ASSIGNMENT - 2

BY GROUP 4
AKSHAY K MURALEEDHARAN – 200088
KARTIKEYAN IYER – 200495
MOHIL – 200596
HRISHANT TRIPATHI - 210449

Q. Design a Verilog module for AES-128 iterative architecture. The design should have two inputs: plaintext (128-bit), key (128-bit), and one output: ciphertext (128-bit). The key expansion utilized to generate the round keys using the master key and the AES round operation should run parallelly in a non-blocking manner. You can use the instantiation of given modules (subbyte, shiftrow, mixcolumn, sbox and key expansion) to design a 10-round iterative AES hardware.

Methodology report for “aes_top” module:

Design Approach:

The key steps of the design approach are as follows:
1. Input Signals: The module takes several input signals, including clk (clock), reset, plaintext (the input data to be encrypted), key (the encryption key), and start (a control signal to initiate the encryption process; should be on only for one clock cycle).
2. State Machine: The module uses a state machine to control the AES encryption process. The state machine transitions through different states to manage key expansion, encryption rounds, and completion.
3. Key Expansion: The aes_key_expand_128 module is instantiated within the "aes_top" module to perform key expansion. It generates the round keys used in the encryption process.
4. Round Key Selection: A multiplexer (multiplexer_16_to_1) selects the appropriate round key based on the current state of the state machine.
5. Encryption Rounds: The module performs AES encryption rounds based on the selected round key and state. The plaintext data is XORed with the round key in each round.

6. Output Generation: The ciphertext is generated as the final encrypted result. Additionally, the module sets the done signal to indicate the completion of the encryption process.

Key Components:

1. State Machine: The state machine manages the control flow of the AES encryption process. It transitions through different states, including key expansion, encryption rounds, and completion states.
2. Key Expansion (aes_key_expand_128 Module) The aes_key_expand_128 module is responsible for expanding the input encryption key into a set of round keys used in the encryption process.
3. Round Key Selection (Multiplexer): A multiplexer (multiplexer_16_to_1) selects the appropriate round key based on the current state of the state machine.
4. Encryption Rounds: The module performs AES encryption rounds based on the selected round key and state. This involves XOR-ing the output of shiftrows or mixcolumns operation with the current round key.
5. Output Generation: The ciphertext is generated as the final encrypted result, and the done signal is set to indicate the completion of the encryption process.

Implementation Details:

The module takes input signals such as clk, reset, plaintext, key, and start, as well as output signals for ciphertext and done.
A state machine is implemented using an always block to control the flow of the encryption process. It transitions between states and manages the done signal accordingly.
The aes_key_expand_128 module is instantiated within the "aes_top" module to perform key expansion. The generated round keys are stored in the key_exp wire array.
The selected round key (curr_rd_key) is chosen by a multiplexer based on the current state of the state machine.
AES encryption rounds are performed based on the selected round key and state. XOR operations between either the shiftrow or mixcolumns output (depending on the round) and the round key are used to generate the ciphertext.
The done signal is set for one clock cycle to indicate the completion of the encryption process, and it is managed by the state machine.

Methodology report for “aes_key_expansion_128” module:

Design Approach:

The primary steps of the key expansion process are as follows:
1. Key Input: The module takes a 128-bit encryption key and a clock signal as input.
2. Word Splitting: The 128-bit key is divided into four 32-bit words, denoted as w0, w1, w2, and w3.
3. Round Constants Calculation: The round constants (rcon, rcon2, ..., rcon10) are calculated based on the AES Rijndael key expansion algorithm. These round constants are used in the key expansion process.
4. Subword Calculation: The subword values (subword, subword2, ..., subword10) are computed by applying the AES S-Box substitution to certain parts of the input key words.
5. Word Expansion: The key expansion process involves multiple rounds of XOR operations between the previously calculated words and the subword values, along with the round constants.
6. Round Key Output: The expanded round keys (key_s0, key_s1, ..., key_s10) are formed by concatenating the resulting words.
7. S-Box Substitution: The module utilizes an S-Box module (sbox) to perform byte-wise substitution in the key expansion process.
8. RCON Calculation: The aes_rcon module is used to calculate the round constants (rcon, rcon2, ..., rcon10) based on the current round number.

Key Components

1. Key Splitting: w0, w1, w2, w3: 32-bit registers to hold the four 32-bit words extracted from the 128-bit key.
2. Round Constants Calculation (aes_rcon Module): rcon, rcon2, ..., rcon10: 8-bit constants representing the round constants for each round of key expansion.
3. Subword Calculation: subword, subword2, ..., subword10: 32-bit wires representing the results of the S-Box substitution applied to specific parts of the key words.
4. Word Expansion: Multiple registers (w4, w5, ..., w43) are used to compute the expanded key words through a series of XOR operations involving the subword values, round constants, and previously calculated words.
5. S-Box Substitution (sbox Modules): Multiple instances of the sbox module are used to perform byte-wise S-Box substitution on different parts of the key words.

Implementation Details:

The key expansion process is divided into 11 rounds, with each round generating a new round key (key_s0, key_s1, ..., key_s10).
The aes_rcon module calculates the round constants based on the current round number, which is used in the word expansion process.
The S-Box substitution is performed using separate instances of the sbox module for different parts of the key words.
The module uses a combination of wire and reg data types to store intermediate values and registers for holding the final round keys.
The resulting round keys (key_s0, key_s1, ..., key_s10) are constructed by concatenating the expanded words.
The entire key expansion process is driven by the clock signal.

Methodology report for “multiplexer_16_to_1”:

Design Approach:

The design approach follows these key steps:

1. Input Signals: The module takes sixteen 128-bit data inputs (A0 to A15), a 4-bit selection signal (*sel*), and provides a 128-bit output signal (B).
2. Selection Logic: The module uses a 4-bit selection signal to determine which input should be routed to the output. The selection signal sel specifies the index of the selected input.
3. Output Assignment: The selected input is assigned to the output B. When the value of sel corresponds to one of the input indexes, the data from that input is routed to B.
4. Default Handling: If the value of sel does not correspond to any of the input indexes (e.g., it is out of range), a default assignment of 128'h0 (a 128-bit zero value) is made to B to ensure predictable behavior.

Key Components:

1. Input Data Sources: Sixteen input data sources (A0 to A15) provide the 128-bit data inputs that can be selected by the multiplexer.
2. Selection Signal (sel): The 4-bit selection signal sel specifies which input data source should be routed to the output B.

3. Output Data (B): The output B is a 128-bit data signal that carries the data selected by the multiplexer based on the value of the selection signal.

Implementation Details:

The module utilizes an always @* block, which is sensitive to changes in the sel signal, to determine the selected input.
A case statement is used to evaluate the value of sel and make the appropriate assignment to the output B. Each case corresponds to one of the sixteen input sources (A0 to A15).
The assignment to B within the case statement is based on the value of sel. If a valid selection is made, the data from the corresponding input is assigned to B. If sel is out of range or invalid, a default assignment of 128'h0 is made to B to ensure a well-defined output.

Methodology report for “sbox” module:

Design Approach:

The design approach follows these key steps:
1. Input and Output: The module takes an 8-bit input signal data and provides an 8-bit output signal dout.
2. Substitution Logic: The module uses a case statement to evaluate the value of the data input and map it to a corresponding value from the substitution table. Each case corresponds to a specific value of data.
3. Substitution Table: The substitution table contains 256 entries, each with an 8-bit output value corresponding to an 8-bit input value. The module uses this table to determine the output for a given input.
4. Default Handling: If the input value data does not match any of the predefined cases, a default assignment of 8'h00 (an 8-bit zero value) is made to the output dout to ensure predictable behavior.

Key Components:

1. Input Data (data): The 8-bit input data signal data is provided to the module for substitution.
2. Output Data (dout): The 8-bit output data signal dout carries the result of the substitution operation.

3. Substitution Table: The substitution table contains 256 entries, each specifying an 8-bit output value for a corresponding 8-bit input value. This table defines the substitution logic of the S-Box.

Implementation Details:

The module uses an always @(data) block, which is sensitive to changes in the data signal, to perform the substitution operation.
A case statement is used to evaluate the value of the data input. Each case corresponds to one of the 256 possible input values (ranging from 8'h00 to 8'hFF). Within the case statement, the module assigns the corresponding output value from the substitution table to the dout output signal. This effectively substitutes the input data with the appropriate value from the table.
If the input value data does not match any of the predefined cases, the default case is triggered, and the output dout is assigned the value 8'h00 to ensure well-defined behavior.

Methodology report for “subbyte” module:

Design Approach:

The design approach follows these key steps:
1. Input and Output: The module takes a 128-bit input signal data and provides a 128-bit output signal s_data_out.
2. Substitution Logic: The module subdivides the 128-bit input data into sixteen 8-bit segments (bytes) and applies the SubBytes operation to each of these bytes independently.
3. S-Box: For each 8-bit input byte, the module instantiates an "S-Box" module (referred to as sbox) to perform the substitution. The S-Box is a lookup table that provides the substitution values for all possible 8-bit inputs.
4. Temporary Storage: The module uses a temporary 128-bit wire signal tmp_out to hold the intermediate results of the SubBytes operation for each byte.
5. Combinational Process: This is a part of reusable hardware connected by *wires* hence is a combinational circuit.

Key Components:

1. Input Data (data): The 128-bit input data signal data is provided to the module for the SubBytes operation.

2. Output Data (s_data_out): The 128-bit output data signal s_data_out carries the result of the SubBytes operation. Each byte of the output corresponds to the substitution of the corresponding byte from the input data based on the AES S-Box.
3. S-Box (sbox): The "S-Box" module (sbox) is instantiated sixteen times to perform the byte-wise SubBytes operation. Each instantiation takes an 8-bit input and provides an 8-bit output based on the AES S-Box.
4. Temporary Storage (tmp_out): The 128-bit wire signal tmp_out temporarily holds the intermediate results of the SubBytes operation for each byte before they are assigned to the output s_data_out.

Implementation Details:

The module was changed to become a combinational block to perform the SubBytes operation.
The SubBytes operation for each byte is carried out by instantiating the "S-Box" module (sbox). The input bytes from data are divided into sixteen 8-bit segments, and each is passed to a separate instantiation of sbox.
The output of each sbox instantiation is stored in the tmp_out wire signal, effectively substituting each byte in the input data with the corresponding S-Box value.
The resulting 128-bit tmp_out value is assigned to the output s_data_out, producing the final output of the SubBytes operation.

Methodology report for “mixcolumn” module:

Design Approach:

The design approach follows these key steps:
1. Input and Output: The module takes a 128-bit input signal data_in and provides a 128-bit output signal data_out.
2. Data Segmentation: The input data is segmented into four 32-bit columns (referred to as n1, n2, n3, and n4).
3. Matrix Multiplication: Each column is individually multiplied with a specific matrix to perform the MixColumns operation.
4. Multiplication Modules: For each column, multiplication by 2, 3, or a mix of both is performed. These multiplications are carried out by instantiating two types of multiplication modules: mul_2 and mul_3.

5. Temporary Storage: Temporary 8-bit wire signals are used to hold the intermediate results of the multiplications (tmp1, tmp2, tmp3, tmp4) as well as the results of the mix columns operation for each column (ma0, ma1, ma2, ma3).
6. Output Formation: The results of the mix columns operation are combined to form the 128-bit output data_out.

Key Components:

1. Input Data (data_in): The 128-bit input data signal data_in is provided to the module for the MixColumns operation.
2. Output Data (data_out): The 128-bit output data signal data_out carries the result of the MixColumns operation.
3. Multiplication Modules (mul_2 and mul_3): The multiplication modules (mul_2 and mul_3) are instantiated to perform multiplications by 2 and 3, respectively.
4. Temporary Storage (tmp1, tmp2, tmp3, tmp4, ma0, ma1, ma2, ma3): These 8-bit wire signals temporarily hold the intermediate results of multiplications and the results of the mix columns operation for each column.
5. Column Segmentation (n1, n2, n3, n4): These 32-bit wire signals hold each of the four columns of the input data data_in.

Implementation Details:

The module is a combinational block after some mods which was done in order to mitigate the problem of taking a whole cycle to execute and then passing on the output to subsequent blocks.
The 128-bit input data_in is segmented into four 32-bit columns, namely n1, n2, n3, and n4.
For each column, matrix multiplication is performed using the mul_2 and mul_3 modules. The results are temporarily stored in tmp1, tmp2, tmp3, and tmp4.
The results of matrix multiplication for each column are XORed to form the columns of the output (ma0, ma1, ma2, ma3).
Finally, the four columns of the output are combined to form the 128-bit data_out, which is the result of the MixColumns operation.

Methodology report for “shiftrows” module:

Design Approach:

The design approach involves the following key steps:
1. Input and Output: The module takes a 128-bit input signal data_in and provides a 128-bit output signal data_out.
2. Byte Shifting: The input data block is divided into 16 bytes (from data_in[127:0] to data_in[7:0]). Each of these bytes is shifted within its respective row.
3. Row Shifting: Byte shifting is performed row-wise to achieve the ShiftRows operation:
Row 0 shifted by one step to the left.
Row 1 is shifted two steps to the left.
Row 2 is shifted three steps to the left.
Row 3 remains unchanged.
4. Combinational Process: The shifting operation is performed using assign statements. This ensures that the operation is synchronized with the clock and operates in a deterministic manner in top module.
5. Output Formation: The shifted bytes are combined to form the 128-bit output data_out.

Key Components:

1. Input Data (data_in): The 128-bit input data signal data_in is provided to the module for the ShiftRows operation.
2. Output Data (data_out): The 128-bit output data signal data_out carries the result of the ShiftRows operation.

Implementation Details:

The input data block data_in is divided into 16 bytes, where each byte is indexed from data_in[127:0] to data_in[7:0].
Byte shifting is performed within each row to implement the ShiftRows operation.
The specific shifts are as follows:
Row 0 shifted by one step to the left.
Row 1 is shifted two steps to the left.
Row 2 is shifted three steps to the left.
Row 3 remains unchanged.
After shifting the bytes within their respective rows, they are combined to form the 128-bit output data_out.