Sonic Security: AES Audio Encryption Accelerator Report

Keep your sound under wraps 🎁

Sonic Security Experts:

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

1) Project Overview

Sonic Security is a hardware-accelerated solution designed to provide 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. 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 key 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

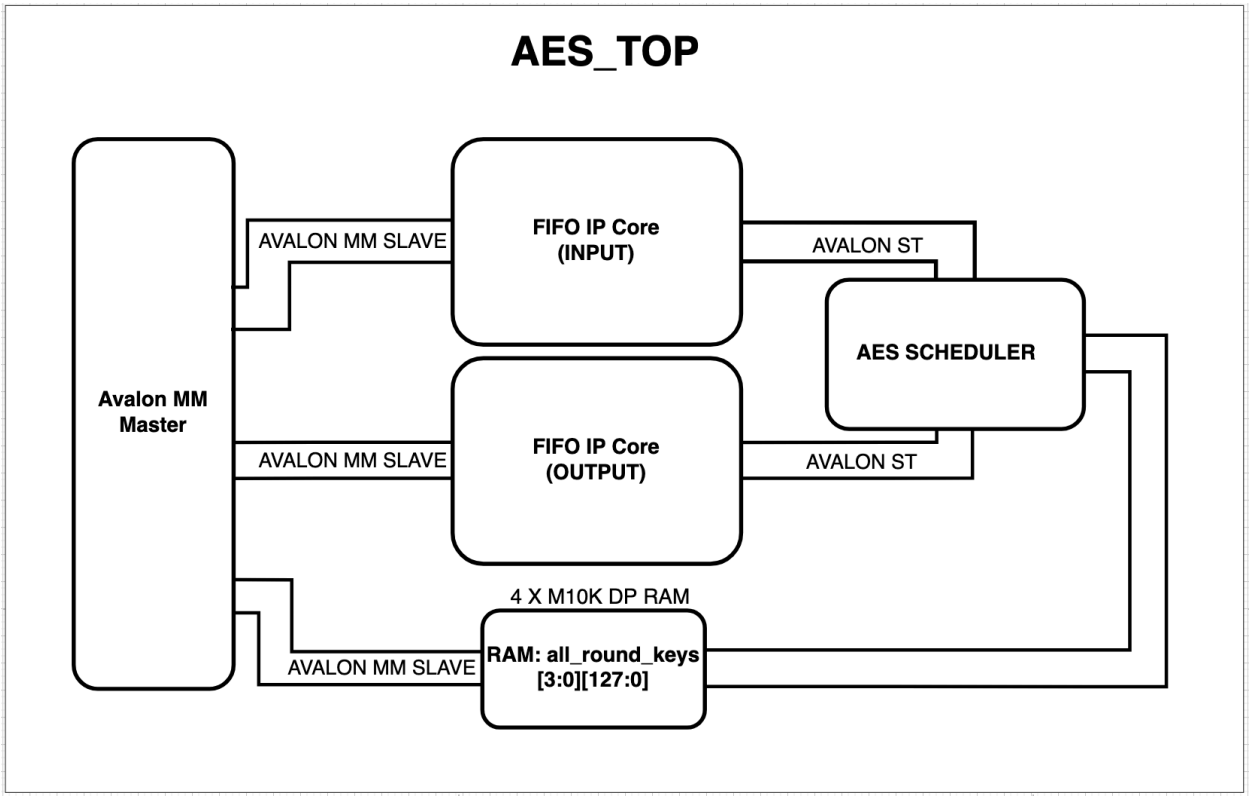
In sum, our implementation in ECB mode will:

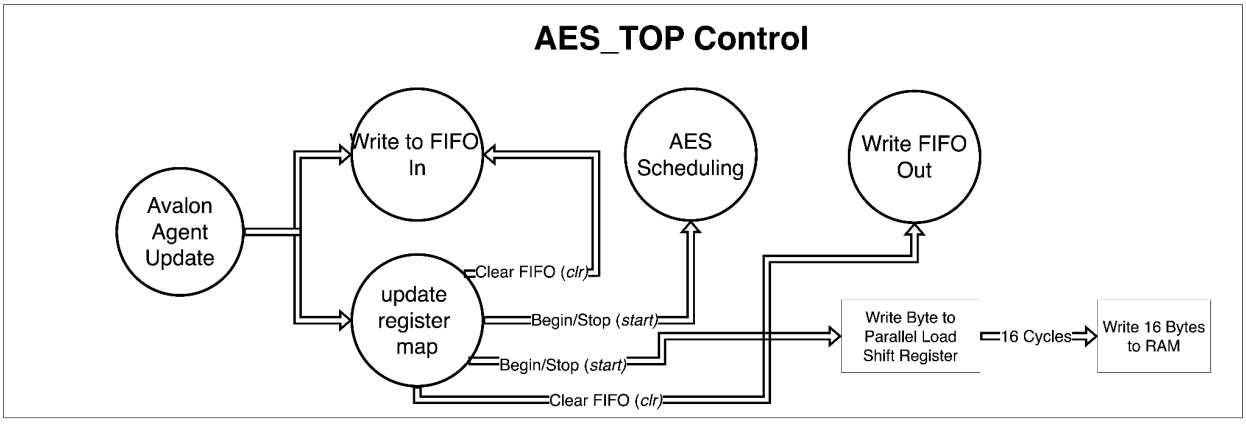
1. Split the input plaintext (e.g., raw WAV file bytes) into 16-byte blocks
2. Pad the last block with PKCS#7 if necessary to reach 16 bytes
3. Encrypt each block independently with AES-128
4. Concatenate all ciphertext blocks to produce the final output

3) System Block Diagrams

1) AES_Top

- Use byte_enable to write to byte specific places in RAM for round key

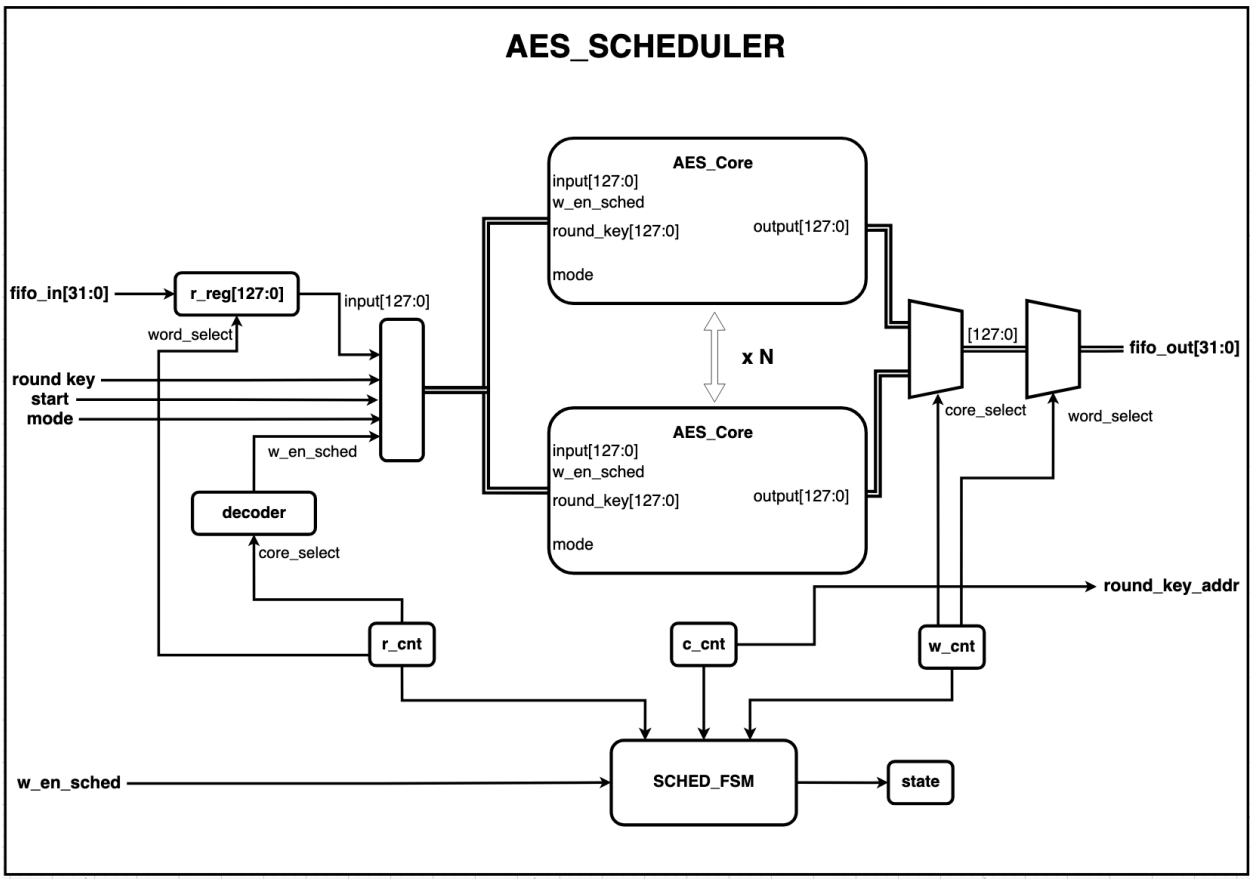




This module serves as the interface between the Avalon MM interface and the internal AES modules, and therefore defines the register map and handles logic regarding the HW's modes of

operation. 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

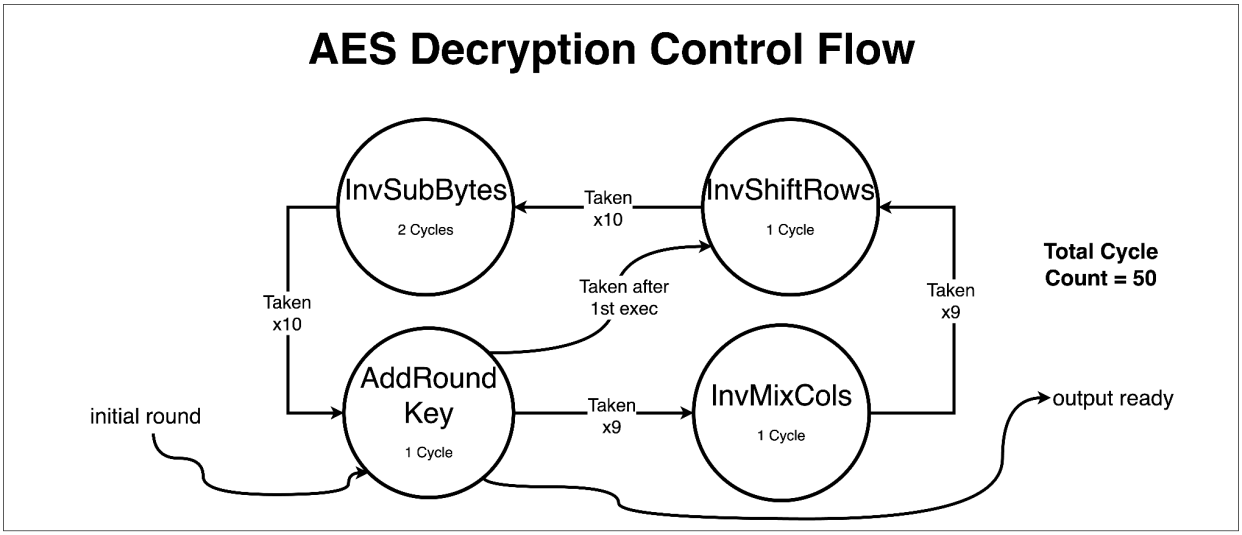


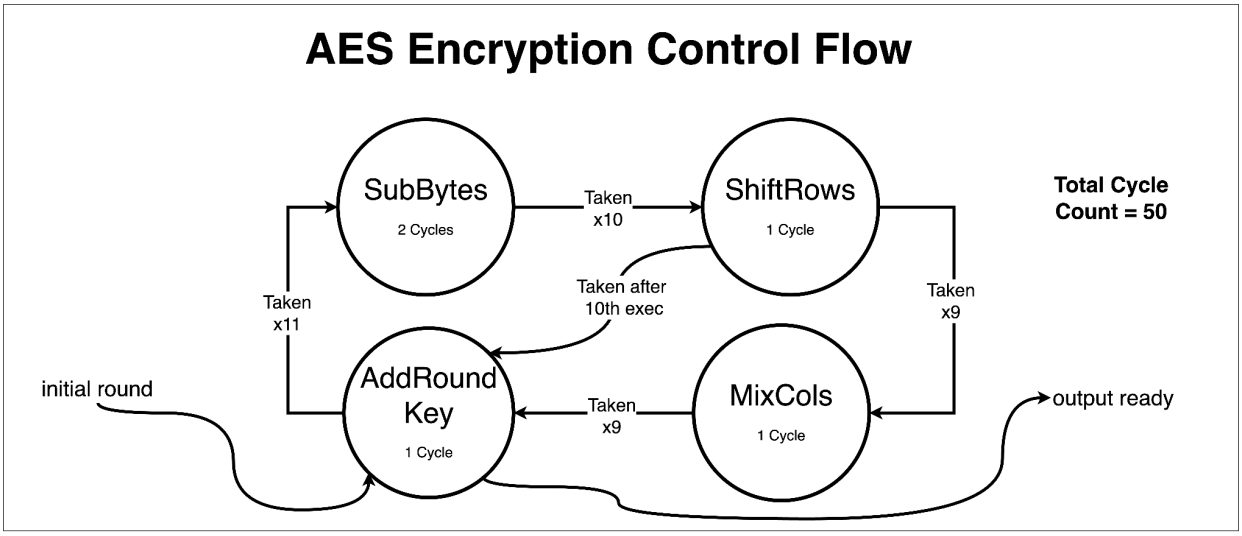
The main operation of AES_Scheduler is to maintain the correct ordering of blocks processed by the AES_Core(s). The byte-wide *q* input for this module will be a read (using *rdreq*) in from a FIFO IP and the scheduler will fill the *input* registers for each core in an ordered fashion. Once all core's *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.

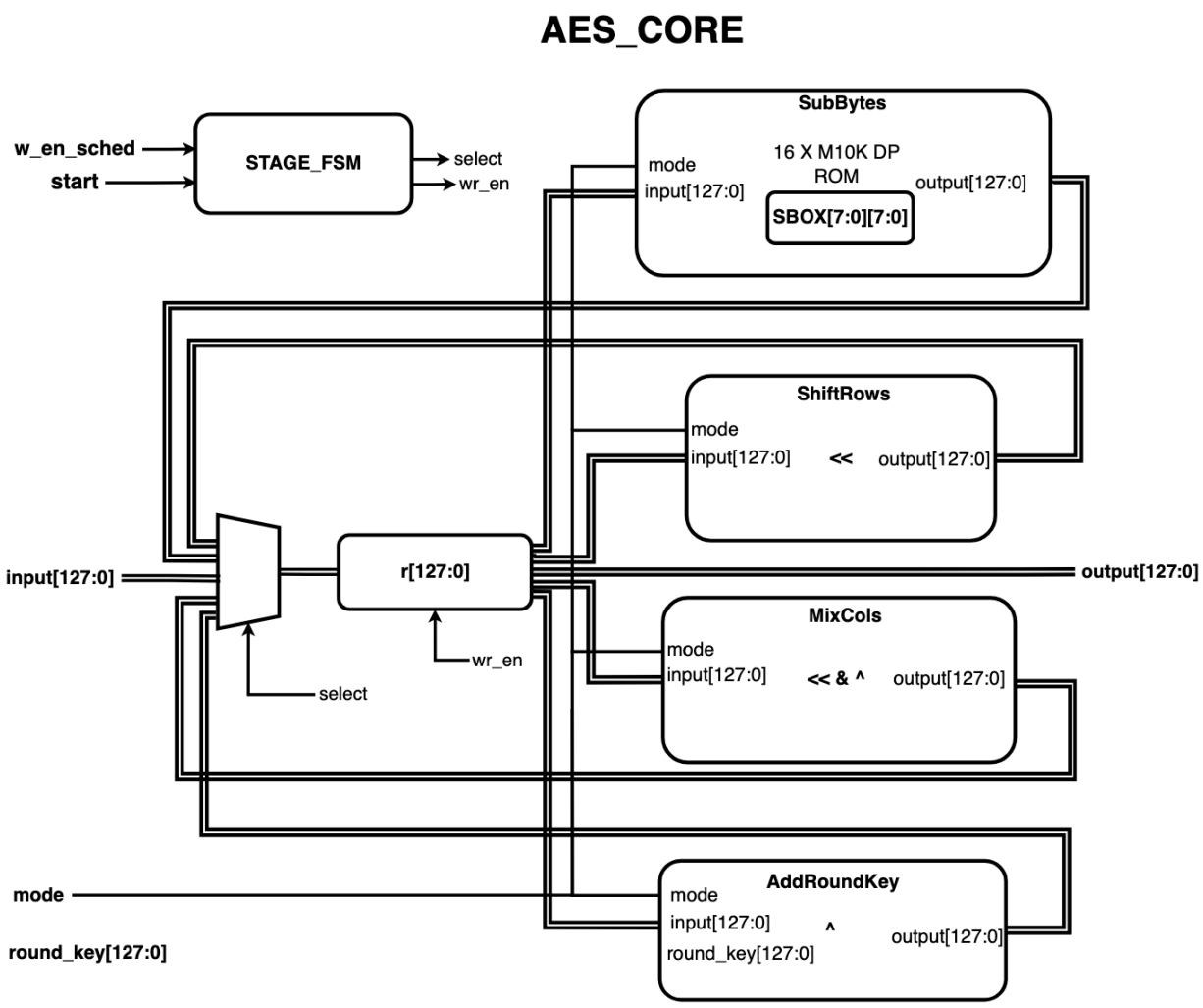
Another important function of this module is to provide the cores with the correct *round* they should be on as well as the correct *round_key* they should be using. Since we have designed the

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

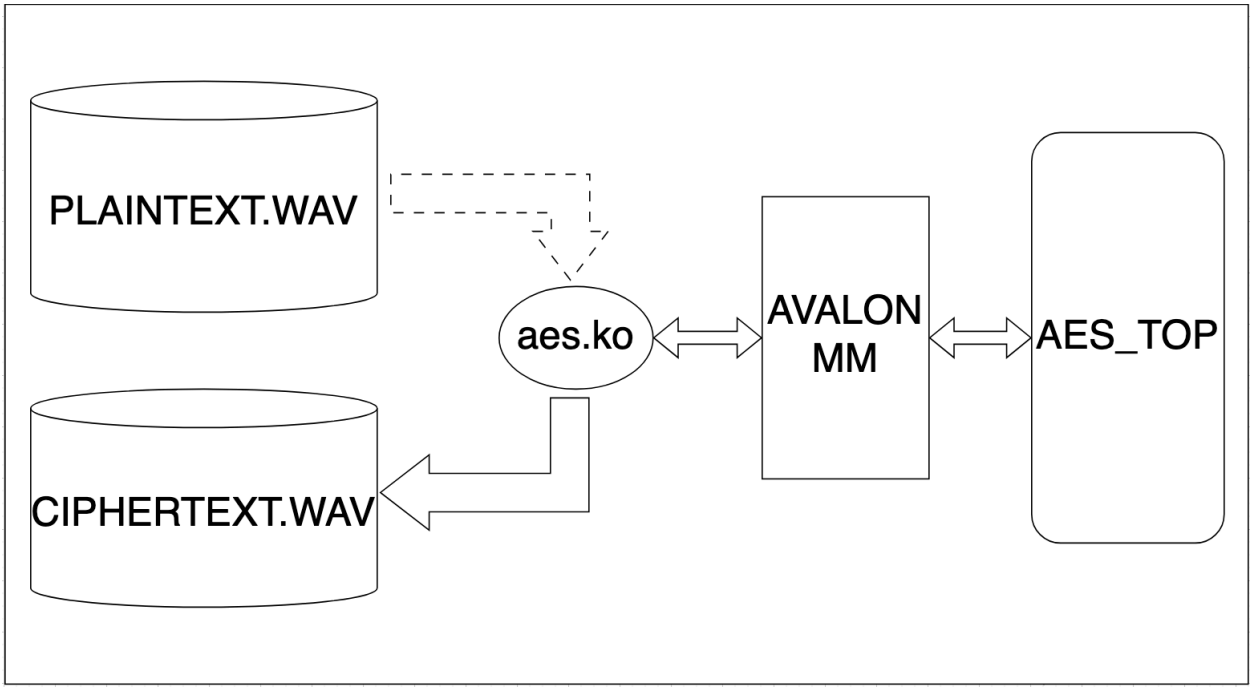
input-full- High when the input FIFO is full, if written to while full data will NOT be overwritten nor will the written value propagate forward

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



6) Task Distribution, Lessons, and Advice

Josh and Tyler worked on the hardware design diagrams, SystemVerilog module implementations, and HW/SW interfacing. Jaewon worked on C implementations of kernel driver modules, Verilator tests for all SV modules, round key expansion, and user space program for input processing. When it came to hardware, we learned that the more time and specificity you put into the design and planning stage of the project, the easier the actual implementation will be. Being very detailed with timing diagrams, state machines, and interfaces is what will help out tremendously in this course. Endianness was a huge culprit to the countless hours spent debugging this project. We would advise to spend time understanding data direction flow rigorously, and also learned that things that might seem insignificant can actually turn out to be bigger issues than expected.

7) Code

HW Code:

add_round_key.sv

```
module add_round_key (
   input [127:0] in_text,  // input text
   input [127:0] round_key, // round key
   output [127:0] out_text // output text
);

   // The same XOR operation, regardless of encryption or decryption
   assign out_text = in_text ^ round_key;

endmodule
```

aes_core.sv

```
module aes_core (
   input clk,
   input [127:0] in_text,
   input [127:0] round_key,
   input  wr_en_sched,
   input start,
   input mode,
   output [127:0] out_text
);

reg wr_en;
reg [2:0] select;
reg [2:0] state;
reg [2:0] next_state;
reg [127:0] r;
reg [5:0] counter;

add_round_key m_add_round_key(.in_text(r), .round_key(round_key), .out_text(out_add));
mix_columns m_mix_columns(.in(r), .mode(mode), .out(out_mix));
shift_rows m_shift_rows(.in_text(r), .mode(mode), .out_text(out_shift));
```

```
substitute_bytes m_substitute_bytes(.in(r), .clk(clk), .mode(mode), .out(out_sub));

logic [127:0] out_add;
logic [127:0] out_sub;
logic [127:0] out_shift;
logic [127:0] out_mix;


typedef enum logic [2:0] {
   IDLE   = 3'd0,
   ADD    = 3'd1,
   SUB_1  = 3'd2,
   SUB_2  = 3'd3,
   SHIFT  = 3'd4,
   MIX    = 3'd5
} state_t;

assign out_text = r;

/* FSM that controls what "stage" of AES we are in */
always_comb begin

   if (~mode) begin
       case (state)
           // TODO: Comments
           IDLE: begin next_state = ~start ? IDLE : ADD; select = 0; wr_en =
wr_en_sched; end
           ADD: begin next_state = (counter == 50) ? IDLE: SUB_1; select = 1; wr_en =
1; end
           SUB_1: begin next_state = SUB_2; select = 2; wr_en = 1; end
           SUB_2: begin next_state = SHIFT; select = 2; wr_en = 1; end
           SHIFT: begin next_state = (counter == 49) ? ADD : MIX; select = 3; wr_en =
1; end
           MIX: begin next_state = ADD; select = 4; wr_en = 1; end
           default: begin next_state = ~start ? IDLE : ADD; select = 0; wr_en =
wr_en_sched; end
       endcase
   end else begin
       case (state)
           // TODO: Comments
           IDLE: begin next_state = ~start ? IDLE : ADD; select = 0; wr_en =
wr_en_sched; end
```

```
          ADD: begin next_state = (counter == 1) ? SHIFT :  (counter == 50) ? IDLE :
MIX; select = 1; wr_en = 1;  end
          SUB_1: begin next_state = SUB_2; select = 2; wr_en = 1; end
          SUB_2: begin next_state = ADD; select = 2; wr_en = 1; end
          SHIFT: begin next_state = SUB_1; select = 3; wr_en = 1; end
          MIX: begin next_state = SHIFT; select = 4; wr_en = 1; end
          default: begin next_state = ~start ? 0 : 1; select = 0; wr_en =
wr_en_sched; end
      endcase
   end

end

always_ff @(posedge clk) begin

   /* Update counter for 50 cycles */
   if (~start)
       counter <= 0;
   else if (counter != 50)
       counter <= counter + 1;

   if (wr_en) begin
       case (select)
           0: r <= in_text;
           1: r <= out_add;
           2: r <= out_sub;
           3: r <= out_shift;
           4: r <= out_mix;
           default: r <= 128'b0;
       endcase
   end


   state <= next_state;

   // Debug print for state machine and counter
   // $display("AES_DEBUG: State=%d, NextState=%d, Counter=%d", state, next_state,
counter);

   // // Debug print for register in hex format with bytes clearly visible
   // $display("AES_DEBUG: Register (hex) = %h", r);
```

```
   // // Display register as 4x4 matrix of bytes in column-major order (AES state
format)
   // $display("AES_DEBUG: State matrix (column-major):");
   // $display("  %h %h %h %h", r[127:120], r[95:88], r[63:56], r[31:24]);   // First
column
   // $display("  %h %h %h %h", r[119:112], r[87:80], r[55:48], r[23:16]);   // Second
column
   // $display("  %h %h %h %h", r[111:104], r[79:72], r[47:40], r[15:8]);    // Third
column
   // $display("  %h %h %h %h", r[103:96], r[71:64], r[39:32], r[7:0]);      // Fourth
column

   // $display("AES_DEBUG: JOSH SUCKs:");
   // $display("  %h %h %h %h", out_mix[127:120], out_mix[95:88], out_mix[63:56],
out_mix[31:24]);   // First column
   // $display("  %h %h %h %h", out_mix[119:112], out_mix[87:80], out_mix[55:48],
out_mix[23:16]);   // Second column
   // $display("  %h %h %h %h", out_mix[111:104], out_mix[79:72], out_mix[47:40],
out_mix[15:8]);    // Third column
   // $display("  %h %h %h %h", out_mix[103:96], out_mix[71:64], out_mix[39:32],
out_mix[7:0]);      // Fourth column

end

endmodule
```

aes_ctl.sv

```
module aes_ctl(
   input logic clk,
   input logic reset,
   input logic write,
   input logic chipselect,
   input logic [31:0] writedata,
   output logic [31:0] readdata,
   output logic start,
   output logic mode
);
```

```
always_ff @(posedge clk) begin

   if (reset) begin
       start <= 1'b0;
       mode <= 1'b0;
   end

   if (write & chipselect) begin
       start <= writedata[0];
       mode <= writedata[1];
   end

   readdata <= {{30{1'b0}}, mode, start};
end



endmodule
```

aes_scheduler.sv

```
module aes_scheduler #(
   parameter int N_CORES = 10
)(
   input clk,
   input reset,
   input start,
   input mode,
   input logic [127:0] round_key, // Packed array syntax
   output logic [3:0] round_key_addr,

   input sink_valid,
   input logic [31:0] sink_data,
   // input sink_channel,
   // input sink_error,
   output logic sink_ready,

   output logic source_valid,
   output logic [31:0] source_data,
   // output source_channel,
   // output source_error,
```

```
   input source_ready
);

logic [127:0] in_text [10-1:0]; // TODO: Parameter N
logic [127:0] out_text [10-1:0]; // TODO: Parameter N

/* Read specific instantions */
logic [7:0] r_cnt; // sequential
logic we_r; // comb
logic we_core; //comb
logic wr_en_sched [10-1:0]; // TODO: Change 9 to parameter N (N number cores created)
//comb

logic [31:0] r_reg [3:0];

/* Execute specific instantiatons */
logic [5:0] c_cnt; // seq
logic [3:0] round; // seq
logic start_cores; // comb

/* Write specific instantiations */
logic [7:0] w_cnt; // seq
logic we_w; // comb
logic [31:0] core_out [3:0]; //comb
// logic we_w;
// logic [7:0] w_reg [15:0];

logic [1:0] state;
logic [1:0] next_state;
typedef enum logic [1:0] {
   IDLE    = 2'd0,
   READ    = 2'd1,
   EXEC    = 2'd2,
   WRITE   = 2'd3
} state_t;

logic [1:0] r_reg_word_sel;
logic [3:0] r_reg_core_sel;
logic [1:0] w_reg_word_sel;
logic [3:0] w_reg_core_sel;

/* Always have labels for sections of the r/w counters */
```

```
assign r_reg_word_sel = r_cnt[1:0];
assign r_reg_core_sel = r_cnt[5:2];
assign w_reg_word_sel = w_cnt[1:0];
assign w_reg_core_sel = w_cnt[5:2];


genvar j;
generate
   for (j = 0; j < 10; j++) begin : gen_aes_cores
       aes_core core_inst (
           .clk(clk),
           .in_text(in_text[j]),
           .round_key(round_key),
           .wr_en_sched(wr_en_sched[j]),
           .start(start_cores),
           .mode(mode),
           .out_text(out_text[j])
       );
   end
endgenerate

always_comb begin : state_handler

   case (state)
       IDLE: next_state = start ? READ : IDLE;

       READ: next_state = wr_en_sched[10-1] ? EXEC : READ;

       EXEC: next_state = (c_cnt == 50) ? WRITE : EXEC;

       WRITE: next_state = (w_cnt == 39) ? IDLE : WRITE;

       default: begin
           next_state = IDLE;
       end
   endcase

   // "Overwrites" the above if true
   if (reset | ~start)
       next_state = IDLE;

end
```

```
always_comb begin : core_input_comb

   // if (state == IDLE) begin
   //     /* Set default low stuff*/
   // end

   if (state == READ) begin
       we_core = (r_reg_word_sel == 0) & (r_cnt != 0);
       sink_ready = start;
       we_r = sink_valid & sink_ready;
       /* Set all cores to recieve r */
       for(int i = 0; i<10; i++) begin
           in_text[i] = {r_reg[0], r_reg[1], r_reg[2], r_reg[3]};
           wr_en_sched[i] = we_core & (r_reg_core_sel - 1 == 4'(i));
       end
   end else begin
       we_core = 0;
       sink_ready = 0;
       we_r = 0;
       for(int i = 0; i<10; i++) begin
           in_text[i] = 0;
           wr_en_sched[i] = 0;
       end
   end

   if (state == EXEC) begin
       start_cores = 1;
   end else begin
       start_cores = 0;
   end

   if (state == WRITE) begin
       source_valid = start;
       we_w = source_valid & source_ready;

       core_out[0] = out_text[w_reg_core_sel][127:96];
       core_out[1] = out_text[w_reg_core_sel][95:64];
       core_out[2] = out_text[w_reg_core_sel][63:32];
       core_out[3] = out_text[w_reg_core_sel][31:0];

       source_data = core_out[w_reg_word_sel];
```

```
    end else begin
        source_valid = 0;
        we_w = 0;
        for (int i = 0; i < 4; i++) begin
            core_out[i] = 32'd0;
        end
        source_data = 0;
    end
end

always_ff @(posedge clk ) begin : core_input_seq

    if (state == READ) begin
        if (we_r) begin
            r_reg[r_reg_word_sel] <= sink_data;
            r_cnt <= r_cnt + 1;
        end
    end else begin
        r_cnt <= 0;
    end

    if (state == EXEC) begin
        logic [3:0] next_round;
        next_round = round + 1;
        c_cnt <= c_cnt + 1;
        if (c_cnt  % 5 == 1) begin
            round <= next_round;
        end else if (c_cnt  % 5 == 0) begin
            round_key_addr <= mode ? 4'd11 - next_round : next_round;
        end
    end else begin
        c_cnt <= 0;
        round <= 0;
        round_key_addr <= 0;
    end

    if (state == WRITE) begin
        if (we_w)
            w_cnt <= w_cnt + 1;
    end else begin
        w_cnt <= 0;
    end
```

```
   state <= next_state;

end

endmodule
```

mix_columns.sv

```
module mix_columns(input logic  [127:0] in,
       input logic mode,
       output logic [127:0] out);

   logic [7:0] in_bytes [15:0];
   logic [7:0] out_bytes [15:0];


   function automatic [7:0] xtime2;
   input [7:0] a;
   begin
       xtime2 = (a << 1) ^ ((a[7]) ? 8'h1B : 8'h00);
   end
   endfunction

   function automatic [7:0] xtime3;
   input [7:0] a;
   begin
       xtime3 = xtime2(a) ^ a;
   end
   endfunction

   function automatic [7:0] xtime9;
   input [7:0] a;
   logic [7:0] a2, a4, a8;
   begin
       a2 = xtime2(a);
       a4 = xtime2(a2);
       a8 = xtime2(a4);
       xtime9 = a8 ^ a;
```

```
end
endfunction

function automatic [7:0] xtime11;
input [7:0] a;
logic [7:0] a2, a4, a8;
begin
    a2 = xtime2(a);          // a * 2
    a4 = xtime2(a2);         // a * 4
    a8 = xtime2(a4);         // a * 8
    xtime11 = a8 ^ a2 ^ a; // a * 8 ^ a * 2 ^ a
end
endfunction

function automatic [7:0] xtime13;
input [7:0] a;
logic [7:0] a2, a4, a8;
begin
    a2 = xtime2(a);          // a * 2
    a4 = xtime2(a2);         // a * 4
    a8 = xtime2(a4);         // a * 8
    xtime13 = a8 ^ a4 ^ a; // a * 8 ^ a * 2 ^ a
end
endfunction

function automatic [7:0] xtime14;
input [7:0] a;
logic [7:0] a2, a4, a8;
begin
    a2 = xtime2(a);          // a * 2
    a4 = xtime2(a2);         // a * 4
    a8 = xtime2(a4);         // a * 8
    xtime14 = a8 ^ a4 ^ a2; // a * 8 ^ a * 2 ^ a
end
endfunction

function automatic logic [7:0] get_byte(input logic [127:0] vec, input int index);
    get_byte = vec[127 - index * 8 -: 8];
endfunction


always_comb begin
```

```
      for (int i = 0; i < 16; i++) begin
          in_bytes[i] = in[127 - i * 8 -: 8];
      end

      if (~mode) begin
          for (int i = 0; i < 16; i+=4) begin

              out_bytes[i] =  (xtime2(in_bytes[i])) ^ (xtime3(in_bytes[i+1])) ^
in_bytes[i+2] ^ in_bytes[i+3];
              out_bytes[i+1] = in_bytes[i] ^ (xtime2(in_bytes[i+1])) ^
(xtime3(in_bytes[i+2])) ^ in_bytes[i+3];
              out_bytes[i+2] = in_bytes[i] ^ in_bytes[i+1] ^ (xtime2(in_bytes[i+2]))
^ xtime3(in_bytes[i+3]);
              out_bytes[i+3] = xtime3(in_bytes[i]) ^ in_bytes[i+1] ^ in_bytes[i+2] ^
xtime2(in_bytes[i+3]);

          end

      end else begin

          for (int i = 0; i < 16; i+=4) begin

              out_bytes[i]    = (xtime14(in_bytes[i])) ^ (xtime11(in_bytes[i+1])) ^
(xtime13(in_bytes[i+2])) ^ (xtime9(in_bytes[i+3]));
              out_bytes[i+1]  = (xtime9(in_bytes[i])) ^ (xtime14(in_bytes[i+1])) ^
(xtime11(in_bytes[i+2])) ^ (xtime13(in_bytes[i+3]));
              out_bytes[i+2]  = (xtime13(in_bytes[i])) ^ (xtime9(in_bytes[i+1])) ^
(xtime14(in_bytes[i+2])) ^ (xtime11(in_bytes[i+3]));
              out_bytes[i+3] = (xtime11(in_bytes[i])) ^ (xtime13(in_bytes[i+1])) ^
(xtime9(in_bytes[i+2])) ^ (xtime14(in_bytes[i+3]));

          end

      end

      for (int i = 0; i < 16; i++) begin
          out[127 - i * 8 -: 8] = out_bytes[i];
      end
   end

endmodule
```

mixed_width_true_dual_port_ram.sv

```
// Quartus Prime SystemVerilog Template
//
// True Dual-Port RAM with single clock and different data width on the two ports
//
// The first datawidth and the widths of the addresses are specified
// The second data width is equal to DATA_WIDTH1 * RATIO, where RATIO = (1 <<
(ADDRESS_WIDTH1 - ADDRESS_WIDTH2)
// RATIO must have value that is supported by the memory blocks in your target
// device.  Otherwise, no RAM will be inferred.
//
// Read-during-write behavior returns old data for all combinations of read and
// write on both ports
//
// This style of RAM cannot be used on certain devices, e.g. Stratix V; in that case
use the template for Dual-Port RAM with new data on read-during write on the same port

module mixed_width_true_dual_port_ram
   #(parameter int
       DATA_WIDTH1 = 8,
       ADDRESS_WIDTH1 = 10,
       ADDRESS_WIDTH2 = 8)
(
       input [ADDRESS_WIDTH1-1:0] addr1,
       input [ADDRESS_WIDTH2-1:0] addr2,
       input [DATA_WIDTH1      -1:0] data_in1,
       input [DATA_WIDTH1*(1<<(ADDRESS_WIDTH1 - ADDRESS_WIDTH2))-1:0] data_in2,
       input we1, we2, clk, rst,
       output reg [DATA_WIDTH1-1      :0] data_out1,
       output reg [DATA_WIDTH1*(1<<(ADDRESS_WIDTH1 - ADDRESS_WIDTH2))-1:0] data_out2);

   localparam RATIO = 1 << (ADDRESS_WIDTH1 - ADDRESS_WIDTH2); // valid values are
2,4,8... family dependent
   localparam DATA_WIDTH2 = DATA_WIDTH1 * RATIO;
   localparam RAM_DEPTH = 1 << ADDRESS_WIDTH2;

   // Use a multi-dimensional packed array to model the different read/ram width
   reg [RATIO-1:0] [DATA_WIDTH1-1:0] ram[0:RAM_DEPTH-1];

   reg [DATA_WIDTH1-1:0] data_reg1;
```

```
    reg [DATA_WIDTH2-1:0] data_reg2;

    // Port A
    always@(posedge clk)
    begin
        if(we1)
            ram[addr1 / RATIO][addr1 % RATIO] <= data_in1;
        data_reg1 <= ram[addr1 / RATIO][addr1 % RATIO];
    end
    assign data_out1 = data_reg1;

    // port B
    always@(posedge clk)
    begin
        if(we2)
            ram[addr2] <= data_in2;
        data_reg2 <= ram[addr2];
    end

    assign data_out2 = data_reg2;
endmodule : mixed_width_true_dual_port_ram
```

shift_rows.sv

```
module shift_rows (
    input [127:0] in_text,  // input text
    input mode,             // mode (0 for encryption, 1 for decryption)
    output [127:0] out_text // output text
);
    // Define byte segments for each row
    logic [31:0] row0, row1, row2, row3;

    // Extract rows from input (each row contains 4 bytes, COLUMN MAJOR!!)
    assign row0 = {in_text[127:120], in_text[95:88], in_text[63:56], in_text[31:24]};
    assign row1 = {in_text[119:112], in_text[87:80], in_text[55:48], in_text[23:16]};
    assign row2 = {in_text[111:104], in_text[79:72], in_text[47:40], in_text[15:8]};
    assign row3 = {in_text[103:96], in_text[71:64], in_text[39:32], in_text[7:0]};

    // Shifted rows
    logic [31:0] shifted_row0, shifted_row1, shifted_row2, shifted_row3;
```

```
    // Row 0: No shift in either direction
    assign shifted_row0 = row0;

    // imagine [byte0]  [byte1] [byte2] [byte3]
    //         [31:24]  [23:16] [15:8]  [7:0]

    // Row 1: Shift by 1 right (decryption) or left (encryption)
    assign shifted_row1 = mode ? {row1[7:0], row1[31:8]} : {row1[23:0], row1[31:24]};

    // Row 2: Shift left/right by 2 in either direction (same result)
    assign shifted_row2 = {row2[15:0], row2[31:16]};

    // Row 3: Shift left by 3 right (decryption) or left (decryption)
    assign shifted_row3 = mode ? {row3[23:0], row3[31:24]} : {row3[7:0], row3[31:8]};

    // Convert back to 128 bit output in column-major order
    assign out_text = {
        shifted_row0[31:24], shifted_row1[31:24], shifted_row2[31:24],
shifted_row3[31:24],
        shifted_row0[23:16], shifted_row1[23:16], shifted_row2[23:16],
shifted_row3[23:16],
        shifted_row0[15:8],  shifted_row1[15:8],  shifted_row2[15:8],
shifted_row3[15:8],
        shifted_row0[7:0],   shifted_row1[7:0],   shifted_row2[7:0],
shifted_row3[7:0]
    };

endmodule
```

simple_rom.sv

```
module simple_rom (
    input  logic        clk,
    input  logic [8:0]  addr,
    output logic [7:0]  data
);

    logic [7:0] rom_data [0:511];
```

```
    initial begin
        $readmemh("sub_box.hex", rom_data);
    end

    always_ff @(posedge clk) begin
        data <= rom_data[addr];
    end

endmodule
```

sub_box.hex

```
:10000000637C777BF26B6FC5300167FEFDAB776C
:10001000CA82C97DFA5947F0ADD4A2AF9CA472C025
:10002000B7FD93263F3FF7CC34A5E5F171D831150B
:1000300004C723C31896059A071280E2EB27B275F9
:10004000098332C1A1B6E5AA0523BD6B3929E32F8486
:10005000D5310000ED20FCB15B6ACB39BE4A4C58CF3D
:10006000D0EFAAFB434D338545F9027F503C9FA891
:10007000A51A3408F929D38F5BCBD21A10FFF3D208
:10008000CD0C13EC5F974417C4A77E3D645D1973F5
:10009000608A14FDC222A90886E46EEB814DE5E0BDB7A
:1000A000E0323A0A490624C5C2D3AC629195E47904
:1000B000E7C8376D8DD54EA96C56F4EA657AAE0806
:1000C000BA78252E1CA6B4C6E8DD741F4BBD8B8A25
:1000D000703EB5664803F60E6135576B986C11D9E8F
:1000E000E1F8981169D98E949B1E87E9CE5528DF3A
:1000F0008CA1890DBFE642684199D2D0FB054BB1617
:10010000520906AD5303665A38BF40A39E81F3D7FB72
:100110007CE339829B2FFF878E34384C4DEECBE9CB54
:10012000547B943A2A6C223D3EE4C950B42FA4C3E4F9
:100130000862EA166282D924B2765BA2496D8BD125D2
:1001400072F8F6648668981D64A45CCC5D65B6929B
:1001500060C7048500FDEDB9DA5E1546576A78D9D8475
:1001600090D8AB008CBC0DA0F7E458056B8B34506F8
:10017000D02C1E8FCA3F0F02C1AFBD030138A6BB9
:100180003A9111414F67DCEA97F2CFCEF0B4E673E5
:1001900096AC7422E7AD3585E2F937E81C75DF6E72
:1001A00047F11A711D29C5896FB7620EAA18BE1B59
:1001B000FC563E4BC6D2790A9DB0CFE78CD5AF43A
:1001C0001FDD8A33880F7C31B1121059278ECEF5F9C
```

```
:1001D00060517FA9196B54A0D2DE57A9F93C99CEF27
:1001E000A0E03B4DAE2AF5B0C8EBBB3C8353996145
:1001F000172B047EBA77D626E1691463552l0C7D41
:00000001FF
```

sub_box_intel.hex

```
63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

substitute_bytes.sv

```
module substitute_bytes(input logic  [127:0] in,
          input logic mode,
          input logic clk,
          output logic [127:0] out);

   logic [7:0] in_bytes [15:0];
   logic [7:0] out_bytes [15:0];
   logic [8:0] addr [15:0];

   // function automatic logic [7:0] get_byte(input logic [127:0] vec, input int
index);
   //  get_byte = vec[127 - index * 8 -: 8];
   // endfunction

   genvar j;
   generate
       for (j = 0; j < 16; j++) begin: gen_rom_instances
           simple_rom simple_rom_j(.clk(clk), .data(out_bytes[j]), .addr(addr[j]));
       end
   endgenerate

   always_comb begin

       for (int i = 0; i < 16; i++) begin
           in_bytes[i] = in[127 - i * 8 -: 8];
           out[127 - i * 8 -: 8] = out_bytes[i];
           addr[i] = {mode, in_bytes[i]};
       end


   end

endmodule
```

Fifo SW Code:

fifo_check.c

```
/*
* Utility to check the fill level of the FIFO devices
*
* Usage: ./fifo_check [fifo_num] [config_mode]
*   where fifo_num is 0 for input FIFO, 1 for output FIFO
*   config_mode (optional) is 1-4 for different endianness and word ordering:
*     1 = Big endian, normal word order
*     2 = Big endian, reverse word order
*     3 = Little endian, normal word order
*     4 = Little endian, reverse word order
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "fifo_write.h"
#include "fifo_read.h"

int main(int argc, char *argv[]) {
   int fd;
   int fifo_num;
   int level;
   int config_mode = 0; /* 0 means don't change the current setting */

   /* Check command line arguments */
   if (argc < 2 || argc > 3) {
       fprintf(stderr, "Usage: %s [0|1] [config_mode]\n", argv[0]);
       fprintf(stderr, "  0: Check input FIFO level\n");
       fprintf(stderr, "  1: Check output FIFO level\n");
       fprintf(stderr, "  config_mode (optional): Set endianness and word order
(1-4):\n");
       fprintf(stderr, "    1 = Big endian, normal word order\n");
       fprintf(stderr, "    2 = Big endian, reverse word order\n");
       fprintf(stderr, "    3 = Little endian, normal word order\n");
       fprintf(stderr, "    4 = Little endian, reverse word order\n");
       return EXIT_FAILURE;
   }
```

```c
/* Parse FIFO number */
fifo_num = atoi(argv[1]);
if (fifo_num != 0 && fifo_num != 1) {
    fprintf(stderr, "Error: FIFO number must be 0 or 1\n");
    return EXIT_FAILURE;
}

/* Parse config mode if provided */
if (argc == 3) {
    config_mode = atoi(argv[2]);
    if (config_mode < 1 || config_mode > 4) {
        fprintf(stderr, "Error: config_mode must be between 1 and 4\n");
        return EXIT_FAILURE;
    }
}

/* Open the appropriate device file */
if (fifo_num == 0) {
    /* Input FIFO (fifo_0) */
    fd = open("/dev/fifo_write", O_RDWR);
    if (fd < 0) {
        perror("Error opening /dev/fifo_write");
        return EXIT_FAILURE;
    }

    /* Set config mode if specified */
    if (config_mode > 0) {
        if (ioctl(fd, FIFO_WRITE_SET_CONFIG, &config_mode) < 0) {
            perror("Error setting configuration mode");
            close(fd);
            return EXIT_FAILURE;
        }
        printf("Set FIFO 0 config mode to %d\n", config_mode);
    }

    /* Get the fill level */
    fifo_write_arg_t fwa;
    if (ioctl(fd, FIFO_WRITE_GET_LEVEL, &fwa) < 0) {
        perror("Error getting FIFO level");
        close(fd);
        return EXIT_FAILURE;
```

```
        }
        level = fwa.data;
    } else {
        /* Output FIFO (fifo_1) */
        fd = open("/dev/fifo_read", O_RDWR);
        if (fd < 0) {
            perror("Error opening /dev/fifo_read");
            return EXIT_FAILURE;
        }

        /* Set config mode if specified */
        if (config_mode > 0) {
            if (ioctl(fd, FIFO_READ_SET_CONFIG, &config_mode) < 0) {
                perror("Error setting configuration mode");
                close(fd);
                return EXIT_FAILURE;
            }
            printf("Set FIFO 1 config mode to %d\n", config_mode);
        }

        /* Get the fill level */
        fifo_read_arg_t fra;
        if (ioctl(fd, FIFO_READ_GET_LEVEL, &fra) < 0) {
            perror("Error getting FIFO level");
            close(fd);
            return EXIT_FAILURE;
        }
        level = fra.data;
    }

    /* Close the device */
    close(fd);

    /* Print the result */
    printf("FIFO %d fill level: %d\n", fifo_num, level);

    return EXIT_SUCCESS;
}
```

fifo_cleanup.sh

```
#!/bin/bash
# FIFO system cleanup script
```

```
echo "=== FIFO System Cleanup ==="
echo

# Check if running as root
if [ "$(id -u)" -ne 0 ]; then
   echo "Error: This script must be run as root (use sudo)" >&2
   exit 1
fi

# Check which modules are loaded
echo "Currently loaded FIFO modules:"
lsmod | grep fifo

# Unload modules
echo "Unloading kernel modules..."
if lsmod | grep -q "fifo_read"; then
   echo "Removing fifo_read module..."
   rmmod fifo_read
   if [ $? -ne 0 ]; then
       echo "Warning: Could not remove fifo_read module" >&2
   fi
else
   echo "fifo_read module is not loaded"
fi

if lsmod | grep -q "fifo_write"; then
   echo "Removing fifo_write module..."
   rmmod fifo_write
   if [ $? -ne 0 ]; then
       echo "Warning: Could not remove fifo_write module" >&2
   fi
else
   echo "fifo_write module is not loaded"
fi

# Verify modules are unloaded
if lsmod | grep -q "fifo"; then
   echo "Warning: Some FIFO modules may still be loaded:" >&2
   lsmod | grep fifo
else
   echo "All FIFO modules have been successfully unloaded"
```

```
fi

# Optionally clean up compiled files
read -p "Do you want to clean up compiled files? (y/n): " clean_files
if [ "$clean_files" = "y" ] || [ "$clean_files" = "Y" ]; then
   echo "Cleaning up compiled files..."
   make clean
   echo "Compiled files have been removed"
fi

echo
echo "=== FIFO System Cleanup Complete ==="
```

fifo_read.c

```
/*
* Device driver for reading from the output FIFO
*
* A Platform device implemented using the misc subsystem
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>  /* For fixed-width integer types in kernel space */
#include <linux/moduleparam.h>
#include "fifo_read.h"

#define DRIVER_NAME "fifo_read"

/* Module parameter for endianness and word order configuration
* 1 = Big endian, normal word order
* 2 = Big endian, reverse word order
```

```
* 3 = Little endian, normal word order
* 4 = Little endian, reverse word order
*/
static int config_mode = 1; /* Default: Big endian, normal word order */
module_param(config_mode, int, 0644);
MODULE_PARM_DESC(config_mode, "Endianness and word order: 1=BE+normal, 2=BE+reverse,
3=LE+normal, 4=LE+reverse");

/* Device registers - FIFO is 8 bytes (64 bits) */
#define FIFO_DATA(x) (x)
#define FIFO_LEVEL_REG(x) ((x)+0x0)     /* Fill level register */
#define FIFO_STATUS_REG(x) ((x)+0x4)    /* Status register (i_status) */
#define FIFO_EVENT_REG(x) ((x)+0x8)     /* Event register */
#define FIFO_IENABLE_REG(x) ((x)+0xC)   /* Interrupt enable register */

/* Status register bit definitions */
#define FIFO_FULL_BIT    0
#define FIFO_EMPTY_BIT   1
#define FIFO_ALMOSTFULL_BIT 2
#define FIFO_ALMOSTEMPTY_BIT 3
#define FIFO_OVERFLOW_BIT 4
#define FIFO_UNDERFLOW_BIT 5

/*
* Information about our device
*/
struct fifo_read_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   void __iomem *csr_virtbase; /* Control and Status Register base */
   unsigned int last_read;    /* Last value read from FIFO */
} dev;

/*
* Read the FIFO level register and return current fill level
*/
static u32 read_fifo_level(void)
{
   return ioread32(FIFO_LEVEL_REG(dev.csr_virtbase));
}

/*
```

```c
* Read the status register and return its value
*/
static u32 read_fifo_status(void)
{
   return ioread32(FIFO_STATUS_REG(dev.csr_virtbase));
}

/*
* Check if the FIFO is empty
*/
static int is_fifo_empty(void)
{
   u32 status = read_fifo_status();
   return (status & (1 << FIFO_EMPTY_BIT)) ? 1 : 0;
}

/*
* Read data from the FIFO
* Returns 0 on success, negative error code on failure
* Writes the data to the provided pointer
*/
static int read_from_fifo(unsigned int *data_out)
{
   /* Check if the FIFO is empty before reading */
   if (is_fifo_empty()) {
       pr_info("FIFO is empty, cannot read data\n");
       return -EAGAIN; /* Resource temporarily unavailable */
   }

   /* Read 32-bit data from the FIFO */
   unsigned int raw_data = ioread32(FIFO_DATA(dev.virtbase));

   /* Process data according to config_mode */
   switch (config_mode) {
   case 1: /* Big endian, normal word order */
       *data_out = raw_data;
       break;
   case 2: /* Big endian, reverse word order */
       *data_out = ((raw_data & 0xFF000000) >> 24) |
               ((raw_data & 0x00FF0000) >> 8) |
               ((raw_data & 0x0000FF00) << 8) |
               ((raw_data & 0x000000FF) << 24);
```

```
        break;
    case 3: /* Little endian, normal word order */
        *data_out = ((raw_data & 0xFF000000) >> 24) |
                ((raw_data & 0x00FF0000) >> 8) |
                ((raw_data & 0x0000FF00) << 8) |
                ((raw_data & 0x000000FF) << 24);
        break;
    case 4: /* Little endian, reverse word order */
        *data_out = ((raw_data & 0x000000FF) << 24) |
                ((raw_data & 0x0000FF00) << 8) |
                ((raw_data & 0x00FF0000) >> 8) |
                ((raw_data & 0xFF000000) >> 24);
        break;
    default:
        *data_out = raw_data;
        break;
    }

    dev.last_read = *data_out;
    return 0; /* Success */
}

/*
* Handle ioctl() calls from userspace
*/
static long fifo_read_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    fifo_read_arg_t fra;
    int ret;
    unsigned int fifo_data;

    switch (cmd) {
    case FIFO_READ_READ_DATA:
        ret = read_from_fifo(&fifo_data);
        if (ret < 0)
            return ret; /* Return error from read_from_fifo */

        fra.data = fifo_data;
        if (copy_to_user((fifo_read_arg_t *) arg, &fra,
                 sizeof(fifo_read_arg_t)))
            return -EACCES;
        break;
```

```
    case FIFO_READ_GET_LEVEL:
        fra.data = read_fifo_level();
        if (copy_to_user((fifo_read_arg_t *) arg, &fra,
                 sizeof(fifo_read_arg_t)))
            return -EACCES;
        break;

    case FIFO_READ_SET_CONFIG:
        if (get_user(config_mode, (int __user *)arg))
            return -EACCES;

        /* Validate config_mode */
        if (config_mode < 1 || config_mode > 4) {
            pr_err(DRIVER_NAME ": Invalid config_mode %d. Must be 1-4\n", config_mode);
            return -EINVAL;
        }

        pr_info(DRIVER_NAME ": Config mode set to %d\n", config_mode);
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fifo_read_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = fifo_read_ioctl,
};

/* Information about our device for the "misc" framework */
static struct miscdevice fifo_read_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &fifo_read_fops,
};

/*
```

```
* Initialization code: get resources and set up device
*/
static int __init fifo_read_probe(struct platform_device *pdev)
{
   int ret;
   const char *node_name;
   struct resource *res_data, *res_csr;

   /* Check if this is the correct FIFO device (fifo_1) */
   node_name = pdev->dev.of_node->name;
   pr_info(DRIVER_NAME ": Probing node name=%s\n", node_name ? node_name : "NULL");

   /* Check for exact fifo name first, fallback to checking if it contains "fifo" */
   if (node_name && ((strcmp(node_name, "fifo@0x100000140") == 0) ||
                     (strcmp(node_name, "fifo_1") == 0) ||
                     strstr(node_name, "fifo") != NULL)) {
       const __be32 *reg = of_get_property(pdev->dev.of_node, "reg", NULL);
       /* For fifo_1, the first register region has offset 0x40 at index 1 in the reg
array */
       if (reg && (be32_to_cpu(reg[1]) == 0x140)) {
           /* This is fifo_1, which is the one we want for reading */
           pr_info(DRIVER_NAME ": Found output FIFO at offset 0x%x\n",
be32_to_cpu(reg[1]));
       } else {
           /* This is not fifo_1 */
           pr_info(DRIVER_NAME ": Not the output FIFO\n");
           return -ENODEV;
       }
   } else {
       /* Not a FIFO node */
       pr_info(DRIVER_NAME ": Not a FIFO node\n");
       return -ENODEV;
   }

   /* Register ourselves as a misc device: creates /dev/fifo_read */
   ret = misc_register(&fifo_read_misc_device);
   if (ret) {
       pr_err(DRIVER_NAME ": Couldn't register misc device\n");
       return ret;
   }

   /* Get the data port and CSR addresses */
```

```
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out_csr");

    if (!res_data || !res_csr) {
        pr_err(DRIVER_NAME ": Could not get FIFO resources\n");
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(res_data->start, resource_size(res_data), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    if (request_mem_region(res_csr->start, resource_size(res_csr), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_release_data_mem_region;
    }

    /* Arrange access to data registers */
    dev.virtbase = ioremap(res_data->start, resource_size(res_data));
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_csr_mem_region;
    }

    /* Arrange access to CSR registers */
    dev.csr_virtbase = ioremap(res_csr->start, resource_size(res_csr));
    if (dev.csr_virtbase == NULL) {
        ret = -ENOMEM;
        goto out_unmap_data;
    }

    return 0;

out_unmap_data:
    iounmap(dev.virtbase);
out_release_csr_mem_region:
    release_mem_region(res_csr->start, resource_size(res_csr));
```

```
out_release_data_mem_region:
   release_mem_region(res_data->start, resource_size(res_data));
out_deregister:
   misc_deregister(&fifo_read_misc_device);
   return ret;
}

/* Clean-up code: release resources */
static int fifo_read_remove(struct platform_device *pdev)
{
   struct resource *res_data, *res_csr;

   /* Get the resources again to release them properly */
   res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out");
   res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out_csr");

   /* Unmap the virtual memory */
   iounmap(dev.virtbase);
   iounmap(dev.csr_virtbase);

   /* Release the memory regions */
   if (res_data)
       release_mem_region(res_data->start, resource_size(res_data));
   if (res_csr)
       release_mem_region(res_csr->start, resource_size(res_csr));

   /* Deregister the misc device */
   misc_deregister(&fifo_read_misc_device);
   return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fifo_read_of_match[] = {
   { .compatible = "ALTR,fifo-21.1" },
   {},
};
MODULE_DEVICE_TABLE(of, fifo_read_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver fifo_read_driver = {
```

```
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(fifo_read_of_match),
    },
    .probe = fifo_read_probe,
    .remove = __exit_p(fifo_read_remove),
};

/* Called when the module is loaded: set things up */
static int __init fifo_read_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_register(&fifo_read_driver);
}

/* Called when the module is unloaded: release resources */
static void __exit fifo_read_exit(void)
{
    platform_driver_unregister(&fifo_read_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(fifo_read_init);
module_exit(fifo_read_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("FIFO read driver");
```

fifo_read.h

```
#ifndef _FIFO_READ_H
#define _FIFO_READ_H

#include <linux/ioctl.h>
#include <linux/types.h>

typedef struct {
    unsigned int data;
} fifo_read_arg_t;

#define FIFO_READ_MAGIC 'r'
```

```
#define FIFO_READ_READ_DATA _IOR(FIFO_READ_MAGIC, 0, fifo_read_arg_t)
#define FIFO_READ_GET_LEVEL _IOR(FIFO_READ_MAGIC, 1, fifo_read_arg_t *)
#define FIFO_READ_SET_CONFIG _IOW(FIFO_READ_MAGIC, 2, int)

#endif
```

fifo_start.sh

```
#!/bin/bash
# FIFO system startup script

echo "=== FIFO System Startup ==="
echo

# Check if running as root
if [ "$(id -u)" -ne 0 ]; then
   echo "Error: This script must be run as root (use sudo)" >&2
   exit 1
fi

# Make sure old modules are removed
echo "Cleaning up any previous modules..."
rmmod fifo_read 2>/dev/null
rmmod fifo_write 2>/dev/null

# Build everything
echo "Building kernel modules and userspace programs..."
make clean
make default

# Load the kernel modules
echo "Loading kernel modules..."
insmod fifo_write.ko
if [ $? -ne 0 ]; then
   echo "Error: Failed to load fifo_write.ko module" >&2
   exit 1
fi

insmod fifo_read.ko
if [ $? -ne 0 ]; then
   echo "Error: Failed to load fifo_read.ko module" >&2
   rmmod fifo_write
```

```
    exit 1
fi

# Check if modules are loaded
echo "Checking loaded modules:"
lsmod | grep fifo

# Display help information
echo
echo "=== FIFO System Ready ==="
echo "Sample commands:"
echo "  # Write data to FIFO:"
echo "  ./write_data input.txt"
echo
echo "  # Read data from FIFO:"
echo "  ./read_data output.txt 5"
echo
echo "  # Read continuously until Ctrl+C:"
echo "  ./read_data output.txt"
echo
echo "  # When finished, run cleanup script:"
echo "  ./fifo_cleanup.sh"
echo "================================="
```

fifo_write.c

```
/*
* Device driver for writing to the input FIFO
*
* A Platform device implemented using the misc subsystem
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
```

```c
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>  /* For fixed-width integer types in kernel space */
#include <linux/moduleparam.h>
#include "fifo_write.h"

#define DRIVER_NAME "fifo_write"

/* Module parameter for endianness and word order configuration
* 1 = Big endian, normal word order
* 2 = Big endian, reverse word order
* 3 = Little endian, normal word order
* 4 = Little endian, reverse word order
*/
static int config_mode = 1; /* Default: Big endian, normal word order */
module_param(config_mode, int, 0644);
MODULE_PARM_DESC(config_mode, "Endianness and word order: 1=BE+normal, 2=BE+reverse,
3=LE+normal, 4=LE+reverse");

/* Device registers - FIFO is 8 bytes (64 bits) */
#define FIFO_DATA(x) (x)
#define FIFO_LEVEL_REG(x) ((x)+0x0)     /* Fill level register */
#define FIFO_STATUS_REG(x) ((x)+0x4)    /* Status register (i_status) */
#define FIFO_EVENT_REG(x) ((x)+0x8)     /* Event register */
#define FIFO_IENABLE_REG(x) ((x)+0xC)   /* Interrupt enable register */

/* Status register bit definitions */
#define FIFO_FULL_BIT    0
#define FIFO_EMPTY_BIT   1
#define FIFO_ALMOSTFULL_BIT 2
#define FIFO_ALMOSTEMPTY_BIT 3
#define FIFO_OVERFLOW_BIT 4
#define FIFO_UNDERFLOW_BIT 5

/*
* Information about our device
*/
struct fifo_write_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   void __iomem *csr_virtbase; /* Control and Status Register base */
} dev;
```

```c
/*
* Read the FIFO level register and return current fill level
*/
static u32 read_fifo_level(void)
{
   return ioread32(FIFO_LEVEL_REG(dev.csr_virtbase));
}

/*
* Read the status register and return its value
*/
static u32 read_fifo_status(void)
{
   return ioread32(FIFO_STATUS_REG(dev.csr_virtbase));
}

/*
* Check if the FIFO is full
*/
static int is_fifo_full(void)
{
   u32 status = read_fifo_status();
   return (status & (1 << FIFO_FULL_BIT)) ? 1 : 0;
}

/*
* Write data to the FIFO
* Returns 0 on success, -EBUSY if FIFO is full
*/
static int write_to_fifo(unsigned int data)
{
   /* Check if the FIFO is full before writing */
   if (is_fifo_full()) {
       pr_info("FIFO is full, cannot write data\n");
       return -EBUSY; /* Device is busy (FIFO full) */
   }

   /* Apply endianness and word order conversions according to config_mode */
   unsigned int adjusted_data;

   switch (config_mode) {
```

```
    case 1: /* Big endian, normal word order */
        adjusted_data = data;
        break;
    case 2: /* Big endian, reverse word order */
        adjusted_data = ((data & 0xFF000000) >> 24) |
                        ((data & 0x00FF0000) >> 8) |
                        ((data & 0x0000FF00) << 8) |
                        ((data & 0x000000FF) << 24);
        break;
    case 3: /* Little endian, normal word order */
        adjusted_data = ((data & 0xFF000000) >> 24) |
                        ((data & 0x00FF0000) >> 8) |
                        ((data & 0x0000FF00) << 8) |
                        ((data & 0x000000FF) << 24);
        break;
    case 4: /* Little endian, reverse word order */
        adjusted_data = ((data & 0x000000FF) << 24) |
                        ((data & 0x0000FF00) << 8) |
                        ((data & 0x00FF0000) >> 8) |
                        ((data & 0xFF000000) >> 24);
        break;
    default:
        adjusted_data = data;
        break;
    }

    /* Write 32-bit data directly to the FIFO */
    iowrite32(adjusted_data, FIFO_DATA(dev.virtbase));
    return 0;
}

/*
* Handle ioctl() calls from userspace
*/
static long fifo_write_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    fifo_write_arg_t fwa;
    int ret;

    switch (cmd) {
    case FIFO_WRITE_WRITE_DATA:
        if (copy_from_user(&fwa, (fifo_write_arg_t *) arg,
```

```
                sizeof(fifo_write_arg_t)))
            return -EACCES;

        ret = write_to_fifo(fwa.data);
        if (ret)
            return ret; /* Return error from write_to_fifo */
        break;

    case FIFO_WRITE_GET_LEVEL:
        fwa.data = read_fifo_level();
        if (copy_to_user((fifo_write_arg_t *) arg, &fwa,
                 sizeof(fifo_write_arg_t)))
            return -EACCES;
        break;

    case FIFO_WRITE_SET_CONFIG:
        if (get_user(config_mode, (int __user *)arg))
            return -EACCES;

        /* Validate config_mode */
        if (config_mode < 1 || config_mode > 4) {
            pr_err(DRIVER_NAME ": Invalid config_mode %d. Must be 1-4\n", config_mode);
            return -EINVAL;
        }

        pr_info(DRIVER_NAME ": Config mode set to %d\n", config_mode);
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fifo_write_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = fifo_write_ioctl,
};

/* Information about our device for the "misc" framework */
```

```c
static struct miscdevice fifo_write_misc_device = {
   .minor      = MISC_DYNAMIC_MINOR,
   .name       = DRIVER_NAME,
   .fops       = &fifo_write_fops,
};

/*
* Initialization code: get resources and set up device
*/
static int __init fifo_write_probe(struct platform_device *pdev)
{
   int ret;
   const char *node_name;
   struct resource *res_data, *res_csr;

   /* Check if this is the correct FIFO device (fifo_0) */
   node_name = pdev->dev.of_node->name;
   pr_info(DRIVER_NAME ": Probing node name=%s\n", node_name ? node_name : "NULL");

   /* Check for exact fifo name first, fallback to checking if it contains "fifo" */
   if (node_name && ((strcmp(node_name, "fifo@0x100000148") == 0) ||
                     (strcmp(node_name, "fifo_0") == 0) ||
                     strstr(node_name, "fifo") != NULL)) {
       const __be32 *reg = of_get_property(pdev->dev.of_node, "reg", NULL);
       /* For fifo_0, the first register region has offset 0x48 at index 1 in the reg
array */
       if (reg && (be32_to_cpu(reg[1]) == 0x148)) {
           /* This is fifo_0, which is the one we want for writing */
           pr_info(DRIVER_NAME ": Found input FIFO at offset 0x%x\n",
be32_to_cpu(reg[1]));
       } else {
           /* This is not fifo_0 */
           pr_info(DRIVER_NAME ": Not the input FIFO\n");
           return -ENODEV;
       }
   } else {
       /* Not a FIFO node */
       pr_info(DRIVER_NAME ": Not a FIFO node\n");
       return -ENODEV;
   }

   /* Register ourselves as a misc device: creates /dev/fifo_write */
```

```
    ret = misc_register(&fifo_write_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": Couldn't register misc device\n");
        return ret;
    }

    /* Get the data port and CSR addresses */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in_csr");

    if (!res_data || !res_csr) {
        pr_err(DRIVER_NAME ": Could not get FIFO resources\n");
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(res_data->start, resource_size(res_data), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    if (request_mem_region(res_csr->start, resource_size(res_csr), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_release_data_mem_region;
    }

    /* Arrange access to data registers */
    dev.virtbase = ioremap(res_data->start, resource_size(res_data));
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_csr_mem_region;
    }

    /* Arrange access to CSR registers */
    dev.csr_virtbase = ioremap(res_csr->start, resource_size(res_csr));
    if (dev.csr_virtbase == NULL) {
        ret = -ENOMEM;
        goto out_unmap_data;
    }
```

```
    return 0;

out_unmap_data:
    iounmap(dev.virtbase);
out_release_csr_mem_region:
    release_mem_region(res_csr->start, resource_size(res_csr));
out_release_data_mem_region:
    release_mem_region(res_data->start, resource_size(res_data));
out_deregister:
    misc_deregister(&fifo_write_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int fifo_write_remove(struct platform_device *pdev)
{
    struct resource *res_data, *res_csr;

    /* Get the resources again to release them properly */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in_csr");

    /* Unmap the virtual memory */
    iounmap(dev.virtbase);
    iounmap(dev.csr_virtbase);

    /* Release the memory regions */
    if (res_data)
        release_mem_region(res_data->start, resource_size(res_data));
    if (res_csr)
        release_mem_region(res_csr->start, resource_size(res_csr));

    /* Deregister the misc device */
    misc_deregister(&fifo_write_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fifo_write_of_match[] = {
    { .compatible = "ALTR,fifo-21.1" },
```

```
    {},
};
MODULE_DEVICE_TABLE(of, fifo_write_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver fifo_write_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(fifo_write_of_match),
    },
    .probe = fifo_write_probe,
    .remove = __exit_p(fifo_write_remove),
};

/* Called when the module is loaded: set things up */
static int __init fifo_write_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_register(&fifo_write_driver);
}

/* Called when the module is unloaded: release resources */
static void __exit fifo_write_exit(void)
{
    platform_driver_unregister(&fifo_write_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(fifo_write_init);
module_exit(fifo_write_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("FIFO write driver");
```

fifo_write.h

```
#ifndef _FIFO_WRITE_H
#define _FIFO_WRITE_H

#include <linux/ioctl.h>
```

```
#include <linux/types.h>

typedef struct {
   unsigned int data;
} fifo_write_arg_t;

#define FIFO_WRITE_MAGIC 'f'

#define FIFO_WRITE_WRITE_DATA _IOW(FIFO_WRITE_MAGIC, 0, fifo_write_arg_t)
#define FIFO_WRITE_GET_LEVEL  _IOR(FIFO_WRITE_MAGIC, 1, fifo_write_arg_t *)
#define FIFO_WRITE_SET_CONFIG _IOW(FIFO_WRITE_MAGIC, 2, int)

#endif
```

Makefile

```
# Makefile for FIFO drivers and userspace programs

ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
obj-m := fifo_write.o fifo_read.o

else

# We are being compiled as a module: use the Kernel build system
KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
PWD := $(shell pwd)

# Default target: build everything
default: modules userspace

# Target to build kernel modules
modules:
   @echo "Building kernel modules..."
   ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

# Target to build userspace programs
userspace: write_data read_data fifo_check

# Compile the writer program
write_data: write_data.c fifo_write.h
   @echo "Building write_data..."
```

```
    $(CC) -Wall -o write_data write_data.c

# Compile the reader program
read_data: read_data.c fifo_read.h
    @echo "Building read_data..."
    $(CC) -Wall -o read_data read_data.c

# Compile the FIFO check utility
fifo_check: fifo_check.c fifo_write.h fifo_read.h
    @echo "Building fifo_check..."
    $(CC) -Wall -o fifo_check fifo_check.c

# Clean up compiled files
clean:
    @echo "Cleaning up..."
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
    rm -f write_data read_data fifo_check

# Install the modules
install:
    @echo "Installing kernel modules..."
    insmod fifo_write.ko
    insmod fifo_read.ko
    @echo "Modules installed successfully"

# Remove the modules
uninstall:
    @echo "Removing kernel modules..."
    -rmmod fifo_read
    -rmmod fifo_write
    @echo "Modules removed successfully"

TARFILES = Makefile fifo_write.h fifo_write.c fifo_read.h fifo_read.c write_data.c
read_data.c fifo_check.c fifo_start.sh fifo_cleanup.sh
TARFILE = fifo-drivers.tar.gz
.PHONY: tar
tar: $(TARFILE)

$(TARFILE): $(TARFILES)
    tar zcfC $(TARFILE) .. $(TARFILES:%=sw/%)

# Phony targets (not files)
```

```
.PHONY: default modules userspace clean install uninstall

endif
```

read_data.c

```c
/*
* Userspace program that communicates with the fifo_read device driver
* through ioctls to read data from the FIFO
* Modified to write raw bytes to an output file
*/

#include <stdio.h>
#include "fifo_read.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <stdbool.h>
#include <errno.h>

int fifo_read_fd;
FILE *output_file = NULL;
bool running = true;
int total_reads = 0;

/* Signal handler to gracefully exit */
void handle_signal(int sig) {
   printf("\nReceived signal %d. Stopping...\n", sig);
   running = false;
}

/* Read a 32-bit value from the FIFO */
int read_value_from_fifo(unsigned int *value)
{
   fifo_read_arg_t fra;
   int ret;

   // Clear errno before ioctl call
```

```
    errno = 0;

    ret = ioctl(fifo_read_fd, FIFO_READ_READ_DATA, &fra);

    if (ret < 0) {
        if (errno == EAGAIN) {
            // FIFO is empty
            return -EAGAIN;
        } else {
            // Some other error
            return -EIO;
        }
    }

    // Get the full 32-bit value
    *value = (unsigned int)fra.data;
    return 0;
}

int main(int argc, char *argv[])
{
    static const char filename[] = "/dev/fifo_read";
    int ret;
    unsigned int value;
    const int delay_us = 1000; // Fixed delay between reads
    int config_mode = 0; // Default: don't change configuration

    printf("FIFO Read Userspace program started\n");

    if (argc < 2 || argc > 3) {
        printf("Usage: %s <output_file> [config_mode]\n", argv[0]);
        printf("  <output_file>: File to write the read values to\n");
        printf("  [config_mode]: Endianness and word order (1-4):\n");
        printf("    1 = Big endian, normal word order\n");
        printf("    2 = Big endian, reverse word order\n");
        printf("    3 = Little endian, normal word order\n");
        printf("    4 = Little endian, reverse word order\n");
        printf("\nPress Ctrl+C to stop reading\n");
        return -1;
    }

    // Open the output file in binary mode
```

```
    output_file = fopen(argv[1], "wb");
    if (!output_file) {
        perror("Failed to open output file");
        return -1;
    }

    // Get optional config_mode
    if (argc == 3) {
        config_mode = atoi(argv[2]);
        if (config_mode < 1 || config_mode > 4) {
            printf("Error: config_mode must be between 1 and 4\n");
            fclose(output_file);
            return -1;
        }
    }

    if ((fifo_read_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        fclose(output_file);
        return -1;
    }

    // Set configuration mode if specified
    if (config_mode > 0) {
        if (ioctl(fifo_read_fd, FIFO_READ_SET_CONFIG, &config_mode) < 0) {
            perror("Error setting configuration mode");
            close(fifo_read_fd);
            fclose(output_file);
            return -1;
        }
        printf("Set config mode to %d\n", config_mode);
    }

    // Set up signal handler for Ctrl+C
    signal(SIGINT, handle_signal);

    printf("Reading continuously from FIFO until Ctrl+C, writing to %s\n", argv[1]);

    // Read loop - continues until Ctrl+C is pressed
    while (running) {
        ret = read_value_from_fifo(&value);
```

```
        if (ret < 0) {
            if (ret == -EAGAIN) {
                // FIFO is empty, wait a bit and try again
                usleep(delay_us);
                continue;
            } else {
                // Some other error occurred, but we'll keep trying
                fprintf(stderr, "Error reading from FIFO: %d, continuing...\n", ret);
                usleep(delay_us * 10);  // Wait a bit longer on error
                continue;
            }
        }

        // Write full 32-bit value directly to the file
        fwrite(&value, sizeof(unsigned int), 1, output_file);

        // Flush to ensure data is written immediately
        fflush(output_file);

        total_reads++;

        // Fixed delay between reads
        usleep(delay_us);
    }

    fclose(output_file);
    printf("FIFO Read complete - wrote %d values (%d bytes) to %s\n",
           total_reads, total_reads * sizeof(unsigned int), argv[1]);
    printf("FIFO Read Userspace program terminating\n");
    return 0;
}
```

write_data.c

```
/*
* Userspace program that communicates with the fifo_write device driver
* through ioctls to write data to the FIFO
* Modified to read raw bytes from a file
*/

#include <stdio.h>
#include "fifo_write.h"
#include <sys/ioctl.h>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

int fifo_write_fd;

/* Write a 32-bit value to the FIFO */
int write_value_to_fifo(unsigned int value)
{
   fifo_write_arg_t fwa;
   int ret;

   // Put the 32-bit value into the data field
   fwa.data = value;

   ret = ioctl(fifo_write_fd, FIFO_WRITE_WRITE_DATA, &fwa);
   if (ret) {
       if (ret == -EBUSY) {
           printf("FIFO is full. Waiting for space...\n");
           return -EBUSY;
       } else {
           perror("ioctl(FIFO_WRITE_WRITE_DATA) failed");
           return -EIO;
       }
   }
   return 0;
}

int main(int argc, char *argv[])
{
   static const char filename[] = "/dev/fifo_write";
   FILE *input_file;
   int count = 0;
   const int delay_us = 1000; // Fixed delay between writes in microseconds
   int ret;
   int config_mode = 0; // Default: don't change configuration
   unsigned int value;
```

```
printf("FIFO Write Userspace program started\n");

if (argc < 2 || argc > 3) {
    printf("Usage: %s <input_file> [config_mode]\n", argv[0]);
    printf("  <input_file>: Path to file containing data to write\n");
    printf("  [config_mode]: Endianness and word order (1-4):\n");
    printf("    1 = Big endian, normal word order\n");
    printf("    2 = Big endian, reverse word order\n");
    printf("    3 = Little endian, normal word order\n");
    printf("    4 = Little endian, reverse word order\n");
    return -1;
}

// Open the input file in binary mode
input_file = fopen(argv[1], "rb");
if (!input_file) {
    perror("Failed to open input file");
    return -1;
}

// Get optional config_mode parameter
if (argc == 3) {
    config_mode = atoi(argv[2]);
    if (config_mode < 1 || config_mode > 4) {
        printf("Error: config_mode must be between 1 and 4\n");
        fclose(input_file);
        return -1;
    }
}

if ((fifo_write_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    fclose(input_file);
    return -1;
}

// Set configuration mode if specified
if (config_mode > 0) {
    if (ioctl(fifo_write_fd, FIFO_WRITE_SET_CONFIG, &config_mode) < 0) {
        perror("Error setting configuration mode");
        close(fifo_write_fd);
        fclose(input_file);
```

```
            return -1;
        }
        printf("Set config mode to %d\n", config_mode);
    }

    printf("Reading values from %s and writing to FIFO (delay: %d us)\n", argv[1],
delay_us);

    // Read the file 32 bits at a time
    while (fread(&value, sizeof(unsigned int), 1, input_file) == 1) {
        // Write to FIFO with retry on full
        while (1) {
            ret = write_value_to_fifo(value);
            if (ret == 0) {
                // Success
                count++;
                break;
            } else if (ret == -EBUSY) {
                // FIFO is full, wait a bit and retry
                usleep(delay_us);
                continue;
            } else {
                // Some other error
                fprintf(stderr, "Error writing to FIFO\n");
                break;
            }
        }

        // Fixed delay between writes
        usleep(delay_us);
    }

    fclose(input_file);

    printf("FIFO Write complete - wrote %d values (%d bytes)\n",
           count, count * sizeof(unsigned int));
    printf("FIFO Write Userspace program terminating\n");
    return 0;
}
```

Final Executable Code + Drivers

drivers/aes_ctl.c

```
/*
* Device driver for controlling the AES hardware
*
* A Platform device implemented using the misc subsystem with hardcoded address
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include "aes_ctl.h"

#define DRIVER_NAME "aes_ctl"

/* The memory address of the AES control register */
#define AES_CTL_PHYS_BASE 0xFF200150
#define AES_CTL_SIZE      4

/*
* Information about our device
*/
struct aes_ctl_dev {
   void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
```

```c
* Write data to the AES control register
*/
static int write_to_aes_ctl(unsigned int data)
{
   pr_info(DRIVER_NAME ": Writing value %u to register\n", data);

   /* Write 32-bit data directly to the control register */
   iowrite32(data, dev.virtbase);
   return 0;
}

/*
* Read data from the AES control register
*/
static int read_from_aes_ctl(unsigned int *data)
{
   /* Read 32-bit data directly from the control register */
   *data = ioread32(dev.virtbase);
   pr_info(DRIVER_NAME ": Read value %u from register\n", *data);
   return 0;
}

/*
* Handle ioctl() calls from userspace
*/
static long aes_ctl_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
   aes_ctl_arg_t ctrl_arg;

   switch (cmd) {
   case AES_CTL_WRITE_DATA:
       if (copy_from_user(&ctrl_arg, (aes_ctl_arg_t *) arg,
                  sizeof(aes_ctl_arg_t)))
           return -EACCES;

       return write_to_aes_ctl(ctrl_arg.data);

   case AES_CTL_READ_DATA:
       if (read_from_aes_ctl(&ctrl_arg.data) != 0)
           return -EIO;

       if (copy_to_user((aes_ctl_arg_t *) arg, &ctrl_arg,
```

```
                sizeof(aes_ctl_arg_t)))
            return -EACCES;

        return 0;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations aes_ctl_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = aes_ctl_ioctl,
};

/* Information about our device for the "misc" framework */
static struct miscdevice aes_ctl_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &aes_ctl_fops,
};

/* Called when the module is loaded: set things up */
static int __init aes_ctl_init(void)
{
    int ret;

    pr_info(DRIVER_NAME ": init\n");

    /* Map the AES control register directly using hardcoded address */
    if (!request_mem_region(AES_CTL_PHYS_BASE, AES_CTL_SIZE, DRIVER_NAME)) {
        pr_err(DRIVER_NAME ": Resources unavailable\n");
        return -EBUSY;
    }

    dev.virtbase = ioremap(AES_CTL_PHYS_BASE, AES_CTL_SIZE);
    if (!dev.virtbase) {
        pr_err(DRIVER_NAME ": ioremap failed\n");
        ret = -ENOMEM;
```

```
        goto out_release_mem_region;
    }

    /* Register ourselves as a misc device: creates /dev/aes_ctl */
    ret = misc_register(&aes_ctl_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": Couldn't register misc device\n");
        goto out_unmap;
    }

    pr_info(DRIVER_NAME ": Device registered successfully at 0x%lx\n",
        (unsigned long)AES_CTL_PHYS_BASE);

    return 0;

out_unmap:
    pr_err(DRIVER_NAME ": Failed at iounmap\n");
    iounmap(dev.virtbase);
out_release_mem_region:
    pr_err(DRIVER_NAME ": Failed at release_mem_region\n");
    release_mem_region(AES_CTL_PHYS_BASE, AES_CTL_SIZE);
    return ret;
}

/* Called when the module is unloaded: release resources */
static void __exit aes_ctl_exit(void)
{
    /* Unmap virtual memory */
    iounmap(dev.virtbase);

    /* Release memory region */
    release_mem_region(AES_CTL_PHYS_BASE, AES_CTL_SIZE);

    /* Deregister misc device */
    misc_deregister(&aes_ctl_misc_device);

    pr_info(DRIVER_NAME ": exit\n");
}

module_init(aes_ctl_init);
module_exit(aes_ctl_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("AES control driver");
```

drivers/aes_ctl.h

```
#ifndef _AES_CTL_H
#define _AES_CTL_H

#include <linux/ioctl.h>

typedef struct {
   unsigned int data;
} aes_ctl_arg_t;

#define AES_CTL_MAGIC 'a'
#define AES_CTL_WRITE_DATA _IOW(AES_CTL_MAGIC, 0, aes_ctl_arg_t)
#define AES_CTL_READ_DATA _IOR(AES_CTL_MAGIC, 1, aes_ctl_arg_t *)

#endif
```

drivers/fifo_read.c

```
/*
* Device driver for reading from the output FIFO
*
* A Platform device implemented using the misc subsystem
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include <linux/moduleparam.h>
```

```c
#include "fifo_read.h"

#define DRIVER_NAME "fifo_read"

/* Uses Big endian, reverse word order (config_mode 2) by default */

/* Device registers - FIFO is 8 bytes (64 bits) */
#define FIFO_DATA(x) (x)
#define FIFO_STATUS_REG(x) ((x)+0x4)    /* Status register (i_status) */

/* Status register bit definitions */
#define FIFO_EMPTY_BIT   1

/*
* Information about our device
*/
struct fifo_read_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   void __iomem *csr_virtbase; /* Control and Status Register base */
   unsigned int last_read;    /* Last value read from FIFO */
} dev;

/*
* Read the status register and return its value
*/
static u32 read_fifo_status(void)
{
   return ioread32(FIFO_STATUS_REG(dev.csr_virtbase));
}

/*
* Check if the FIFO is empty
*/
static int is_fifo_empty(void)
{
   u32 status = read_fifo_status();
   return (status & (1 << FIFO_EMPTY_BIT)) ? 1 : 0;
}

/*
* Read data from the FIFO
```

```c
* Returns the data read on success, negative error code on failure
* Uses Big endian, reverse word order (config_mode 2) by default
*/
static int read_from_fifo(void)
{
   unsigned int data;

   /* Check if the FIFO is empty before reading */
   if (is_fifo_empty()) {
       pr_info("FIFO is empty, cannot read data\n");
       return -EAGAIN; /* Resource temporarily unavailable */
   }

   /* Read 32-bit data from the FIFO */
   unsigned int raw_data = ioread32(FIFO_DATA(dev.virtbase));

   /* Process data using Big endian, reverse word order (config_mode 2) */
   data = ((raw_data & 0xFF000000) >> 24) |
          ((raw_data & 0x00FF0000) >> 8) |
          ((raw_data & 0x0000FF00) << 8) |
          ((raw_data & 0x000000FF) << 24);

   dev.last_read = data;
   return dev.last_read;
}

/*
* Handle ioctl() calls from userspace
*/
static long fifo_read_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
   fifo_read_arg_t fra;
   int ret;

   switch (cmd) {
   case FIFO_READ_READ_DATA:
       ret = read_from_fifo();
       if (ret < 0)
           return ret; /* Return error from read_from_fifo */

       fra.data = ret;
       if (copy_to_user((fifo_read_arg_t *) arg, &fra,
```

```
                sizeof(fifo_read_arg_t)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fifo_read_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = fifo_read_ioctl,
};

/* Information about our device for the "misc" framework */
static struct miscdevice fifo_read_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &fifo_read_fops,
};

/*
* Initialization code: get resources and set up device
*/
static int __init fifo_read_probe(struct platform_device *pdev)
{
    int ret;
    const char *node_name;
    struct resource *res_data, *res_csr;

    /* Check if this is the correct FIFO device (fifo_1) */
    node_name = pdev->dev.of_node->name;
    pr_info(DRIVER_NAME ": Probing node name=%s\n", node_name ? node_name : "NULL");

    /* Check for exact fifo name first, fallback to checking if it contains "fifo" */
    if (node_name && ((strcmp(node_name, "fifo@0x100000140") == 0) ||
                      (strcmp(node_name, "fifo_1") == 0) ||
                      strstr(node_name, "fifo") != NULL)) {
        const __be32 *reg = of_get_property(pdev->dev.of_node, "reg", NULL);
```

```
        /* For fifo_1, the first register region has offset 0x40 at index 1 in the reg
array */
        if (reg && (be32_to_cpu(reg[1]) == 0x140)) {
            /* This is fifo_1, which is the one we want for reading */
            pr_info(DRIVER_NAME ": Found output FIFO at offset 0x%x\n",
be32_to_cpu(reg[1]));
        } else {
            /* This is not fifo_1 */
            pr_info(DRIVER_NAME ": Not the output FIFO\n");
            return -ENODEV;
        }
    } else {
        /* Not a FIFO node */
        pr_info(DRIVER_NAME ": Not a FIFO node\n");
        return -ENODEV;
    }

    /* Register ourselves as a misc device: creates /dev/fifo_read */
    ret = misc_register(&fifo_read_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": Couldn't register misc device\n");
        return ret;
    }

    /* Get the data port and CSR addresses */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out_csr");

    if (!res_data || !res_csr) {
        pr_err(DRIVER_NAME ": Could not get FIFO resources\n");
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(res_data->start, resource_size(res_data), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }
```

```
    if (request_mem_region(res_csr->start, resource_size(res_csr), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_release_data_mem_region;
    }

    /* Arrange access to data registers */
    dev.virtbase = ioremap(res_data->start, resource_size(res_data));
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_csr_mem_region;
    }

    /* Arrange access to CSR registers */
    dev.csr_virtbase = ioremap(res_csr->start, resource_size(res_csr));
    if (dev.csr_virtbase == NULL) {
        ret = -ENOMEM;
        goto out_unmap_data;
    }

    return 0;

out_unmap_data:
    iounmap(dev.virtbase);
out_release_csr_mem_region:
    release_mem_region(res_csr->start, resource_size(res_csr));
out_release_data_mem_region:
    release_mem_region(res_data->start, resource_size(res_data));
out_deregister:
    misc_deregister(&fifo_read_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int fifo_read_remove(struct platform_device *pdev)
{
    struct resource *res_data, *res_csr;

    /* Get the resources again to release them properly */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "out_csr");
```

```
    /* Unmap the virtual memory */
    iounmap(dev.virtbase);
    iounmap(dev.csr_virtbase);

    /* Release the memory regions */
    if (res_data)
        release_mem_region(res_data->start, resource_size(res_data));
    if (res_csr)
        release_mem_region(res_csr->start, resource_size(res_csr));

    /* Deregister the misc device */
    misc_deregister(&fifo_read_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fifo_read_of_match[] = {
    { .compatible = "ALTR,fifo-21.1" },
    {},
};
MODULE_DEVICE_TABLE(of, fifo_read_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver fifo_read_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(fifo_read_of_match),
    },
    .probe = fifo_read_probe,
    .remove = __exit_p(fifo_read_remove),
};

/* Called when the module is loaded: set things up */
static int __init fifo_read_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_register(&fifo_read_driver);
}
```

```c
/* Called when the module is unloaded: release resources */
static void __exit fifo_read_exit(void)
{
   platform_driver_unregister(&fifo_read_driver);
   pr_info(DRIVER_NAME ": exit\n");
}

module_init(fifo_read_init);
module_exit(fifo_read_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("FIFO read driver");
```

drivers/fifo_read.h

```c
#ifndef _FIFO_READ_H
#define _FIFO_READ_H

#include <linux/ioctl.h>
#include <linux/types.h>

typedef struct {
   unsigned int data;
} fifo_read_arg_t;

#define FIFO_READ_MAGIC 'r'

#define FIFO_READ_READ_DATA _IOR(FIFO_READ_MAGIC, 0, fifo_read_arg_t)

#endif
```

drivers/fifo_write.c

```c
/*
* Device driver for writing to the input FIFO
*
* A Platform device implemented using the misc subsystem
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
```

```c
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include <linux/moduleparam.h>
#include "fifo_write.h"

#define DRIVER_NAME "fifo_write"

/* Uses Big endian, reverse word order (config_mode 2) by default */

/* Device registers - FIFO is 8 bytes (64 bits) */
#define FIFO_DATA(x) (x)
#define FIFO_STATUS_REG(x) ((x)+0x4)    /* Status register (i_status) */

/* Status register bit definitions */
#define FIFO_FULL_BIT    0

/*
* Information about our device
*/
struct fifo_write_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   void __iomem *csr_virtbase; /* Control and Status Register base */
} dev;

/*
* Read the status register and return its value
*/
static u32 read_fifo_status(void)
{
   return ioread32(FIFO_STATUS_REG(dev.csr_virtbase));
}

/*
```

```
* Check if the FIFO is full
*/
static int is_fifo_full(void)
{
   u32 status = read_fifo_status();
   return (status & (1 << FIFO_FULL_BIT)) ? 1 : 0;
}

/*
* Write data to the FIFO
* Returns 0 on success, -EBUSY if FIFO is full
* Uses Big endian, reverse word order (config_mode 2) by default
*/
static int write_to_fifo(unsigned int data)
{
   /* Check if the FIFO is full before writing */
   if (is_fifo_full()) {
       pr_info("FIFO is full, cannot write data\n");
       return -EBUSY; /* Device is busy (FIFO full) */
   }

   /* Process data using Big endian, reverse word order (config_mode 2) */
   unsigned int adjusted_data = ((data & 0xFF000000) >> 24) |
                        ((data & 0x00FF0000) >> 8) |
                        ((data & 0x0000FF00) << 8) |
                        ((data & 0x000000FF) << 24);

   /* Write 32-bit data directly to the FIFO */
   iowrite32(adjusted_data, FIFO_DATA(dev.virtbase));
   return 0;
}

/*
* Handle ioctl() calls from userspace
*/
static long fifo_write_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
   fifo_write_arg_t fwa;
   int ret;

   switch (cmd) {
   case FIFO_WRITE_WRITE_DATA:
```

```
        if (copy_from_user(&fwa, (fifo_write_arg_t *) arg,
                   sizeof(fifo_write_arg_t)))
            return -EACCES;

        ret = write_to_fifo(fwa.data);
        if (ret)
            return ret; /* Return error from write_to_fifo */
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fifo_write_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = fifo_write_ioctl,
};

/* Information about our device for the "misc" framework */
static struct miscdevice fifo_write_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &fifo_write_fops,
};

/*
* Initialization code: get resources and set up device
*/
static int __init fifo_write_probe(struct platform_device *pdev)
{
    int ret;
    const char *node_name;
    struct resource *res_data, *res_csr;

    /* Check if this is the correct FIFO device (fifo_0) */
    node_name = pdev->dev.of_node->name;
    pr_info(DRIVER_NAME ": Probing node name=%s\n", node_name ? node_name : "NULL");
```

```
    /* Check for exact fifo name first, fallback to checking if it contains "fifo" */
    if (node_name && ((strcmp(node_name, "fifo@0x100000148") == 0) ||
                      (strcmp(node_name, "fifo_0") == 0) ||
                      strstr(node_name, "fifo") != NULL)) {
        const __be32 *reg = of_get_property(pdev->dev.of_node, "reg", NULL);
        /* For fifo_0, the first register region has offset 0x48 at index 1 in the reg
array */
        if (reg && (be32_to_cpu(reg[1]) == 0x148)) {
            /* This is fifo_0, which is the one we want for writing */
            pr_info(DRIVER_NAME ": Found input FIFO at offset 0x%x\n",
be32_to_cpu(reg[1]));
        } else {
            /* This is not fifo_0 */
            pr_info(DRIVER_NAME ": Not the input FIFO\n");
            return -ENODEV;
        }
    } else {
        /* Not a FIFO node */
        pr_info(DRIVER_NAME ": Not a FIFO node\n");
        return -ENODEV;
    }

    /* Register ourselves as a misc device: creates /dev/fifo_write */
    ret = misc_register(&fifo_write_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": Couldn't register misc device\n");
        return ret;
    }

    /* Get the data port and CSR addresses */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in_csr");

    if (!res_data || !res_csr) {
        pr_err(DRIVER_NAME ": Could not get FIFO resources\n");
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(res_data->start, resource_size(res_data), DRIVER_NAME) ==
NULL) {
```

```
        ret = -EBUSY;
        goto out_deregister;
    }

    if (request_mem_region(res_csr->start, resource_size(res_csr), DRIVER_NAME) ==
NULL) {
        ret = -EBUSY;
        goto out_release_data_mem_region;
    }

    /* Arrange access to data registers */
    dev.virtbase = ioremap(res_data->start, resource_size(res_data));
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_csr_mem_region;
    }

    /* Arrange access to CSR registers */
    dev.csr_virtbase = ioremap(res_csr->start, resource_size(res_csr));
    if (dev.csr_virtbase == NULL) {
        ret = -ENOMEM;
        goto out_unmap_data;
    }

    return 0;

out_unmap_data:
    iounmap(dev.virtbase);
out_release_csr_mem_region:
    release_mem_region(res_csr->start, resource_size(res_csr));
out_release_data_mem_region:
    release_mem_region(res_data->start, resource_size(res_data));
out_deregister:
    misc_deregister(&fifo_write_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int fifo_write_remove(struct platform_device *pdev)
{
    struct resource *res_data, *res_csr;
```

```
    /* Get the resources again to release them properly */
    res_data = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in");
    res_csr = platform_get_resource_byname(pdev, IORESOURCE_MEM, "in_csr");

    /* Unmap the virtual memory */
    iounmap(dev.virtbase);
    iounmap(dev.csr_virtbase);

    /* Release the memory regions */
    if (res_data)
        release_mem_region(res_data->start, resource_size(res_data));
    if (res_csr)
        release_mem_region(res_csr->start, resource_size(res_csr));

    /* Deregister the misc device */
    misc_deregister(&fifo_write_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fifo_write_of_match[] = {
    { .compatible = "ALTR,fifo-21.1" },
    {},
};
MODULE_DEVICE_TABLE(of, fifo_write_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver fifo_write_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(fifo_write_of_match),
    },
    .probe = fifo_write_probe,
    .remove = __exit_p(fifo_write_remove),
};

/* Called when the module is loaded: set things up */
static int __init fifo_write_init(void)
{
```

```
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_register(&fifo_write_driver);
}

/* Called when the module is unloaded: release resources */
static void __exit fifo_write_exit(void)
{
    platform_driver_unregister(&fifo_write_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(fifo_write_init);
module_exit(fifo_write_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("FIFO write driver");
```

drivers/fifo_write.h

```
#ifndef _FIFO_WRITE_H
#define _FIFO_WRITE_H

#include <linux/ioctl.h>
#include <linux/types.h>

typedef struct {
    unsigned int data;
} fifo_write_arg_t;

#define FIFO_WRITE_MAGIC 'f'

#define FIFO_WRITE_WRITE_DATA _IOW(FIFO_WRITE_MAGIC, 0, fifo_write_arg_t)

#endif
```

drivers/Makefile

```
# Makefile for AES kernel modules
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
obj-m := aes_ctl.o fifo_read.o fifo_write.o round_keys.o
```

```
else

# We are being compiled as a module: use the Kernel build system
KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
PWD := $(shell pwd)

# Default target: build everything
default: modules install

# Target to build kernel modules
modules:
	@echo "Building kernel modules..."
	${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

# Clean up compiled files
clean:
	@echo "Cleaning up..."
	${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean

# Install the modules
install:
	@echo "Installing kernel modules..."
	-insmod aes_ctl.ko
	-insmod fifo_read.ko
	-insmod fifo_write.ko
	-insmod round_keys.ko
	@echo "Modules installed successfully"

# Remove the modules
uninstall:
	@echo "Removing kernel modules..."
	-rmmod aes_ctl
	-rmmod fifo_read
	-rmmod fifo_write
	-rmmod round_keys
	@echo "Modules removed successfully if they were loaded"

# Restart: uninstall then install
restart: uninstall install

# Phony targets (not files)
.PHONY: default modules clean install uninstall restart
```

```
endif
```

drivers/round_keys.c

```
/*
* Device driver for accessing AES round keys RAM
*
* A kernel module implemented using the misc subsystem with hardcoded address
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include "round_keys.h"

#define DRIVER_NAME "round_keys"

/* The memory address of the round keys RAM */
#define ROUND_KEYS_PHYS_BASE 0xFF200000

/* Uses Big endian, reverse word order (config_mode 2) by default */

/*
* Information about our device
*/
struct round_keys_dev {
   void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
* Write data to the round keys RAM using Big endian, reverse word order (mode 2)
*/
static int write_round_keys(const unsigned char *data)
```

```
{
    int i;
    int words = ROUND_KEYS_SIZE / 4; /* Number of 32-bit words */
    u32 word;

    pr_info(DRIVER_NAME ": Writing %d bytes of round keys data (%d words)\n",
        ROUND_KEYS_SIZE, words);

    /* Write all the round keys, 32 bits at a time */
    for (i = 0; i < words; i++) {
        /* Reverse word order within each round key (mode 2) */
        int target_idx = (i / 4) * 4 + (3 - (i % 4));

        /* Pack 4 bytes into a 32-bit word (big endian) */
        word = (data[i*4] << 24) |
               (data[i*4+1] << 16) |
               (data[i*4+2] << 8) |
               (data[i*4+3] << 0);

        iowrite32(word, dev.virtbase + (target_idx * 4));
    }

    return 0;
}

/*
* Read data from the round keys RAM using Big endian, reverse word order (mode 2)
*/
static int read_round_keys(unsigned char *data)
{
    int i;
    int words = ROUND_KEYS_SIZE / 4; /* Number of 32-bit words */
    u32 word;

    pr_info(DRIVER_NAME ": Reading %d bytes of round keys data (%d words)\n",
        ROUND_KEYS_SIZE, words);

    /* Read all the round keys, 32 bits at a time */
    for (i = 0; i < words; i++) {
        /* Reverse word order within each round key (mode 2) */
        int source_idx = (i / 4) * 4 + (3 - (i % 4));
```

```
        word = ioread32(dev.virtbase + (source_idx * 4));

        /* Unpack 32-bit word into 4 bytes (big endian) */
        data[i*4]   = (word >> 24) & 0xFF;
        data[i*4+1] = (word >> 16) & 0xFF;
        data[i*4+2] = (word >> 8) & 0xFF;
        data[i*4+3] = word & 0xFF;
    }

    return 0;
}

/*
* Handle ioctl() calls from userspace
*/
static long round_keys_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    round_keys_data_t rk_data;

    switch (cmd) {
    case ROUND_KEYS_WRITE_DATA:
        if (copy_from_user(&rk_data, (round_keys_data_t *) arg,
                    sizeof(round_keys_data_t)))
            return -EACCES;

        return write_round_keys(rk_data.data);

    case ROUND_KEYS_READ_DATA:
        memset(&rk_data, 0, sizeof(round_keys_data_t));

        read_round_keys(rk_data.data);

        if (copy_to_user((round_keys_data_t *) arg, &rk_data,
                  sizeof(round_keys_data_t)))
            return -EACCES;

        return 0;

    default:
        return -EINVAL;
    }
```

```
    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations round_keys_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = round_keys_ioctl,
};

/* Information about our device for the "misc" framework */
static struct miscdevice round_keys_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &round_keys_fops,
};

/* Called when the module is loaded: set things up */
static int __init round_keys_init(void)
{
    int ret;

    pr_info(DRIVER_NAME ": init\n");

    /* Map the round keys RAM directly using hardcoded address */
    if (!request_mem_region(ROUND_KEYS_PHYS_BASE, ROUND_KEYS_SIZE, DRIVER_NAME)) {
        pr_err(DRIVER_NAME ": Resources unavailable\n");
        return -EBUSY;
    }

    dev.virtbase = ioremap(ROUND_KEYS_PHYS_BASE, ROUND_KEYS_SIZE);
    if (!dev.virtbase) {
        pr_err(DRIVER_NAME ": ioremap failed\n");
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    /* Register ourselves as a misc device: creates /dev/round_keys */
    ret = misc_register(&round_keys_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": Couldn't register misc device\n");
        goto out_unmap;
    }
```

```
    pr_info(DRIVER_NAME ": Device registered successfully at 0x%lx\n",
        (unsigned long)ROUND_KEYS_PHYS_BASE);

    return 0;

out_unmap:
    pr_err(DRIVER_NAME ": Failed at iounmap\n");
    iounmap(dev.virtbase);
out_release_mem_region:
    pr_err(DRIVER_NAME ": Failed at release_mem_region\n");
    release_mem_region(ROUND_KEYS_PHYS_BASE, ROUND_KEYS_SIZE);
    return ret;
}

/* Called when the module is unloaded: release resources */
static void __exit round_keys_exit(void)
{
    /* Unmap virtual memory */
    iounmap(dev.virtbase);

    /* Release memory region */
    release_mem_region(ROUND_KEYS_PHYS_BASE, ROUND_KEYS_SIZE);

    /* Deregister misc device */
    misc_deregister(&round_keys_misc_device);

    pr_info(DRIVER_NAME ": exit\n");
}

module_init(round_keys_init);
module_exit(round_keys_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Student");
MODULE_DESCRIPTION("AES round keys RAM driver");
```

drivers/round_keys.h

```
#ifndef _ROUND_KEYS_H
#define _ROUND_KEYS_H

#include <linux/ioctl.h>
```

```
#define ROUND_KEYS_SIZE 176 // 11 round keys * 16 bytes each

typedef struct {
   unsigned char data[ROUND_KEYS_SIZE];
} round_keys_data_t;

/* IOCTL commands */
#define ROUND_KEYS_MAGIC 'r'
#define ROUND_KEYS_WRITE_DATA _IOW(ROUND_KEYS_MAGIC, 0, round_keys_data_t)
#define ROUND_KEYS_READ_DATA _IOR(ROUND_KEYS_MAGIC, 1, round_keys_data_t)

#endif
```

aes_wav.c

```
/*
* AES WAV File Encryptor/Decryptor
*
* Userspace program that encrypts or decrypts WAV files using the hardware
* AES implementation via FIFOs and control registers.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <errno.h>
#include <stdbool.h>
#include <getopt.h>
#include "drivers/fifo_write.h"
#include "drivers/fifo_read.h"
#include "drivers/aes_ctl.h"
#include "drivers/round_keys.h"

// AES is a block cipher with 16-byte (128-bit) blocks
#define AES_BLOCK_SIZE 16

// Default FIFO processing delay in microseconds
```

```c
#define DEFAULT_FIFO_DELAY 1000

// Buffer for reading/writing data
#define BUFFER_SIZE 4096

// AES Control Values
#define AES_ENCRYPT_MODE 1
#define AES_DECRYPT_MODE 3

// Define WAV header structure
typedef struct {
   // RIFF Header
   char riff_header[4];      // Contains "RIFF"
   uint32_t wav_size;        // Size of the wav portion of the file
   char wave_header[4];      // Contains "WAVE"

   // Format Header
   char fmt_header[4];       // Contains "fmt " (note space after fmt)
   uint32_t fmt_chunk_size;  // Should be 16 for PCM
   uint16_t audio_format;    // Should be 1 for PCM
   uint16_t num_channels;    // 1 for mono, 2 for stereo
   uint32_t sample_rate;     // 8000, 44100, etc.
   uint32_t byte_rate;       // SampleRate * NumChannels * BitsPerSample/8
   uint16_t block_align;     // NumChannels * BitsPerSample/8
   uint16_t bits_per_sample; // 8 bits, 16 bits, etc.

   // Data Header
   char data_header[4];      // Contains "data"
   uint32_t data_size;       // Size of the data section
} WAVHeader;

// File descriptors for device drivers
int fifo_write_fd = -1;
int fifo_read_fd = -1;
int aes_ctl_fd = -1;
int round_keys_fd = -1;

// Program options and state
typedef struct {
   bool encrypt;
   bool decrypt;
   char *input_file;
```

```
    char *output_file;
    char *key_hex;
    int fifo_delay;
} program_options;

// Function prototypes
static void print_usage(const char *program_name);
static int parse_arguments(int argc, char *argv[], program_options *options);
static int initialize_devices(void);
static void cleanup_devices(void);
static int load_round_keys(const char *key_hex);
static int set_aes_mode(int mode);
static int process_wav_file(program_options *options);
static int read_wav_header(FILE *fp, WAVHeader *header);
static int write_to_fifo(uint32_t value);
static int read_from_fifo(uint32_t *value);
static int hex_to_bytes(const char* hex_str, uint8_t* bytes, size_t bytes_len);
static void print_hex(const char* prefix, const uint8_t* data, size_t length);
static uint8_t* apply_padding(uint8_t* data, uint32_t data_size, uint32_t*
padded_size);
static uint8_t* remove_padding(uint8_t* data, uint32_t data_size, uint32_t*
unpadded_size);
static int process_data_block(uint8_t* input_data, uint32_t input_size, uint8_t**
output_data, uint32_t* output_size, bool encrypt, int fifo_delay);

int main(int argc, char *argv[]) {
    program_options options = {
        .encrypt = false,
        .decrypt = false,
        .input_file = NULL,
        .output_file = NULL,
        .key_hex = NULL,
        .fifo_delay = DEFAULT_FIFO_DELAY
    };
    int ret;

    printf("AES WAV File Processor\n");

    // Parse command line arguments
    ret = parse_arguments(argc, argv, &options);
    if (ret != 0) {
        return ret;
```

```
    }

    // Initialize device drivers
    ret = initialize_devices();
    if (ret != 0) {
        return ret;
    }

    // Load AES round keys
    ret = load_round_keys(options.key_hex);
    if (ret != 0) {
        cleanup_devices();
        return ret;
    }

    // Set AES mode (encrypt or decrypt)
    if (options.encrypt) {
        ret = set_aes_mode(AES_ENCRYPT_MODE);
    } else if (options.decrypt) {
        ret = set_aes_mode(AES_DECRYPT_MODE);
    }

    if (ret != 0) {
        cleanup_devices();
        return ret;
    }

    // Process the WAV file
    ret = process_wav_file(&options);
    if (ret != 0) {
        cleanup_devices();
        return ret;
    }

    // Cleanup and exit
    cleanup_devices();
    printf("\nProcessing complete! Output written to %s\n", options.output_file);
    return 0;
}

// Print program usage information
static void print_usage(const char *program_name) {
```

```
    printf("Usage: %s [options]\n\n", program_name);
    printf("Options:\n");
    printf("  -e, --encrypt          Encrypt the input file\n");
    printf("  -d, --decrypt          Decrypt the input file\n");
    printf("  -i, --input FILE       Input WAV file\n");
    printf("  -o, --output FILE      Output WAV file\n");
    printf("  -k, --key HEX          AES key as 32 hex characters (16 bytes)\n");
    printf("  -t, --delay TIME       Delay between FIFO operations in microseconds
(default: %d)\n", DEFAULT_FIFO_DELAY);
    printf("  -h, --help             Show this help message\n\n");
    printf("Example (encrypt):\n");
    printf("  %s --encrypt --input input.wav --output encrypted.wav --key
2B7E151628AED2A6ABF7158809CF4F3C\n\n", program_name);
    printf("Example (decrypt):\n");
    printf("  %s --decrypt --input encrypted.wav --output decrypted.wav --key
2B7E151628AED2A6ABF7158809CF4F3C\n", program_name);
}

// Parse command line arguments
static int parse_arguments(int argc, char *argv[], program_options *options) {
    static struct option long_options[] = {
        {"encrypt", no_argument,       0, 'e'},
        {"decrypt", no_argument,       0, 'd'},
        {"input",   required_argument, 0, 'i'},
        {"output",  required_argument, 0, 'o'},
        {"key",     required_argument, 0, 'k'},
        {"delay",   required_argument, 0, 't'},
        {"help",    no_argument,       0, 'h'},
        {0, 0, 0, 0}
    };

    int opt;
    int option_index = 0;

    while ((opt = getopt_long(argc, argv, "edi:o:k:t:h", long_options, &option_index))
!= -1) {
        switch (opt) {
            case 'e':
                options->encrypt = true;
                break;
            case 'd':
                options->decrypt = true;
```

```
            break;
        case 'i':
            options->input_file = optarg;
            break;
        case 'o':
            options->output_file = optarg;
            break;
        case 'k':
            options->key_hex = optarg;
            break;
        case 't':
            options->fifo_delay = atoi(optarg);
            if (options->fifo_delay < 0) {
                options->fifo_delay = DEFAULT_FIFO_DELAY;
            }
            break;
        case 'h':
            print_usage(argv[0]);
            return 1;
        default:
            print_usage(argv[0]);
            return 1;
    }
}

// Validate arguments
if (options->encrypt && options->decrypt) {
    printf("Error: Cannot specify both encrypt and decrypt modes\n");
    print_usage(argv[0]);
    return 1;
}

if (!options->encrypt && !options->decrypt) {
    printf("Error: Must specify either encrypt or decrypt mode\n");
    print_usage(argv[0]);
    return 1;
}

if (options->input_file == NULL) {
    printf("Error: Input file must be specified\n");
    print_usage(argv[0]);
    return 1;
```

```
    }

    if (options->output_file == NULL) {
        printf("Error: Output file must be specified\n");
        print_usage(argv[0]);
        return 1;
    }

    if (options->key_hex == NULL) {
        printf("Error: AES key must be specified\n");
        print_usage(argv[0]);
        return 1;
    }

    // Validate key length (16 bytes = 32 hex characters)
    if (strlen(options->key_hex) != 32) {
        printf("Error: AES key must be 32 hex characters (16 bytes)\n");
        print_usage(argv[0]);
        return 1;
    }

    return 0;
}

// Initialize device drivers
static int initialize_devices(void) {
    // Open the FIFO write device
    fifo_write_fd = open("/dev/fifo_write", O_RDWR);
    if (fifo_write_fd < 0) {
        perror("Error opening /dev/fifo_write");
        return 1;
    }

    // Open the FIFO read device
    fifo_read_fd = open("/dev/fifo_read", O_RDWR);
    if (fifo_read_fd < 0) {
        perror("Error opening /dev/fifo_read");
        close(fifo_write_fd);
        return 1;
    }

    // Open the AES control device
```

```
    aes_ctl_fd = open("/dev/aes_ctl", O_RDWR);
    if (aes_ctl_fd < 0) {
        perror("Error opening /dev/aes_ctl");
        close(fifo_write_fd);
        close(fifo_read_fd);
        return 1;
    }

    // Open the round keys device
    round_keys_fd = open("/dev/round_keys", O_RDWR);
    if (round_keys_fd < 0) {
        perror("Error opening /dev/round_keys");
        close(fifo_write_fd);
        close(fifo_read_fd);
        close(aes_ctl_fd);
        return 1;
    }

    printf("All devices opened successfully\n");
    return 0;
}

// Clean up device handles
static void cleanup_devices(void) {
    if (fifo_write_fd >= 0) {
        close(fifo_write_fd);
    }
    if (fifo_read_fd >= 0) {
        close(fifo_read_fd);
    }
    if (aes_ctl_fd >= 0) {
        close(aes_ctl_fd);
    }
    if (round_keys_fd >= 0) {
        close(round_keys_fd);
    }
}

// Function to write a 32-bit value to the AES control register
static int set_aes_mode(int mode) {
    aes_ctl_arg_t ctrl_arg;
```

```
    printf("Setting AES mode to %s\n", mode == AES_ENCRYPT_MODE ? "ENCRYPT" :
"DECRYPT");

    // Set the control value
    ctrl_arg.data = mode;

    if (ioctl(aes_ctl_fd, AES_CTL_WRITE_DATA, &ctrl_arg) != 0) {
        perror("ioctl(AES_CTL_WRITE_DATA) failed");
        return 1;
    }

    printf("AES mode set successfully\n");
    return 0;
}

// Load AES round keys into RAM
static int load_round_keys(const char *key_hex) {
    uint8_t initial_key[16];

    // Convert hex string to bytes
    if (!hex_to_bytes(key_hex, initial_key, 16)) {
        printf("Error: Invalid key format\n");
        return 1;
    }

    printf("Using AES key: ");
    print_hex("", initial_key, 16);
    printf("\n");

    // Create a full round keys structure
    // Note: We'll use the existing driver to expand the key and write to RAM
    char load_keys_cmd[200];
    snprintf(load_keys_cmd, sizeof(load_keys_cmd), "./load_round_keys %s", key_hex);

    printf("Loading round keys with little endian, reverse word order...\n");
    int result = system(load_keys_cmd);
    if (result != 0) {
        printf("Error: Failed to load round keys (exit code: %d)\n", result);
        return 1;
    }

    printf("Round keys loaded successfully\n");
```

```c
    return 0;
}

// Read the WAV header from a file
static int read_wav_header(FILE *fp, WAVHeader *header) {
    // Read RIFF header
    if (fread(header->riff_header, 1, 4, fp) != 4 ||
        fread(&header->wav_size, 4, 1, fp) != 1 ||
        fread(header->wave_header, 1, 4, fp) != 4) {
        printf("Error: Could not read RIFF header\n");
        return 1;
    }

    // Check if file has "RIFF" signature
    if (memcmp(header->riff_header, "RIFF", 4) != 0) {
        printf("Error: Not a valid WAV file (missing RIFF signature)\n");
        return 1;
    }

    // Check if file has "WAVE" format
    if (memcmp(header->wave_header, "WAVE", 4) != 0) {
        printf("Error: Not a valid WAV file (missing WAVE format)\n");
        return 1;
    }

    // Read fmt chunk
    if (fread(header->fmt_header, 1, 4, fp) != 4 ||
        fread(&header->fmt_chunk_size, 4, 1, fp) != 1) {
        printf("Error: Could not read fmt chunk header\n");
        return 1;
    }

    // Verify fmt header
    if (memcmp(header->fmt_header, "fmt ", 4) != 0) {
        printf("Error: Missing fmt chunk\n");
        return 1;
    }

    // Read format data
    if (fread(&header->audio_format, 2, 1, fp) != 1 ||
        fread(&header->num_channels, 2, 1, fp) != 1 ||
        fread(&header->sample_rate, 4, 1, fp) != 1 ||
```

```
    fread(&header->byte_rate, 4, 1, fp) != 1 ||
    fread(&header->block_align, 2, 1, fp) != 1 ||
    fread(&header->bits_per_sample, 2, 1, fp) != 1) {
    printf("Error: Could not read format data\n");
    return 1;
}

// Skip any extra format bytes
if (header->fmt_chunk_size > 16) {
    if (fseek(fp, header->fmt_chunk_size - 16, SEEK_CUR) != 0) {
        printf("Error: Could not skip extra format bytes\n");
        return 1;
    }
}

// Find the data chunk (skip any non-data chunks)
bool found_data = false;
char chunk_id[4];
uint32_t chunk_size;

while (!found_data) {
    if (fread(chunk_id, 1, 4, fp) != 4 ||
        fread(&chunk_size, 4, 1, fp) != 1) {
        printf("Error: Unexpected end of file before data chunk\n");
        return 1;
    }

    if (memcmp(chunk_id, "data", 4) == 0) {
        found_data = true;
        memcpy(header->data_header, chunk_id, 4);
        header->data_size = chunk_size;
    } else {
        // Skip this non-data chunk
        if (fseek(fp, chunk_size, SEEK_CUR) != 0) {
            printf("Error: Could not skip non-data chunk\n");
            return 1;
        }
    }
}

printf("WAV header loaded successfully\n");
printf("  Format: %d\n", header->audio_format);
```

```
    printf("  Channels: %d\n", header->num_channels);
    printf("  Sample Rate: %d Hz\n", header->sample_rate);
    printf("  Bits per Sample: %d\n", header->bits_per_sample);
    printf("  Data Size: %u bytes\n", header->data_size);

    return 0;
}

// Write a value to the input FIFO
static int write_to_fifo(uint32_t value) {
    fifo_write_arg_t fwa;

    // Put the 32-bit value into the data field
    fwa.data = value;

    int ret = ioctl(fifo_write_fd, FIFO_WRITE_WRITE_DATA, &fwa);
    if (ret) {
        if (ret == -EBUSY) {
            // FIFO is full, return a special code to handle retry
            return -EBUSY;
        } else {
            perror("ioctl(FIFO_WRITE_WRITE_DATA) failed");
            return -EIO;
        }
    }
    return 0;
}

// Read a value from the output FIFO
static int read_from_fifo(uint32_t *value) {
    fifo_read_arg_t fra;

    int ret = ioctl(fifo_read_fd, FIFO_READ_READ_DATA, &fra);
    if (ret < 0) {
        if (errno == EAGAIN) {
            // FIFO is empty, return a special code to handle retry
            return -EAGAIN;
        } else {
            perror("ioctl(FIFO_READ_READ_DATA) failed");
            return -EIO;
        }
    }
```

```
    *value = fra.data;
    return 0;
}

// Function to convert hex string to bytes
static int hex_to_bytes(const char* hex_str, uint8_t* bytes, size_t bytes_len) {
    size_t hex_len = strlen(hex_str);

    // Check if hex string has the right length
    if (hex_len != bytes_len * 2) {
        return 0;
    }

    // Convert each pair of hex characters to a byte
    for (size_t i = 0; i < bytes_len; i++) {
        char byte_str[3] = {hex_str[i*2], hex_str[i*2+1], '\0'};
        char* end_ptr;
        bytes[i] = (uint8_t)strtol(byte_str, &end_ptr, 16);

        // Check if conversion was successful
        if (*end_ptr != '\0') {
            return 0;
        }
    }

    return 1;
}

// Print data in hexadecimal format
static void print_hex(const char* prefix, const uint8_t* data, size_t length) {
    printf("%s", prefix);
    for (size_t i = 0; i < length; i++) {
        printf("%02X", data[i]);
    }
}

// Apply PKCS#7 padding to make data a multiple of AES_BLOCK_SIZE
static uint8_t* apply_padding(uint8_t* data, uint32_t data_size, uint32_t*
padded_size) {
    // Calculate padding length (PKCS#7)
    uint32_t padding_length = AES_BLOCK_SIZE - (data_size % AES_BLOCK_SIZE);
```

```
    if (padding_length == 0) {
        padding_length = AES_BLOCK_SIZE; // If data is already a multiple, add a full
block
    }

    *padded_size = data_size + padding_length;

    printf("Original data size: %u bytes\n", data_size);
    printf("Adding %u bytes of PKCS#7 padding\n", padding_length);
    printf("Padded data size: %u bytes\n", *padded_size);

    // Ensure padded size is a multiple of the hardware block size
    uint32_t remainder = *padded_size % (AES_BLOCK_SIZE * 10); // Hardware processes 10
blocks at once
    if (remainder != 0) {
        uint32_t extra_padding = (AES_BLOCK_SIZE * 10) - remainder;
        printf("Adding %u additional bytes to align with hardware block size\n",
extra_padding);
        *padded_size += extra_padding;
    }

    // Allocate memory for padded data
    uint8_t* padded_data = (uint8_t*)malloc(*padded_size);
    if (!padded_data) {
        printf("Error: Memory allocation failed for padding\n");
        return NULL;
    }

    // Copy original data
    memcpy(padded_data, data, data_size);

    // Add padding bytes (PKCS#7)
    for (uint32_t i = 0; i < padding_length; i++) {
        padded_data[data_size + i] = (uint8_t)padding_length;
    }

    // If we had to add extra padding to align with hardware, fill with zeros
    if (*padded_size > data_size + padding_length) {
        memset(padded_data + data_size + padding_length, 0, *padded_size - (data_size +
padding_length));
    }
```

```
    // Verify final padded size is a multiple of AES block size
    if (*padded_size % AES_BLOCK_SIZE != 0) {
        printf("Error: Padded size %u is not a multiple of AES block size %u\n",
*padded_size, AES_BLOCK_SIZE);
        free(padded_data);
        return NULL;
    }

    return padded_data;
}

// Remove PKCS#7 padding from decrypted data
static uint8_t* remove_padding(uint8_t* data, uint32_t data_size, uint32_t*
unpadded_size) {
    if (data_size == 0 || data_size % AES_BLOCK_SIZE != 0) {
        printf("Error: Invalid padded data size (%u), not a multiple of AES block size
(%u)\n",
               data_size, AES_BLOCK_SIZE);
        return NULL;
    }

    // Get padding length from the last byte
    uint8_t padding_length = data[data_size - 1];

    printf("Detected padding length: %u\n", padding_length);

    // Validate padding
    if (padding_length == 0 || padding_length > AES_BLOCK_SIZE) {
        printf("Warning: Invalid padding detected (value %u), keeping all data\n",
padding_length);
        // Just copy the data as is
        *unpadded_size = data_size;
        uint8_t* unpadded_data = (uint8_t*)malloc(*unpadded_size);
        if (!unpadded_data) {
            printf("Error: Memory allocation failed\n");
            return NULL;
        }
        memcpy(unpadded_data, data, *unpadded_size);
        return unpadded_data;
    }

    // Verify all padding bytes
```

```
    bool valid_padding = true;
    for (uint32_t i = data_size - padding_length; i < data_size; i++) {
        if (data[i] != padding_length) {
            valid_padding = false;
            printf("Invalid padding byte at position %u: expected %u, got %u\n",
                   i, padding_length, data[i]);
            break;
        }
    }

    if (!valid_padding) {
        printf("Warning: Padding verification failed, keeping all data\n");
        *unpadded_size = data_size;
        uint8_t* unpadded_data = (uint8_t*)malloc(*unpadded_size);
        if (!unpadded_data) {
            printf("Error: Memory allocation failed\n");
            return NULL;
        }
        memcpy(unpadded_data, data, *unpadded_size);
        return unpadded_data;
    }

    // Calculate unpadded size
    *unpadded_size = data_size - padding_length;
    printf("After removing %u bytes of padding, data size is %u bytes\n",
           padding_length, *unpadded_size);

    // Allocate memory for unpadded data
    uint8_t* unpadded_data = (uint8_t*)malloc(*unpadded_size);
    if (!unpadded_data) {
        printf("Error: Memory allocation failed\n");
        return NULL;
    }

    // Copy unpadded data
    memcpy(unpadded_data, data, *unpadded_size);

    return unpadded_data;
}

// Process data through the AES hardware
```

```
static int process_data_block(uint8_t* input_data, uint32_t input_size, uint8_t**
output_data, uint32_t* output_size, bool encrypt, int fifo_delay) {
   // Define the hardware block size (160 bytes = 40 32-bit words)
   // AES works on 16-byte blocks, but our hardware processes 10 blocks at once
   #define AES_BLOCK_SIZE_BYTES 16
   #define HARDWARE_BLOCKS_COUNT 10
   #define HARDWARE_BLOCK_SIZE_WORDS (HARDWARE_BLOCKS_COUNT * AES_BLOCK_SIZE_BYTES /
4)
   #define HARDWARE_BLOCK_SIZE_BYTES (HARDWARE_BLOCK_SIZE_WORDS * 4)

   // Ensure input size is a multiple of AES block size
   if (input_size % AES_BLOCK_SIZE_BYTES != 0) {
       printf("Error: Input data size (%d) is not a multiple of AES block size
(%d)\n",
              input_size, AES_BLOCK_SIZE_BYTES);
       return 1;
   }

   printf("Processing %u bytes of data through AES hardware\n", input_size);
   printf("Using hardware block size: %u bytes (%u words)\n",
          HARDWARE_BLOCK_SIZE_BYTES, HARDWARE_BLOCK_SIZE_WORDS);

   // Output will be the same size as input
   *output_size = input_size;
   *output_data = (uint8_t*)malloc(*output_size);
   if (!*output_data) {
       printf("Error: Memory allocation failed\n");
       return 1;
   }

   // Word buffers for processing
   uint32_t word_buffer[HARDWARE_BLOCK_SIZE_WORDS];
   uint32_t output_word_buffer[HARDWARE_BLOCK_SIZE_WORDS];

   // Process data in hardware-sized blocks
   for (uint32_t offset = 0; offset < input_size; offset += HARDWARE_BLOCK_SIZE_BYTES)
{
       // Determine size of current block
       uint32_t block_size = (offset + HARDWARE_BLOCK_SIZE_BYTES <= input_size) ?
                             HARDWARE_BLOCK_SIZE_BYTES : input_size - offset;

       if (offset == 0) {
```

```
        printf("Processing first block, size: %u bytes\n", block_size);
    }

    // Pad the last block with zeros if it's not a full hardware block
    uint32_t word_count = (block_size + 3) / 4; // Round up to handle partial words
    for (uint32_t i = 0; i < HARDWARE_BLOCK_SIZE_WORDS; i++) {
        if (i < word_count) {
            // Handle the case where we might not have a complete word at the end
            if (i == word_count - 1 && block_size % 4 != 0) {
                // Last partial word
                uint32_t value = 0;
                uint32_t bytes_to_copy = block_size % 4;
                memcpy(&value, &input_data[offset + (i * 4)], bytes_to_copy);
                word_buffer[i] = value;
            } else {
                memcpy(&word_buffer[i], &input_data[offset + (i * 4)], 4);
            }
        } else {
            word_buffer[i] = 0;
        }
    }

    // Write words to FIFO
    for (uint32_t i = 0; i < HARDWARE_BLOCK_SIZE_WORDS; i++) {
        int retries = 0;
        while (1) {
            int ret = write_to_fifo(word_buffer[i]);
            if (ret == 0) {
                break;
            } else if (ret == -EBUSY) {
                // FIFO is full, wait and retry
                usleep(fifo_delay);
                retries++;
                if (retries > 100) {
                    printf("Warning: FIFO write retry limit exceeded\n");
                    retries = 0;
                }
            } else {
                // Other error
                printf("Error writing to FIFO\n");
                free(*output_data);
                *output_data = NULL;
```

```
                return 1;
            }
        }
    }

    // Read processed words from FIFO
    for (uint32_t i = 0; i < HARDWARE_BLOCK_SIZE_WORDS; i++) {
        int retries = 0;
        while (1) {
            int ret = read_from_fifo(&output_word_buffer[i]);
            if (ret == 0) {
                break;
            } else if (ret == -EAGAIN) {
                // FIFO is empty, wait and retry
                usleep(fifo_delay);
                retries++;
                if (retries > 100) {
                    printf("Warning: FIFO read retry limit exceeded\n");
                    retries = 0;
                }
            } else {
                // Other error
                printf("Error reading from FIFO\n");
                free(*output_data);
                *output_data = NULL;
                return 1;
            }
        }
    }

    // Copy processed data to output buffer
    for (uint32_t i = 0; i < word_count && (offset + i * 4) < *output_size; i++) {
        // Handle the case where we might not have a complete word at the end
        if (i == word_count - 1 && block_size % 4 != 0) {
            // Last partial word
            uint32_t bytes_to_copy = block_size % 4;
            memcpy(*output_data + offset + (i * 4), &output_word_buffer[i],
bytes_to_copy);
        } else if (offset + (i * 4) + 4 <= *output_size) {
            memcpy(*output_data + offset + (i * 4), &output_word_buffer[i], 4);
        } else {
            // We're at the end of the buffer and don't have room for a full word
```

```
                uint32_t bytes_to_copy = *output_size - (offset + (i * 4));
                memcpy(*output_data + offset + (i * 4), &output_word_buffer[i],
bytes_to_copy);
            }
        }

        // Print progress
        uint32_t processed = offset + block_size;
        if (processed % (1024 * 1024) < HARDWARE_BLOCK_SIZE_BYTES) {
            printf("\rProcessed: %.2f MB", processed / (1024.0 * 1024.0));
            fflush(stdout);
        }
    }

    printf("\rProcessed: %.2f MB\n", input_size / (1024.0 * 1024.0));
    return 0;
}

// Process a WAV file (encrypt or decrypt)
static int process_wav_file(program_options *options) {
    FILE *infile = NULL, *outfile = NULL;
    WAVHeader header;
    uint8_t *audio_data = NULL;
    uint8_t *processed_data = NULL;
    uint8_t *final_data = NULL;
    uint32_t audio_data_size, processed_data_size, final_data_size;
    int ret = 1;  // Default to error

    printf("\n=====================================================\n");
    printf("Starting WAV file processing\n");
    printf("Mode: %s\n", options->encrypt ? "ENCRYPT" : "DECRYPT");
    printf("Input: %s\n", options->input_file);
    printf("Output: %s\n", options->output_file);
    printf("=====================================================\n\n");

    // Open input file
    infile = fopen(options->input_file, "rb");
    if (!infile) {
        perror("Error opening input file");
        return 1;
    }
```

```
// Read and validate WAV header
if (read_wav_header(infile, &header) != 0) {
    fclose(infile);
    return 1;
}

// Read the audio data
audio_data_size = header.data_size;
audio_data = (uint8_t*)malloc(audio_data_size);
if (!audio_data) {
    printf("Error: Memory allocation failed for audio data\n");
    fclose(infile);
    return 1;
}

size_t bytes_read = fread(audio_data, 1, audio_data_size, infile);
if (bytes_read != audio_data_size) {
    printf("Error: Failed to read audio data. Expected %u bytes, got %zu bytes\n",
           audio_data_size, bytes_read);
    free(audio_data);
    fclose(infile);
    return 1;
}

printf("Successfully read %u bytes of audio data\n", audio_data_size);

// Close input file
fclose(infile);
infile = NULL;

// Process based on mode
if (options->encrypt) {
    // Apply padding for encryption
    uint8_t *padded_data;
    uint32_t padded_size;

    printf("\n--- Encryption Process ---\n");
    printf("Applying PKCS#7 padding for encryption...\n");
    padded_data = apply_padding(audio_data, audio_data_size, &padded_size);
    if (!padded_data) {
        free(audio_data);
        return 1;
```

```
        }

        printf("Encrypting audio data (%d bytes with padding)...\n", padded_size);
        if (process_data_block(padded_data, padded_size, &processed_data,
&processed_data_size, true, options->fifo_delay) != 0) {
            free(audio_data);
            free(padded_data);
            return 1;
        }

        // Use the processed data as final data
        final_data = processed_data;
        final_data_size = processed_data_size;

        free(padded_data);

        printf("Encryption completed successfully\n");
    } else {  // Decrypt
        printf("\n--- Decryption Process ---\n");
        printf("Decrypting audio data (%d bytes)...\n", audio_data_size);

        // Verify data size is valid for AES processing (multiple of block size)
        if (audio_data_size % AES_BLOCK_SIZE != 0) {
            printf("Error: Input data size for decryption (%u bytes) is not a multiple
of AES block size (%d).\n",
                   audio_data_size, AES_BLOCK_SIZE);
            printf("This suggests the file was not properly encrypted or is
corrupted.\n");
            free(audio_data);
            return 1;
        }

        if (process_data_block(audio_data, audio_data_size, &processed_data,
&processed_data_size, false, options->fifo_delay) != 0) {
            free(audio_data);
            return 1;
        }

        // Remove padding from decrypted data
        printf("Removing PKCS#7 padding...\n");
        final_data = remove_padding(processed_data, processed_data_size,
&final_data_size);
```

```
        if (!final_data) {
            free(audio_data);
            free(processed_data);
            return 1;
        }

        printf("Decryption completed successfully\n");
    }

    // Update WAV header with new data size
    printf("\n--- Updating WAV Header ---\n");

    WAVHeader original_header = header; // Save original for comparison

    header.data_size = final_data_size;
    header.wav_size = 36 + final_data_size;  // Total size - 8 bytes for RIFF header

    // Also update byte_rate and block_align which depend on data size
    // byte_rate = SampleRate * NumChannels * BitsPerSample/8
    header.byte_rate = header.sample_rate * header.num_channels *
(header.bits_per_sample / 8);
    // block_align = NumChannels * BitsPerSample/8
    header.block_align = header.num_channels * (header.bits_per_sample / 8);

    printf("Original WAV header values:\n");
    printf("  format: %u (1=PCM)\n", original_header.audio_format);
    printf("  channels: %u\n", original_header.num_channels);
    printf("  sample_rate: %u Hz\n", original_header.sample_rate);
    printf("  bits_per_sample: %u\n", original_header.bits_per_sample);
    printf("  data_size: %u bytes\n", original_header.data_size);

    printf("\nUpdated WAV header values for output file:\n");
    printf("  format: %u (unchanged)\n", header.audio_format);
    printf("  channels: %u (unchanged)\n", header.num_channels);
    printf("  sample_rate: %u Hz (unchanged)\n", header.sample_rate);
    printf("  bits_per_sample: %u (unchanged)\n", header.bits_per_sample);
    printf("  data_size: %u bytes (%s%d bytes)\n",
           header.data_size,
           header.data_size > original_header.data_size ? "+" : "",
           (int)header.data_size - (int)original_header.data_size);

    // Open output file
```

```
    printf("\n--- Writing Output File ---\n");
    outfile = fopen(options->output_file, "wb");
    if (!outfile) {
        perror("Error opening output file");
        free(audio_data);
        free(processed_data);
        if (final_data != processed_data) {
            free(final_data);
        }
        return 1;
    }

    // Write WAV header
    printf("Writing WAV header to output file...\n");
    if (fwrite(header.riff_header, 1, 4, outfile) != 4 ||
        fwrite(&header.wav_size, 4, 1, outfile) != 1 ||
        fwrite(header.wave_header, 1, 4, outfile) != 4 ||
        fwrite(header.fmt_header, 1, 4, outfile) != 4 ||
        fwrite(&header.fmt_chunk_size, 4, 1, outfile) != 1 ||
        fwrite(&header.audio_format, 2, 1, outfile) != 1 ||
        fwrite(&header.num_channels, 2, 1, outfile) != 1 ||
        fwrite(&header.sample_rate, 4, 1, outfile) != 1 ||
        fwrite(&header.byte_rate, 4, 1, outfile) != 1 ||
        fwrite(&header.block_align, 2, 1, outfile) != 1 ||
        fwrite(&header.bits_per_sample, 2, 1, outfile) != 1 ||
        fwrite(header.data_header, 1, 4, outfile) != 4 ||
        fwrite(&header.data_size, 4, 1, outfile) != 1) {
        printf("Error: Failed to write WAV header\n");
        goto cleanup;
    }

    // Write processed audio data
    printf("Writing %u bytes of audio data...\n", final_data_size);
    if (fwrite(final_data, 1, final_data_size, outfile) != final_data_size) {
        printf("Error: Failed to write processed audio data\n");
        goto cleanup;
    }

    // Verify output file
    fflush(outfile);
    fclose(outfile);
    outfile = NULL;
```

```
    printf("Verifying output file...\n");
    outfile = fopen(options->output_file, "rb");
    if (!outfile) {
        printf("Error: Could not open output file for verification\n");
        goto cleanup;
    }

    // Read header of output file to verify
    WAVHeader check_header;
    if (read_wav_header(outfile, &check_header) != 0) {
        printf("Error: Output file header validation failed\n");
        goto cleanup;
    }

    // Check data size matches
    if (check_header.data_size != final_data_size) {
        printf("Warning: Output file data size mismatch. Expected %u, got %u\n",
               final_data_size, check_header.data_size);
    } else {
        printf("Output file validated successfully\n");
    }

    printf("\nSuccessfully wrote %d bytes of %s audio data\n",
           final_data_size, options->encrypt ? "encrypted" : "decrypted");
    ret = 0;  // Success

cleanup:
    if (infile) fclose(infile);
    if (outfile) fclose(outfile);
    if (audio_data) free(audio_data);
    if (processed_data) free(processed_data);
    if (final_data && final_data != processed_data) free(final_data);

    if (ret == 0) {
        printf("\n=======================================================\n");
        printf("WAV processing completed successfully\n");
        printf("Output file: %s\n", options->output_file);
        printf("=======================================================\n");
    } else {
        printf("\n=======================================================\n");
        printf("WAV processing failed!\n");
```

```
        printf("=====================================================\n");
    }

    return ret;
}
```

load_round_keys.c

```
/**
* Load AES Round Keys into true_mixed_dual_port_ram using the kernel driver
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdint.h>
#include <errno.h>
#include "drivers/round_keys.h"

void print_usage(const char* progname) {
    printf("Usage: %s <key_hex>\n", progname);
    printf("  <key_hex>: 32 hex characters representing the 16-byte initial key\n");
    printf("Example: %s 2B7E151628AED2A6ABF7158809CF4F3C\n", progname);
}

// AES S-box
static const uint8_t SBOX[256] = {
    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
};

// Round constants for key schedule
static const uint8_t RCON[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
0x36};

// Key schedule functions
static void sub_word(uint8_t* word) {
    for (int i = 0; i < 4; i++) {
        word[i] = SBOX[word[i]];
    }
}

static void rot_word(uint8_t* word) {
    uint8_t temp = word[0];
    word[0] = word[1];
    word[1] = word[2];
    word[2] = word[3];
    word[3] = temp;
}

static void key_schedule_core(uint8_t* word, int iteration) {
    rot_word(word);
    sub_word(word);
```

```
    word[0] ^= RCON[iteration];
}

void expand_key(uint8_t* key, uint8_t round_keys[11][16]) {
    // Copy the initial key to round 0
    memcpy(round_keys[0], key, 16);

    // Convert key into 32-bit words (4 words total for AES-128)
    uint32_t w[44]; // 11 rounds * 4 words per round

    // Initialize first four words with the initial key
    for (int i = 0; i < 4; i++) {
        w[i] = (key[4*i] << 24) | (key[4*i+1] << 16) | (key[4*i+2] << 8) | key[4*i+3];
    }

    // Expand the key
    for (int i = 4; i < 44; i++) {
        // Extract the previous word as bytes
        uint8_t temp[4];
        temp[0] = (w[i-1] >> 24) & 0xFF;
        temp[1] = (w[i-1] >> 16) & 0xFF;
        temp[2] = (w[i-1] >> 8) & 0xFF;
        temp[3] = w[i-1] & 0xFF;

        if (i % 4 == 0) {
            // Apply key schedule core for the first word of each round
            key_schedule_core(temp, (i/4)-1);
        }

        // Convert bytes back to word
        uint32_t temp_word = (temp[0] << 24) | (temp[1] << 16) | (temp[2] << 8) |
temp[3];

        // XOR with word 4 positions back
        w[i] = w[i-4] ^ temp_word;
    }

    // Convert words back to bytes in the round keys
    for (int round = 0; round < 11; round++) {
        for (int word = 0; word < 4; word++) {
            int idx = round * 4 + word;
            round_keys[round][word*4]   = (w[idx] >> 24) & 0xFF;
```

```
            round_keys[round][word*4+1] = (w[idx] >> 16) & 0xFF;
            round_keys[round][word*4+2] = (w[idx] >> 8) & 0xFF;
            round_keys[round][word*4+3] = w[idx] & 0xFF;
        }
    }
}

// Function to convert hex string to bytes
int hex_to_bytes(const char* hex_str, uint8_t* bytes, size_t bytes_len) {
    size_t hex_len = strlen(hex_str);

    // Check if hex string has the right length
    if (hex_len != bytes_len * 2) {
        return 0;
    }

    // Convert each pair of hex characters to a byte
    for (size_t i = 0; i < bytes_len; i++) {
        char byte_str[3] = {hex_str[i*2], hex_str[i*2+1], '\0'};
        char* end_ptr;
        bytes[i] = (uint8_t)strtol(byte_str, &end_ptr, 16);

        // Check if conversion was successful
        if (*end_ptr != '\0') {
            return 0;
        }
    }

    return 1;
}

// Print round keys (for debugging)
void print_round_keys(uint8_t round_keys[11][16]) {
    for (int i = 0; i < 11; i++) {
        printf("Round %2d: ", i);
        for (int j = 0; j < 16; j++) {
            printf("%02x", round_keys[i][j]);
            if (j % 4 == 3 && j < 15) printf(" ");
        }
        printf("\n");
    }
}
```

```c
int main(int argc, char* argv[]) {
    int fd;
    uint8_t initial_key[16];
    uint8_t round_keys[11][16];
    round_keys_data_t rk_data;

    if (argc != 2) {
        print_usage(argv[0]);
        return 1;
    }

    // Parse the key
    if (!hex_to_bytes(argv[1], initial_key, 16)) {
        printf("Error: Invalid key format. Must be 32 hex characters.\n");
        return 1;
    }

    // Generate all round keys
    expand_key(initial_key, round_keys);

    // Print the round keys (for debugging)
    printf("Generated AES-128 Round Keys:\n");
    print_round_keys(round_keys);
    printf("Using little endian, reverse word order (mode 4)\n");

    // Open the round keys device
    fd = open("/dev/round_keys", O_RDWR);
    if (fd < 0) {
        perror("Error opening /dev/round_keys");
        return 1;
    }

    printf("Successfully opened /dev/round_keys\n");

    // Copy round keys to our data structure
    for (int round = 0; round < 11; round++) {
        for (int byte = 0; byte < 16; byte++) {
            rk_data.data[round * 16 + byte] = round_keys[round][byte];
        }
    }
```

```c
// Write round keys to the device
printf("Writing round keys to RAM...\n");
if (ioctl(fd, ROUND_KEYS_WRITE_DATA, &rk_data) < 0) {
    perror("Error writing round keys");
    close(fd);
    return 1;
}

printf("Successfully wrote 11 round keys (176 bytes) to RAM\n");

// Read back for verification
printf("\nVerifying written values:\n");

// Clear our data structure before reading
memset(&rk_data, 0, sizeof(rk_data));

// Read back the round keys
if (ioctl(fd, ROUND_KEYS_READ_DATA, &rk_data) < 0) {
    perror("Error reading round keys");
    close(fd);
    return 1;
}

// Verify the data
int errors = 0;
for (int round = 0; round < 11; round++) {
    for (int byte = 0; byte < 16; byte++) {
        int offset = round * 16 + byte;
        uint8_t expected = round_keys[round][byte];
        uint8_t actual = rk_data.data[offset];

        if (actual != expected) {
            printf("Mismatch at round %d, byte %d: expected 0x%02x, got 0x%02x\n",
                   round, byte, expected, actual);
            errors++;
        }
    }
}

if (errors == 0) {
    printf("Verification successful! All round keys written correctly.\n");
} else {
```

```
        printf("Verification failed with %d errors.\n", errors);
    }

    // Clean up
    close(fd);

    return 0;
}
```

Makefile

```
# Makefile for AES WAV processing project
CC := gcc
CFLAGS := -Wall -Werror -O2
LDFLAGS := -lm

# Userspace programs
USERSPACE_PROGS := aes_wav load_round_keys

# Kernel module directories
KERNEL_MODULES := drivers

# Default target
all: userspace modules

# Build userspace programs
userspace: $(USERSPACE_PROGS)
    @echo "Userspace programs built successfully"

aes_wav: aes_wav.c
    @echo "Building aes_wav..."
    $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)

load_round_keys: load_round_keys.c
    @echo "Building load_round_keys..."
    $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)

# Build kernel modules
modules:
    @echo "Building kernel modules..."
    $(MAKE) -C $(KERNEL_MODULES)

# Clean everything
```

```
clean:
   @echo "Cleaning userspace programs..."
   rm -f $(USERSPACE_PROGS)
   @echo "Cleaning kernel modules..."
   $(MAKE) -C $(KERNEL_MODULES) clean

# Install kernel modules
install:
   @echo "Installing kernel modules..."
   $(MAKE) -C $(KERNEL_MODULES) install

# Uninstall kernel modules
uninstall:
   @echo "Uninstalling kernel modules..."
   $(MAKE) -C $(KERNEL_MODULES) uninstall

# Restart: uninstall then install
restart: uninstall install
   @echo "Kernel modules restarted"

# Run full test
test: all
   @echo "Running full test suite..."
   # Add your test commands here

.PHONY: all userspace modules clean install uninstall restart test
```

Python Code

aes_ecb.py

```
import os
import wave
import argparse
import struct
import base64
import time

# 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
]

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

# Galois Field multiplication
def gmul(a, b):
    p = 0
    for _ in range(8):
        if b & 1:            # If the lowest bit of b is set
            p ^= a           # XOR the current value of a into the product
        high_bit_set = a & 0x80  # Check if the highest bit (x^7) is set
        a <<= 1              # Double a (equivalent to multiplying by x)
        if high_bit_set:     # If we would overflow to x^8
            a ^= 0x1b        # XOR with 0x1b (the irreducible polynomial without x^8)
        b >>= 1              # Shift b right to check the next bit
    return p & 0xff          # Ensure the result fits in a byte
```

```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

# 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

# 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

# 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

# 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

# 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

# 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

# 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 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)

# 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)

# 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

# Validate and convert a hexadecimal key string to bytes
def validate_and_convert_key(key_hex):
    if len(key_hex) != 32:
        raise ValueError(f"AES-128 requires a 32-character hex key. Got {len(key_hex)}
characters instead.")

    try:
        # Convert hex string to bytes
        key_bytes = bytes.fromhex(key_hex)
        return key_bytes
    except ValueError:
        raise ValueError("Invalid hexadecimal key. Key must contain only hexadecimal
characters (0-9, a-f).")

# Function to encrypt a WAV file using ECB mode
def encrypt_wav(input_file, output_file, key_hex):
    total_start_time = time.time()

    # Validate and convert the hex key
    key = validate_and_convert_key(key_hex)

    # Read the WAV file
    with wave.open(input_file, 'rb') as wav_file:
        params = wav_file.getparams()
        frames = wav_file.readframes(wav_file.getnframes())

    # Encrypt the audio data using ECB mode
    encryption_start_time = time.time()
    encrypted_data = encrypt_ecb(frames, key)
    encryption_end_time = time.time()

    # Create a new WAV file with the same parameters but encrypted audio data
    with wave.open(output_file, 'wb') as wav_file:
        wav_file.setparams(params)
        wav_file.writeframes(encrypted_data)

    total_end_time = time.time()
```

```
    # Calculate and print timing information
    encryption_time = encryption_end_time - encryption_start_time
    total_time = total_end_time - total_start_time

    print(f"File encrypted successfully: {output_file}")
    print(f"Encryption time: {encryption_time:.4f} seconds")
    print(f"Total processing time (including I/O): {total_time:.4f} seconds")
    print(f"File size: {len(frames) / 1024:.2f} KB")
    print(f"Encryption speed: {(len(frames) / 1024) / encryption_time:.2f} KB/s")

# Function to decrypt a WAV file
def decrypt_wav(input_file, output_file, key_hex):
    total_start_time = time.time()

    # Validate and convert the hex key
    key = validate_and_convert_key(key_hex)

    # Read the encrypted WAV file
    with wave.open(input_file, 'rb') as wav_file:
        params = wav_file.getparams()
        encrypted_data = wav_file.readframes(wav_file.getnframes())

    # Decrypt the audio data using ECB mode
    decryption_start_time = time.time()
    decrypted_data = decrypt_ecb(encrypted_data, key)
    decryption_end_time = time.time()

    # Create a new WAV file with the original parameters and decrypted audio data
    with wave.open(output_file, 'wb') as wav_file:
        wav_file.setparams(params)
        wav_file.writeframes(decrypted_data)

    total_end_time = time.time()

    # Calculate and print timing information
    decryption_time = decryption_end_time - decryption_start_time
    total_time = total_end_time - total_start_time

    print(f"File decrypted successfully: {output_file}")
    print(f"Decryption time: {decryption_time:.4f} seconds")
    print(f"Total processing time (including I/O): {total_time:.4f} seconds")
```

```
    print(f"File size: {len(encrypted_data) / 1024:.2f} KB")
    print(f"Decryption speed: {(len(encrypted_data) / 1024) / decryption_time:.2f}
KB/s")

def main():
    parser = argparse.ArgumentParser(description='Encrypt or decrypt WAV files using
AES-128 in ECB mode from scratch.')
    parser.add_argument('action', choices=['encrypt', 'decrypt'], help='Action to
perform')
    parser.add_argument('input_file', help='Path to the input file')
    parser.add_argument('output_file', help='Path to the output file')
    parser.add_argument('key', help='Hexadecimal key (32 characters / 16 bytes)')

    args = parser.parse_args()

    if args.action == 'encrypt':
        encrypt_wav(args.input_file, args.output_file, args.key)
    else:
        decrypt_wav(args.input_file, args.output_file, args.key)

if __name__ == '__main__':
    main()
```

keygen.py

```
"""
Simple AES-128 Key Generator

This script generates a random 16-byte (128-bit) key suitable for AES-128 encryption
and writes it to a file.
"""

import os
import sys

# Generate a random 16-byte key
key = os.urandom(16)

# Print key in hexadecimal format
print(f"Generated AES-128 key (hex): {key.hex()}")

# Write key to file if filename provided
```

```
if len(sys.argv) > 1:
   filename = sys.argv[1]
   with open(filename, 'wb') as f:
       f.write(key)
   print(f"Key saved to {filename}")
else:
   print("No filename provided. Key was not saved.")
```