convert bytes to state matrix
def bytes_to_state(data):
    state = [[0 for _ in range(4)] for _ in range(4)]
    for i in range(4):
        for j in range(4):
            state[j][i] = data[i * 4 + j]
    return state

# Convert state matrix to bytes
def state_to_bytes(state):
    output = bytearray(16)
    for i in range(4):
        for j in range(4):
            output[i * 4 + j] = state[j][i]
    return output
```

byte_to_state(data) and state_to_bytes(state) in Python

AES's design is mathematically rooted on arithmetic in finite fields. All byte-wise operations occur in the finite field **GF**(2^8) (Galois Field of 2^8 elements), where operations such as addition and multiplication are performed modulo an irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

2) Explanation of Each AES-128 Stage

AES-128 operates on a 128-bit block of data which is conceptually organized as a 4x4 matrix of bytes called the *state*. For clarity, we can label the bytes of the state as $s_{r,c}$ with r, c $\in \{0, 1, 2, 3\}$, where *r* is the row index and *c* is the column index. The input 16-byte plaintext is initially mapped into this state matrix in column-major order: the first 4 bytes form the first column of the state, the next 4 bytes form the second column, and so on. Likewise, during output, the 4x4 state matrix is flattened back to 16 bytes in column-major order to produce the ciphertext.

SubBytes (Byte Substitution): The SubBytes stage is a non-linear byte-wise substitution that provides confusion (aka non-linearity) in AES. Each byte of the state is independently replaced using an 8 × 8 substitution box (S-box). The AES S-box is constructed by composing two mathematical operations in **GF**(2^8): first by taking the multiplicative inverse of the byte (except that 0 is mapped to 0), then applying a fixed affine transformation. The result is a set fixed

permutation of the 256 possible byte values; for example, a byte value '0x53' is substituted with '0xED' in the AES S-box. Our team's implementation uses a precomputed table 'SBOX[0..255]' containing these substitutions. Applying SubBytes means $s_{r,c} = S(s_{r,c})$ for each byte of the state. This transformation is invertible by our inverse S-box table (which is used in decryption) which maps each output byte back to its original value. Our Python implementation defines the S-box as a static list of 256 byte values and *sub_bytes*(state) iterates through all 16 state bytes—replacing each with the corresponding S-box entry.

```python
# AES S-box
SBOX = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
]

# AES Inverse S-box
INV_SBOX = [
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
]
```

S-box Tables in Python

```python
# SubBytes transformation
def sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = SBOX[state[i][j]]
    return state

# InvSubBytes transformation
def inv_sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = INV_SBOX[state[i][j]]
    return state
```

sub_bytes(state) and *inv_sub_bytes(state)* functions in Python

ShiftRows (Row Rotation): The ShiftRows transformation is a cyclic row shift that provides diffusion by permuting the byte positions in the state. The first row (r=0) is left unchanged. However, the second row (r=2) is cyclically left-shifted by 1 byte position, the third row by 2, and the fourth by 3. Essentially, the byte at position $s_{r,c}$ moves to position $s_{r,(c-r) \bmod 4}$ after shifting (for r>0). For example, before ShiftRows, the second row has bytes ($s_{1,0}$, $s_{1,1}$, $s_{1,2}$, $s_{1,3}$); but after a left

rotate by 1, it becomes ($s_{1,1}$, $s_{1,2}$, $s_{1,3}$, $s_{1,0}$). The inverse ShiftRows (for decryption) rotates each non-first row in the opposite direction (to the right) by the same amount to undo the shift.

```python
# ShiftRows transformation
def shift_rows(state):
    state[1] = state[1][1:] + state[1][:1]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][3:] + state[3][:3]
    return state

# InvShiftRows transformation
def inv_shift_rows(state):
    state[1] = state[1][3:] + state[1][:3]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][1:] + state[3][:1]
    return state
```

shift_rows(state) and *inv_shift_rows(state)* in Python

MixColumns (Column Mixing): MixColumns is a linear mixing operation that operates on each column of our state, viewed as a four-term polynomial over **GF**(2^8). Each column (4 bytes) is transformed by multiplying it with a fixed 4x4 matrix over **GF**(2^8). In standard AES, the transformation in polynomial form takes a column vector ($s_{0,c}$, $s_{1,c}$, $s_{2,c}$, $s_{3,c}$)^T and produces a new column (s'$_{0,c}$, s'$_{1,c}$, s'$_{2,c}$, s'$_{3,c}$)^T, which is given by:

$$\begin{pmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{pmatrix},$$

where our arithmetic is done in **GF**(2^8) and constants 1,2,3 represent field elements (in this case: 0x01, 0x02, 0x03 in hex). In practice though, this means:

$$\begin{aligned} s'_{0,c} &= (2 \cdot s_{0,c}) \oplus (3 \cdot s_{1,c}) \oplus (1 \cdot s_{2,c}) \oplus (1 \cdot s_{3,c}), \\ s'_{1,c} &= (1 \cdot s_{0,c}) \oplus (2 \cdot s_{1,c}) \oplus (3 \cdot s_{2,c}) \oplus (1 \cdot s_{3,c}), \\ s'_{2,c} &= (1 \cdot s_{0,c}) \oplus (1 \cdot s_{1,c}) \oplus (2 \cdot s_{2,c}) \oplus (3 \cdot s_{3,c}), \\ s'_{3,c} &= (3 \cdot s_{0,c}) \oplus (1 \cdot s_{1,c}) \oplus (1 \cdot s_{2,c}) \oplus (2 \cdot s_{3,c}) \ , \end{aligned}$$

where $\cdot$ denotes multiplication in **GF**(2^8) and $\oplus$ is byte-wise XOR (field addition). The fixed matrix essentially mixes each byte with its neighbors in the column, which ensures that a change in one byte of the state affects all four bytes of that column in the next round. As you can see, our Python implementation of *mix_columns(state)* follows this formula—using *gmul(a,b)* to multiply bytes by the constants 2 and 3 in **GF**(2^8), then XORing the results accordingly. This function also implements multiplication via the "Russian peasant" multiplication, iteratively accumulating the result *p* by XORing *a* into *p* whenever the least significant bit of *b* is 1, then repeatedly doubles *a* (with a polynomial reduction via XOR with 0x1B when *a* overflows 8 bits). This accomplishes the

modulo *m(x)* multiplication. The inverse MixColumns (used in decryption) multiplies by the matrix inverse, which corresponds to constants {0x0E, 0x0B, 0x0D, 0x09} in **GF**(2^8)(these are 14, 11, 13, 9 in decimal) such that the original column is recovered.

```python
# Galois Field multiplication
def gmul(a, b):
    p = 0
    for _ in range(8):
        if b & 1:
            p ^= a
        high_bit_set = a & 0x80
        a <<= 1
        if high_bit_set:
            a ^= 0x1b  # XOR with the irreducible polynomial x^8 + x^4 + x^3 + x + 1
        b >>= 1
    return p & 0xff
```

gmul(a,b) in Python

```python
# MixColumns transformation
def mix_columns(state):
    for i in range(4):
        s0 = state[0][i]
        s1 = state[1][i]
        s2 = state[2][i]
        s3 = state[3][i]

        state[0][i] = gmul(0x02, s0) ^ gmul(0x03, s1) ^ s2 ^ s3
        state[1][i] = s0 ^ gmul(0x02, s1) ^ gmul(0x03, s2) ^ s3
        state[2][i] = s0 ^ s1 ^ gmul(0x02, s2) ^ gmul(0x03, s3)
        state[3][i] = gmul(0x03, s0) ^ s1 ^ s2 ^ gmul(0x02, s3)
    return state

# InvMixColumns transformation
def inv_mix_columns(state):
    for i in range(4):
        s0 = state[0][i]
        s1 = state[1][i]
        s2 = state[2][i]
        s3 = state[3][i]

        state[0][i] = gmul(0x0e, s0) ^ gmul(0x0b, s1) ^ gmul(0x0d, s2) ^ gmul(0x09, s3)
        state[1][i] = gmul(0x09, s0) ^ gmul(0x0e, s1) ^ gmul(0x0b, s2) ^ gmul(0x0d, s3)
        state[2][i] = gmul(0x0d, s0) ^ gmul(0x09, s1) ^ gmul(0x0e, s2) ^ gmul(0x0b, s3)
        state[3][i] = gmul(0x0b, s0) ^ gmul(0x0d, s1) ^ gmul(0x09, s2) ^ gmul(0x0e, s3)
    return state
```

mix_columns(state) and *inv_mix_columns(state)* in Python

AddRoundKey (Key Mixing): In AddRoundKey, our 128-bit round key is XORed with the state. Since XOR in binary is the group addition operation in **GF**(2), this stage here combines the current data with the round's subkey. The round key also follows a 4x4 byte matrix conceptually, derived from the cipher key via the Key Expansion algorithm which is discussed below. AddRoundKey is quite straightforward: each byte of the state $s_{r,c}$ is replaced by $s_{r,c} \oplus k_{r,c}$ where $k_{r,c}$ is the corresponding byte of the round key. This is the only stage in AES that incorporates the secret key, and it also ensures that each round's output depends on the key. In our Python implementation, *add_round_key(state, round_key)* loops through the 4x4 matrix and XORs each state byte with the corresponding round key byte. Notice that XOR is its own inverse, so the inverse of this function (for the purpose of decryption) would be identical to encryption.

```python
# AddRoundKey transformation
def add_round_key(state, round_key):
    for i in range(4):
        for j in range(4):
            state[i][j] ^= round_key[i][j]
    return state
```

add_round_key(state, round_key) in Python

Key Expansion (Key Schedule): AES-128 uses a key schedule to derive 11 round keys (each 128 bits) from the initial cipher key of 128 bits. The key schedule is vital for security purposes as it ensures that each round uses a different key while being efficiently computable at the same time. The input key is divided into four 32-bit words: W[0..3]. The algorithm then generates new words W[i] for i=4 to 43 (since AES-128 requires 4 x (10+1) = 44 words for 11 round keys). Each new word is either the XOR of the previous word and the word four positions back, or, for the position that are multiples of four, a transformed version of the previous word XORed with the word four back. More formally, for i ≥ 4:

$$\text{If } i \mod 4 \neq 0, \text{ then } W[i] = W[i-4] \oplus W[i-1].$$
$$\text{If } i \mod 4 = 0, \text{ then } W[i] = W[i-4] \oplus \texttt{SubWord}(\texttt{RotWord}(W[i-1])) \oplus Rcon[i/4].$$

Here, RotWord takes a 4-byte word (a_0, a_1, a_2, a_3) and cyclically rotates it to (a_1, a_2, a_3, a_0). SubWord applies our AES S-box to each of the 4 bytes of its word input (just like SubBytes does to state bytes). Rcon[j] is a round constant word for the j^{th} round, which is defined as (R_j, 0x00, 0x00, 0x00), with R_j being an element in **GF**(2^8) that exponentes 2 to the power (j-1). In hexadecimal, the sequence of R_j for AES-128 rounds j=1 to 10 is 01, 02, 04, 08, 10, 20, 40, 80, 1B, 36. For example, R_1 = 0x01, R_2=0x02, R_3=0x04, etc. where each is essentially 2^{j-1} in **GF**(2^8) modulo our irreducible polynomial. These constants break symmetry between rounds, albeit in a non-repetitive yet predictable way. In Python, the *expand_key(key)* function implements this schedule by starting from the 16-byte kley and computing all round key matrices. The helper function *key_schedule_core(word, iteration)* performs the RotWord, SubWord (via our handy S-box), and XOR with the appropriate Rcon byte for the given iteration. The expanded key results in a list of round keys round_keys[0]...round_keys[10], each of which is a 4x4 byte matrix suitable for the AddRoundKey step in each round.

```python
# Round constants for key schedule
RCON = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36]
```

Rcon Array in Python

```python
# Key schedule core function
def key_schedule_core(word, iteration):
    # Rotate left by one byte
    word = word[1:] + word[:1]

    # Apply S-box to all bytes
    for i in range(len(word)):
        word[i] = SBOX[word[i]]

    # XOR with round constant on the first byte
    word[0] ^= RCON[iteration]

    return word
```

key_schedule_core(word, iteration) in Python

```python
# Expand key for all rounds
def expand_key(key, rounds=10):
    # Convert key to words (4-byte chunks)
    key_words = [key[i:i+4] for i in range(0, len(key), 4)]

    # Expand to get words for all rounds
    expanded_key_words = list(key_words)
    for i in range(len(key_words), 4 * (rounds + 1)):
        temp = list(expanded_key_words[i-1])

        if i % len(key_words) == 0:
            temp = key_schedule_core(temp, i // len(key_words) - 1)

        for j in range(4):
            temp[j] ^= expanded_key_words[i-len(key_words)][j]

        expanded_key_words.append(temp)

    # Convert expanded key words to round keys (4x4 matrices)
    round_keys = []
    for i in range(0, len(expanded_key_words), 4):
        round_key = [[] for _ in range(4)]
        for j in range(4):
            for k in range(4):
                round_key[k].append(expanded_key_words[i+j][k])
        round_keys.append(round_key)

    return round_keys
```

expand_key(key, rounds=10) in Python

3) Round Structure

An AES-128 encryption consists of an initial key addition, followed by 9 full rounds, and a final round which omits the MixColumns step. We can summarize the sequence of transformation steps as:

- Initial Round: AddRoundKey using round key 0 (the original cipher key)
- Rounds 1-9: Each round consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey (in that order) using round keys 1-9
- Round 10 (Final Round): SubBytes, ShiftRows, and AddRoundKey (using round key 10). MixColumns is not performed in the final round. This is because after the last AddRoundKey, there is no need for further mixing (the ciphertext is the output).

```python
# AES Encryption function
def aes_encrypt_block(data, key):
    state = bytes_to_state(data)

    # Generate round keys
    round_keys = expand_key(key)

    # Initial round key addition
    state = add_round_key(state, round_keys[0])

    # Main rounds
    for i in range(1, 10):
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state, round_keys[i])

    # Final round (no MixColumns)
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, round_keys[10])

    return state_to_bytes(state)
```

aes_encrypt_block(data, key) in Python

Decryption follows the inverse sequence, starting with the final round key and applying inverse transformations in reverse order (AddRoundKey, InvShiftRows, InvSubBytes, etc.), with an analogous structure of 10 rounds.

```python
# AES Decryption function
def aes_decrypt_block(data, key):
    state = bytes_to_state(data)

    # Generate round keys
    round_keys = expand_key(key)

    # Initial round key addition
    state = add_round_key(state, round_keys[10])

    # Main rounds
    for i in range(9, 0, -1):
        state = inv_shift_rows(state)
        state = inv_sub_bytes(state)
        state = add_round_key(state, round_keys[i])
        state = inv_mix_columns(state)

    # Final round (no MixColumns)
    state = inv_shift_rows(state)
    state = inv_sub_bytes(state)
    state = add_round_key(state, round_keys[0])

    return state_to_bytes(state)
```

aes_decrypt_block(data, key) in Python

4) ECB Mode Implementation

Our AES-128 implementation is used in Electronic Codebook (ECB) mode of operation for encrypting data, particularly WAV files. ECB is the simplest block cipher mode: the plaintext is divided into independent 16-byte blocks, and each block is encrypted separately with the same key. In the context of this project, ECB was chosen for simplicity and because it allows for straightforward parallelization (since each block encryption is independent, multiple blocks can be processed in parallel in FPGA design, and there is no feedback dependency between blocks).

Before encryption, data that is not an exact multiple of 16 bytes must be padded. We employ the standard PKCS#7 padding scheme to ensure the plaintext length is a multiple of the AES block size. In PKCS#7 padding, if *n* bytes of padding are required (1 to 16 bytes), each of those *n* bytes are set to the value *n*. For example, if the plaintext is 5 bytes short of a 16-byte multiple, 5 bytes of value 0x05 will be appended. If the plaintext length is already exactly a multiple of 16, a full 16 bytes of value 0x10 are added as padding (this unambiguously indicates padding as well). In our Python implementation, we check the length of the final block and, if it is shorter than 16 bytes, we compute the needed padding length and append the padding bytes. On decryption, the code verifies the padding by examining the last byte to see how many padding bytes should be removed, and confirming that all of them have the expected value. Proper handling of padding is necessary to recover the exact original plaintext after decryption.

```python
# ECB Mode encryption
def encrypt_ecb(data, key):
    # Split data into blocks of 16 bytes
    blocks = [data[i:i+16] for i in range(0, len(data), 16)]

    # Pad the last block if necessary (PKCS#7 padding)
    last_block_len = len(blocks[-1])
    if last_block_len < 16:
        padding_length = 16 - last_block_len
        blocks[-1] = blocks[-1] + bytes([padding_length]) * padding_length

    # Encrypt each block independently
    encrypted_blocks = []
    for block in blocks:
        encrypted_block = aes_encrypt_block(block, key)
        encrypted_blocks.append(encrypted_block)

    # Concatenate all encrypted blocks
    return b''.join(encrypted_blocks)
```

encrpyt_ecb(data, key) in Python

```python
# ECB Mode decryption
def decrypt_ecb(data, key):
    # Split data into blocks of 16 bytes
    blocks = [data[i:i+16] for i in range(0, len(data), 16)]

    # Decrypt each block independently
    decrypted_blocks = []
    for block in blocks:
        decrypted_block = aes_decrypt_block(block, key)
        decrypted_blocks.append(decrypted_block)

    # Concatenate all decrypted blocks
    result = b''.join(decrypted_blocks)

    # Remove padding
    padding_length = result[-1]
    if padding_length > 0 and padding_length <= 16:
        # Verify padding (all padding bytes should be the same)
        padding = result[-padding_length:]
        if all(p == padding_length for p in padding):
            return result[:-padding_length]

    # Return the result without removing padding if padding is invalid
    return result
```

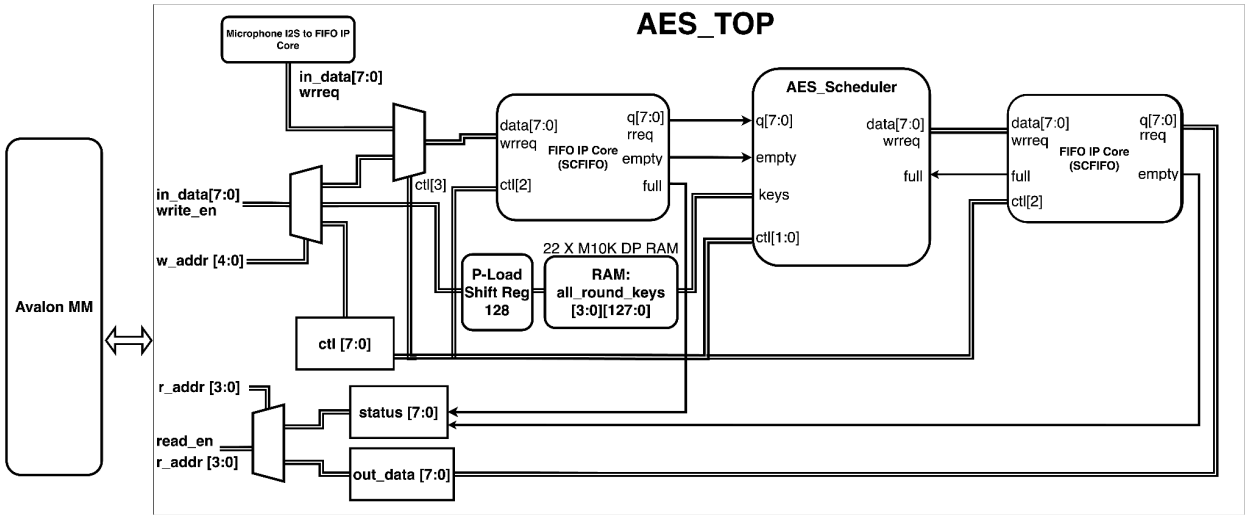
decrypt_ecb(data, key) in Python

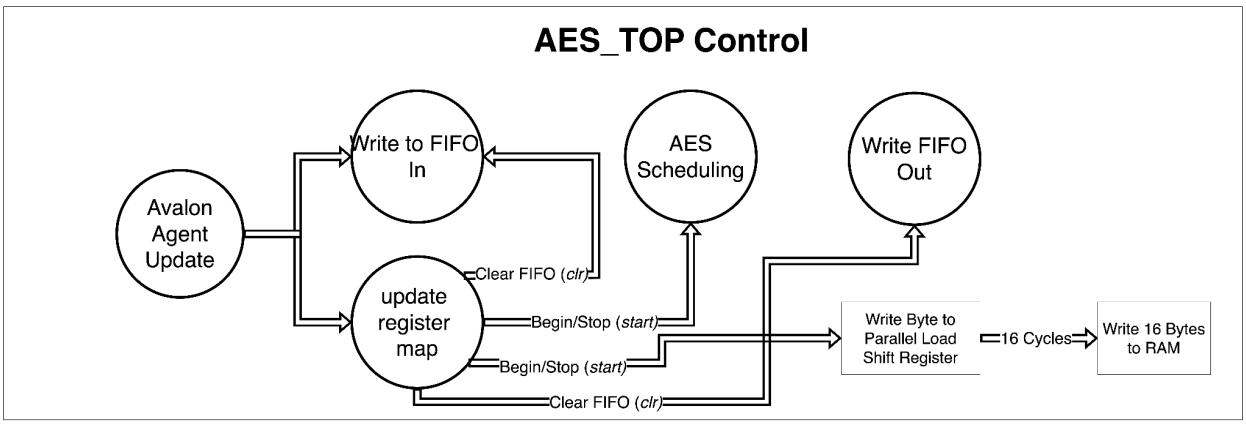
In sum, our implementation in ECB mode will:

1. Split the input plaintext (e.g., raw WAV file bytes) into 16-byte blocks
2. Pad the last block with PKCS#7 if necessary to reach 16 bytes
3. Encrypt each block independently with AES-128
4. Concatenate all ciphertext blocks to produce the final output

3) System Block Diagrams

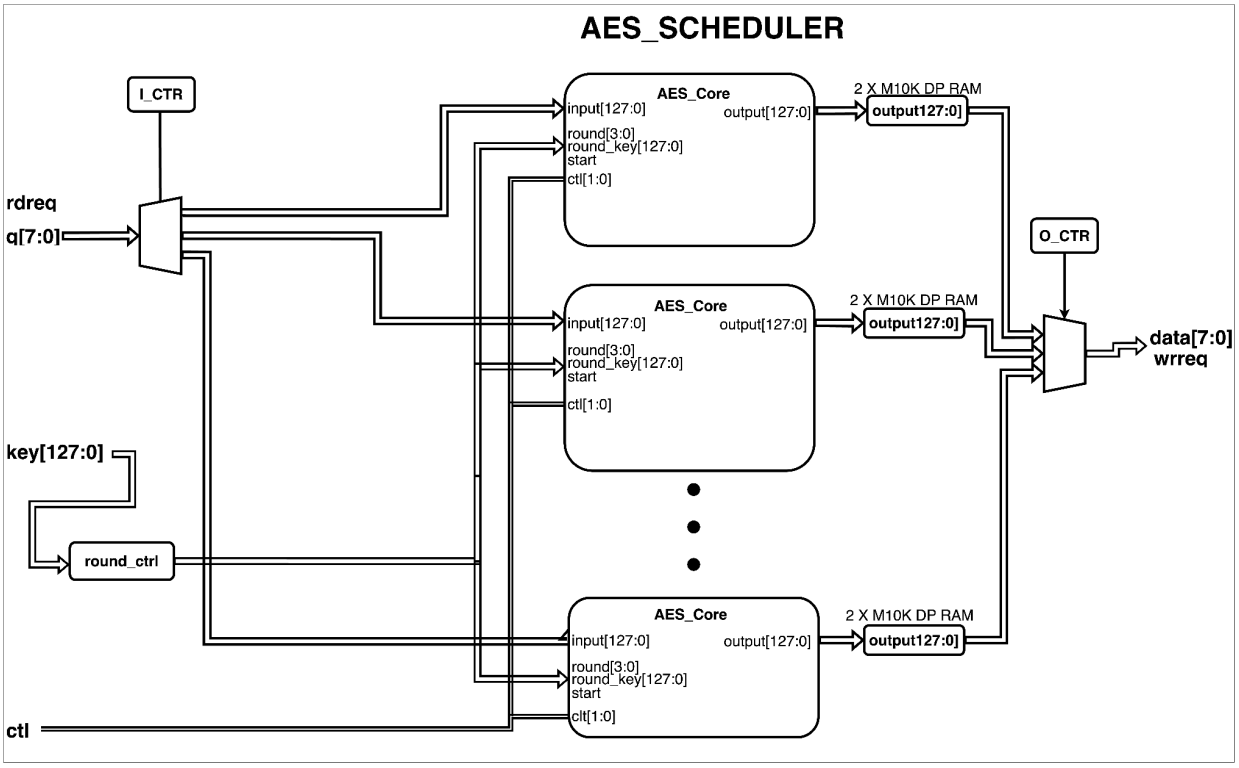
1) AES_Top

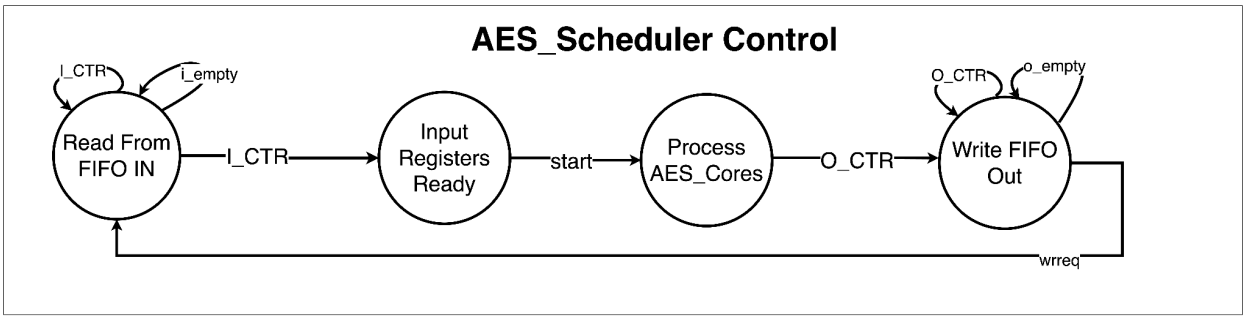




This module serves as the interface between the Avalon MM interface and the internal AES modules, and therefore defines the register map and handles logic regarding the HW's modes of operation. Of importance, it controls whether the input to the downstream AES algorithm will be streamed from an attached microphone or fed through the driver and register map. This module also stitches together the inputs and outputs of our AES_Scheduler module with the FIFOs that come before and after it. We will use the SCFIFO IP provided by Intel with a depth of N * 1k bits (N = number of parallel cores instantiated) because it seems reasonable for our purposes.

2) AES_Scheduler

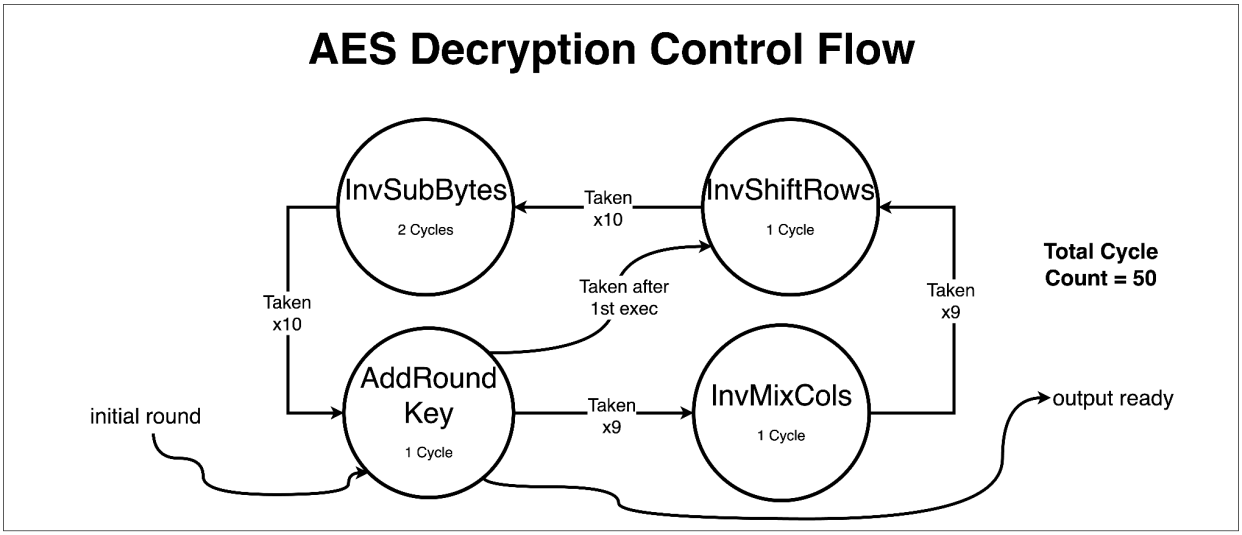


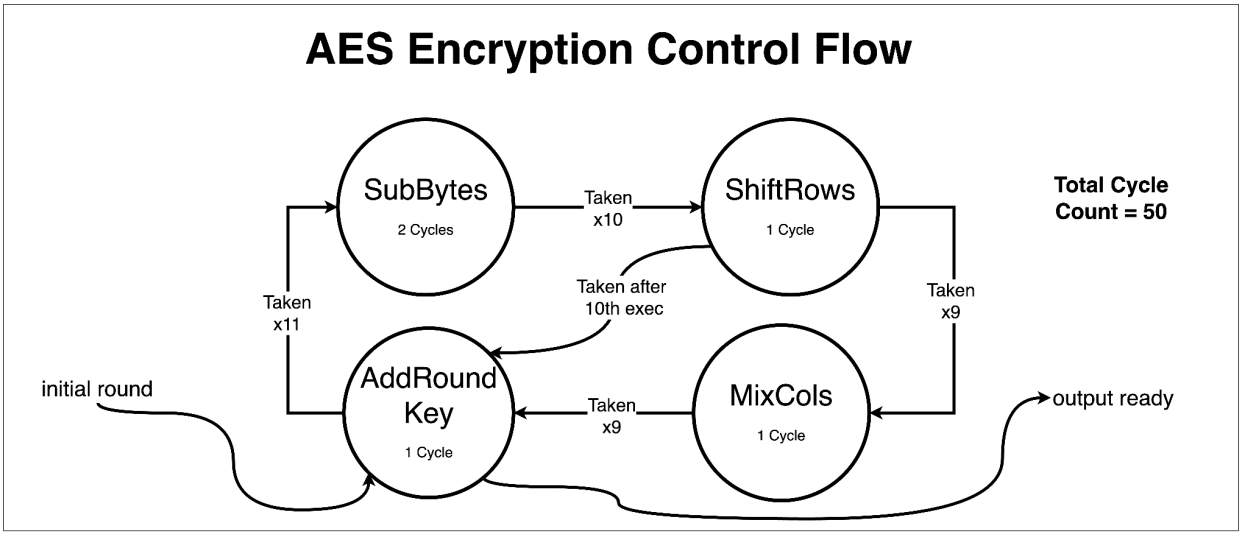


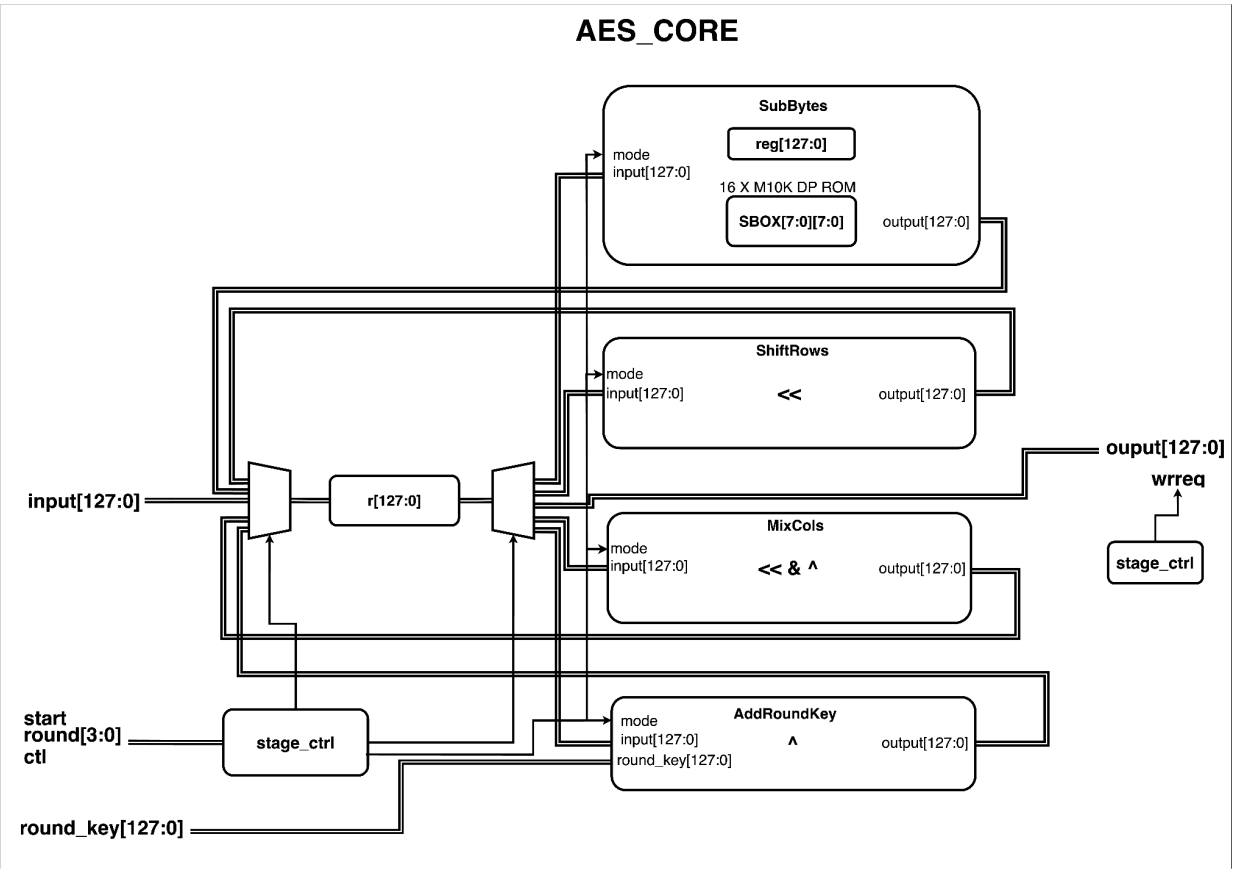
The main operation of AES_Scheduler is to maintain the correct ordering of blocks processed by the AES_Core(s). The byte-wide *q* input for this module will be a read (using *rdreq*) in from a FIFO IP and the scheduler will fill the *input* registers for each core in an ordered fashion. Once all core's *input* registers are filled it will synchronously send a pulse on the *start*. We have designed the cores to provide an encrypted/decrypted output in a predetermined cycle count of 50, and once that has been reached we will begin attempting to write (using *wrreq*) the values in the *output* registers to the output FIFO. We will use counter logic to ensure order is preserved in the byte-by-byte reading and writing with the FIFOs. This logic will also inform when the AES_Core(s) should start in order to make sure that previously computed outputs aren't overwritten.

Another important function of this module is to provide the cores with the correct *round* they should be on as well as the correct *round_key* they should be using. Since we have designed the stages in the core to have a deterministic cycle time, we can update the *round* register based on a counter that begins once the *start* pulse is sent. Furthermore, since the round keys will be the same for every 16 byte block, we have chosen to store these keys in RAM which populates the *round_key* register when the *round* is incremented. The key expansion is done on the driver side and this RAM that stores the keys is within the register space of the module.

3) AES_Core


AES Decryption Control Flow
InvSubBytes
2 Cycles
Taken
x10
InvShiftRows
1 Cycle
Total Cycle
Count = 50
Taken after
1st exec
Taken
x10
Taken
x9
AddRound
Key
1 Cycle
Taken
x9
InvMixCols
1 Cycle
initial round
output ready


AES Encryption Control Flow
SubBytes
2 Cycles
Taken
x10
ShiftRows
1 Cycle
Total Cycle
Count = 50
Taken after
10th exec
Taken
x11
Taken
x9
AddRound
Key
1 Cycle
Taken
x9
MixCols
1 Cycle
initial round
output ready



The AES_Core module encapsulates encryption/decryption on a 128-bit data block.. Depending on the *mode* input, AES_Core will either be in encryption or decryption mode. As described in the Algorithms section, we decompose this core into 4 distinct stages. We use a single local 128-bit wide register to store the intermediary results of each stage. The shift, mix, and add stages all are transformations that can be reduced to a set of XORs and left/right shifts and thus can be done in a single cycle. The substitution stage requires reading from a ROM which we allot 2 cycles for latching. This module's main logic will consist of ensuring correct data flow based on the static scheduling and cycle time of each stage.

4) Resources

Parameters: N (number of cores)
Memory:

Round Keys: 1408 bits (RAM) → 22 M10K Blocks
Substitution Table: N * 2 * 2,048 bits (ROM) → N * 16 M10K Blocks
FIFO: N * 2* 1k bits (FIFO IP) → N * 2 M10K Blocks

Maximum:

M10k Blocks: 445 → 4450 Kbits

Reasonable N = 10 ⇒ 202 M10K Blocks (2020 KBits of memory used)

5) Hardware-Software Interface

The interface between hardware and software will be utilizing the Avalon MM interface in the same way that vga_ball did. We will create drivers for encryption and decryption that at first modify the state of the register map to set the mode of operation and signal start of execution. The driver will also then provide an api to write and read bytes to/from the HW accelerator and the basic user-space application will simply open a WAV file and encrypt it, storing the output into a different WAV file.

1) Register Map

SECTION	REGISTER MAP	
0x0 - 0x8	in_data [7:0]	W
0x8 - 0x10	out_data [7:0]	R
0x10 - 0x18	ctl [7:0]	W
0x18 - 0x20	status [7:0]	R
0x20 - 0x5A0	round_keys [1407:0]	W

status							
7	6	5	4	3	2	1	0
x	x	x	x	x	x	input_full	output_empty

ctl							
7	6	5	4	3	2	1	0
x	x	x	x	clear	mode	input_mode	start

We employ a single byte control register and single byte status register as the primary control interface for the user.

start - High while the AES module should be active, will enable the scheduler to continually schedule FIFO read/writes and start pulses to the core

input_mode - 1 for input data coming from microphone, 0 for data to be streamed from *input_data* in the register map

mode - 0 for encryption mode, 1 for decryption mode

clear - When high will clear the data that exists in the FIFOs

output-empty - High when the output FIFO is empty, if read while empty data will be same as previous read

input-full- High when the input FIFO is full, if written to while full data will NOT be overwritten nor will the written value propagate forward

Note: The per-round-keys will have to be calculated and written to the respective portion of the register map in the following order

[Initial Key] [Round 1 Key] [Round 2 Key] ... [Round 10 Key]

Each round key should be stored in this format: [MSB ...LSB]

2) Logic Flow of File-To-File Encryption