COL215 Hardware Assignment 3 Part II

Submission Deadlines: 10th November 2024

1 Introduction

The objective of the assignment is: **implementation of AES decryption operation**. In the part I, you have created subcomponents for the individual operations involved in the decryption. In the current assignment, you will create controller/FSM to connect and control each component. On high level, the FSM should perform following tasks:

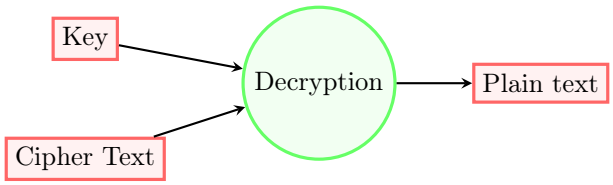
- Control read/write from memories to the ports of compute units.
- Execution of each round described in Figure 3. You should implement for 10 rounds (initial + 1 $\rightarrow$ 8 + final round).

Please note that the diagrams in this document are for illustration purposes and explanation of the problem statement. They do not represent the exact design to be implemented. Design choices can vary across groups and should be carefully explained in the assignment report.

2 Problem Description

Given a cipher text and a key, your task is to perform an AES decryption operation to display plain text on 7-seven display. An overview of the problem is shown in Figure 1.

Figure 1: Overview of task: Decryption



- Input cipher text is of the size in multiples of 128 bits and will be provided in the COE file (8 bit binary unsigned). The input file should be stored in 1-D array format in the block RAM, starting from address 0 (0000_{16}) in row major format.
- Key will be provided via the COE file. All the round keys will be provided in the COE file.

You are allowed to use VHDL modules from previous hardware assignments.

3 Basics of AES Encryption/Decryption

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm that encrypts and decrypts data in blocks of 128 bits using keys of 128, 192, or 256 bits. The AES algorithm consists of several rounds, depending on the key size. Each round includes a series of transformations applied to the data.

The decryption process mirrors the encryption process but with inverse operations. Below is a step-by-step explanation of both encryption and decryption operations. You can refer to the animation AES Rijndael Cipher explained as a Flash animation to understand more about the encryption and decryption process.

3.1 AES Encryption

AES encryption consists of the following steps:

- **Key Expansion**: The encryption key is expanded into an array of key schedule words (round keys), which will be used in the different rounds of encryption.
- **Initial Round**:
  - **AddRoundKey**: The state is combined with the round key using a bitwise XOR operation.
- **Main Rounds (We are considering 8 rounds)**:
  - **SubBytes**: A non-linear substitution step where each byte in the state is replaced with its corresponding value from a fixed lookup table S-box (3.2).
  - **ShiftRows**: This is a transposition step where each row of the state matrix is shifted cyclically by a specific number of positions to the left. The first row remains unchanged, the second row is shifted by one position, the third by two, and the fourth by three positions.
    This ensures that the columns of the state are mixed and that data from different columns interacts with each other in the following rounds, which improves diffusion across the state.
  - **MixColumns**: In this step, each column of the state matrix is treated as a polynomial and multiplied by a fixed polynomial over the Galois Field $GF(2^8)$. This operation mixes the bytes within each column, ensuring that the output bytes of a column are linear combinations of the input bytes. This step also increases diffusion by ensuring that even small changes in the input propagate throughout the entire column in subsequent rounds.
  - **AddRoundKey**: The state is again combined with a round key derived from the main key.
- **Final Round (without MixColumns)**:
  - **SubBytes**, **ShiftRows**, **AddRoundKey**.

The flow of encryption is depicted in Figure 2. Please note that we are using the standard AES-128 protocol with 8 rounds instead of 10.

3.2 AES Decryption

AES decryption reverses the encryption process by applying the inverse operations in the reverse order:

- **Key Expansion**: The same key expansion process used in encryption is performed.
- **Initial Round**:
  - **AddRoundKey**: The state is combined with the round key using a bitwise XOR operation.
- **Main Rounds (We are considering 8 rounds)**:
  - **InvShiftRows**: This is the inverse of the ShiftRows operation from encryption. Each row of the state is shifted cyclically to the right by a certain number of positions. The first row remains unchanged, the second row is shifted by one position to the right, the third by two, and the fourth by three. This restores the original row arrangement, undoing the transposition from the encryption phase.
  - **InvSubBytes**: In this step, each byte in the state is replaced by its inverse value from the inverse S-box (3.2). This reverses the non-linear substitution performed in the encryption's SubBytes step, effectively restoring the original byte values before substitution.
  - **AddRoundKey**: The state is again combined with a round key using a bitwise XOR operation. This step mirrors the encryption process, except the round keys are applied in reverse order.

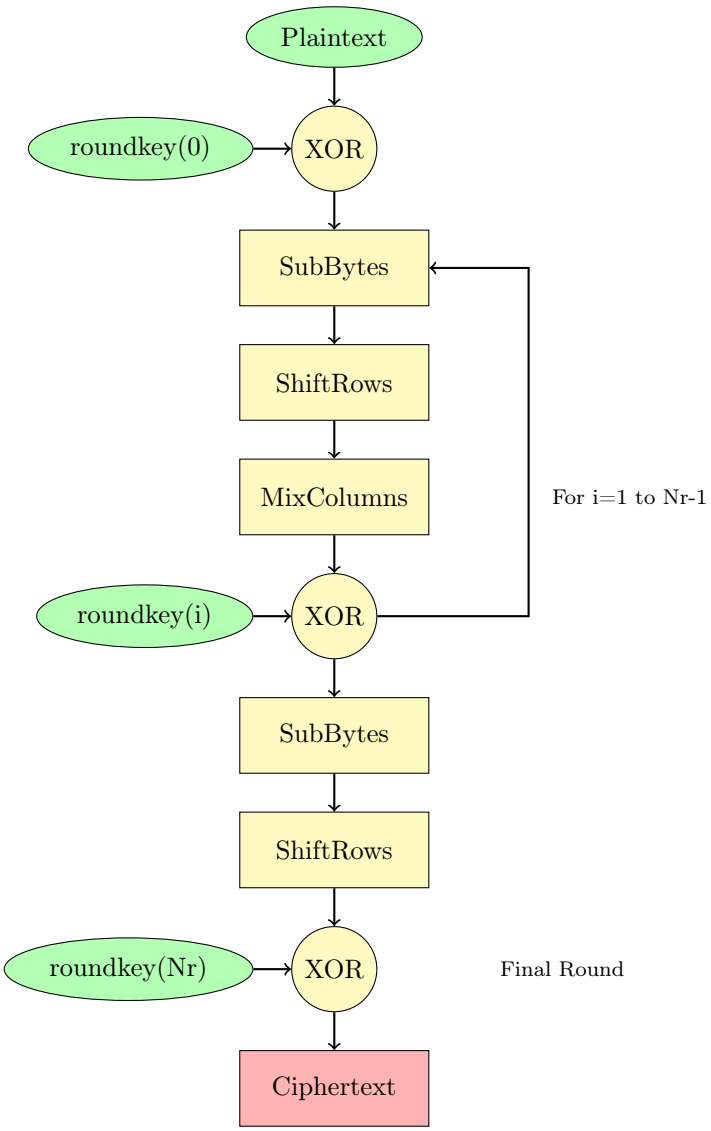


Figure 2: AES Encryption Flowchart

  - **InvMixColumns**: This step reverses the effect of MixColumns. Each column is treated as a polynomial and multiplied by the inverse fixed polynomial in the Galois Field $GF(2^8)$, effectively undoing the mixing operation and restoring the original byte arrangement within each column.

- **Final Round (without InvMixColumns)**:
  - **InvShiftRows**, **InvSubBytes**, **AddRoundKey**.

The flow of decryption is depicted in Figure 3.

```
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]
```

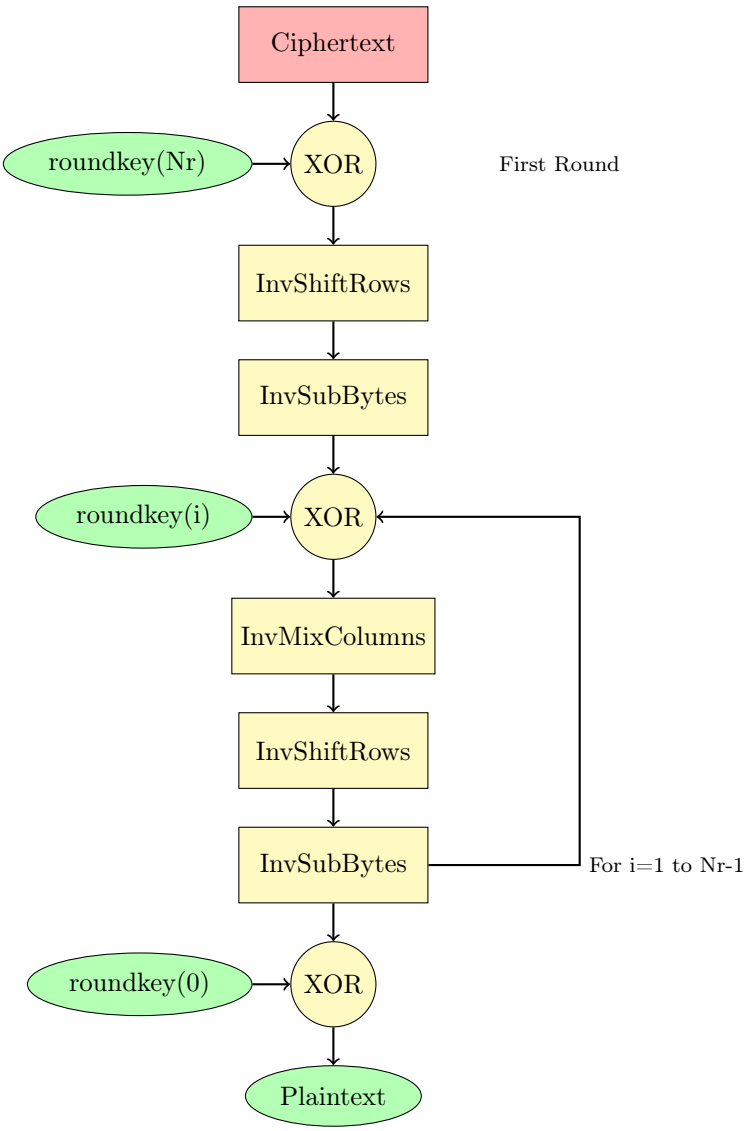


Figure 3: AES Decryption Flowchart

```
]
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]
]
```

3.2.1 Illustration of Decryption process

Following Figure 4 show the last round in decryption process, that is $InvMixColumns \rightarrow InvShiftRows \rightarrow InvSubBytes \rightarrow XOR$. The figure 4 shows the State and the corresponding operation involved. Kindly note, multiplication in the InvMixColumns is a $GF(2^8)$ (Not a decimal operation). We present an example for the steps

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

*

8B	0C	68	DA
42	70	43	4E
6D	30	00	D7
D5	1F	8A	EE

=

63	F9	5B	6F
A2	AA	12	63
67	63	6A	23
D7	63	82	82

(a) InvMixColumns operation

63	F9	5B	6F
A2	AA	12	63
67	63	6A	23
D7	63	82	82

InvRowShift

63	F9	5B	6F
63	A2	AA	12
6A	23	67	63
63	82	82	D7

(b) InvRowShift operation

63	F9	5B	6F
63	A2	AA	12
6A	23	67	63
63	82	82	D7

InvSubbytes

00	69	57	06
00	1A	62	39
58	32	0A	00
00	11	11	0D

(c) InvSubbytes operation

54	49	36	65
68	73	42	4B
31	41	79	65
73	31	74	79

$\oplus$

00	69	57	06
00	1A	62	39
58	32	0A	00
00	11	11	0D

=

54	20	61	63
68	69	20	72
69	73	73	65
73	20	65	74

(d) XOR operation

Figure 4: Illustration to perform various operations in decryption

- First operation is **InvMixColumns**. It reverses the mixing of bytes performed during the encryption's `MixColumns` step.

  The transformation uses Galois Field arithmetic in $GF(2^8)$, a finite field that allows operations on bytes (8-bit values). In $GF(2^8)$, bytes are treated as polynomials of degree less than 8, and multiplication is defined arithmetic modulo a specific irreducible polynomial, $x^8+x^4+x^3+x+1$. It is used in AES because it ensures that small changes in the input propagate through the entire ciphertext, providing strong diffusion.

In the `InvMixColumns` step, each column in the state matrix is transformed by multiplying each byte by a set of fixed constants: 0x0e, 0x0b, 0x0d, and 0x09, all in the Galois Field. These constants are derived from the inverse matrix of the `MixColumns` operation. (The multiplicative inverses of the constants $0x02$, $0x03$, and $0x01$.)

MixColumns uses the matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

The **Inverse MixColumns** undoes the diffusion using the inverse matrix:

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

The transformation for each byte is as follows, here c_0, c_1, c_2, and c_3 represent the four bytes of a single column in the state:

- For the first byte of the column:

$$\texttt{col[0]} = \texttt{xtimes\_0e}(c_0) \oplus \texttt{xtimes\_0b}(c_1) \oplus \texttt{xtimes\_0d}(c_2) \oplus \texttt{xtimes\_09}(c_3)$$

  where each term represents multiplication by the respective constant in $GF(2^8)$, and $\oplus$ represents bitwise XOR.

- For the second byte of the column:

$$\texttt{col[1]} = \texttt{xtimes\_09}(c_0) \oplus \texttt{xtimes\_0e}(c_1) \oplus \texttt{xtimes\_0b}(c_2) \oplus \texttt{xtimes\_0d}(c_3)$$

- Similarly, for `col[2]` and `col[3]`, the bytes are transformed using Galois Field multiplication.

Each function (e.g., `xtimes_0e`, `xtimes_0b`) represents multiplication by a fixed constant in the Galois Field $GF(2^8)$. For example:

- `xtimes_0e`(c_0): Multiplies the byte c_0 by 0x0e in $GF(2^8)$. This multiplication is done using shifts and XOR operations, not normal integer/decimal multiplication.
- `xtimes_09`(c_3): Multiplies the byte c_3 by 0x09 in the same manner.

In hardware, multiplication in $GF(2^8)$ is not the same as normal integer multiplication. Instead, it is implemented using a series of shift and XOR operations. Each constant multiplication (e.g., by $0x0e$, $0x0b$) can be implemented using a series of bitwise operations. For instance:

- Multiply by $0x02$: Shift the byte left by one bit.
- Multiply by $0x03$: Shift left by one bit, then XOR the original byte with the shifted result.
- Multiply by other constants: Perform additional shifts and XORs.

Hardware implementations would use combinational logic (gates) to achieve this.

We present an example for the steps for Multiplication by $0x09$. Multiplying by $0x09$ in $GF(2^8)$ is broken down as follows:

$$0x09 = 0x08 \oplus 0x01 = x^3 + 1$$

This means multiplying a byte by $0x09$ is equivalent to multiplying the byte by $0x08$ (which is the same as shifting it left by 3 bits modulo our polynomial) and then XORing the result with the original byte.

- Multiply by $0x02$: Shift the byte left by one bit. If the highest bit in result is set, XOR (modulo) it with $0x1B$.
- Multiply by $0x04$: Shift the result of $0x02$ multiplication left by one bit. If the highest bit in result is set, XOR it with $0x1B$.
- Multiply by $0x08$: Shift the result of $0x04$ multiplication left by one bit. If the highest bit is set, XOR it with $0x1B$.

– Multiply by $0x09$: First, compute multiplication by $0x08$ (shift left by 3 as in previous step), then XOR with the original byte. The XOR operation does not suffer from overflow, so we return.

- Second operation is rotating the elements in the given row. As seen in Figure 4a, no elements are shifted in row 1. Then in row 2, the first element is shifted by one place. In row 3, two elements are shifted and in row 4 three elements are left shifted.
- Further, in inverse sub bytes operation, each element is replaced by value determined using INV_SBOX (3.2). Like INV_SBOX(0x63) is 0x00, that is value at 6th row and 3rd column. Similarly, INV_SBOX(0xF9) is 0x69: value at F row and 9th column.
- The last operation is bitwise XOR with the 0th round key. The final state is the recovered plaintext which can be converted using ASCII table to actual text (**This is a secret key**). Refer to following link : ASCII Table

3.3 Components of the Decryption unit

The design consists of three parts, (i) Memory, (ii) Compute unit, and (iii) Finite State Machine (FSM). The FSM controls the reading/writing from the ROM/RAM and logical operations in the Compute unit. The Compute unit consists of components such as arithmetic units, multiplexers and other logical units. For the given design, some alternative designs are shown in Figure 5 - (i) One FSM to control both memory read/write and compute unit. (ii) Two separate FSMs: FSM1 for memory operations and FSM2 for the compute unit, with a synchronization/interaction between the two FSMs.

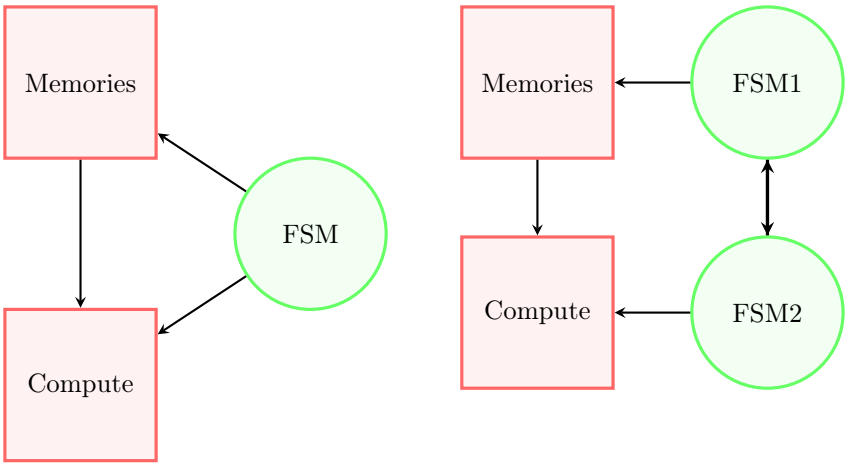


Figure 5: Alternative designs

The overall design would consist of the following:

1. Dedicated units to perform various round operations. Like multiplexer, gate array and Multiplier-Accumulator block for $GF(2^8)$ (MAC): to perform element wise multiplication and accumulation to get one output value.
2. A Read-Write Memory (RWM, more popularly known as RAM or Random-Access Memory) - Use this memory to store the output.
3. A Read-Only Memory (ROM) - Use this memory to store the input text and key.
4. Registers - Use these to store temporary results across clock cycles
5. An overall circuit that stitches all modules together to perform the filtering operation.

All the components are described in the Figure 7, and the block diagram for MAC and register block are expanded in Figure 6.

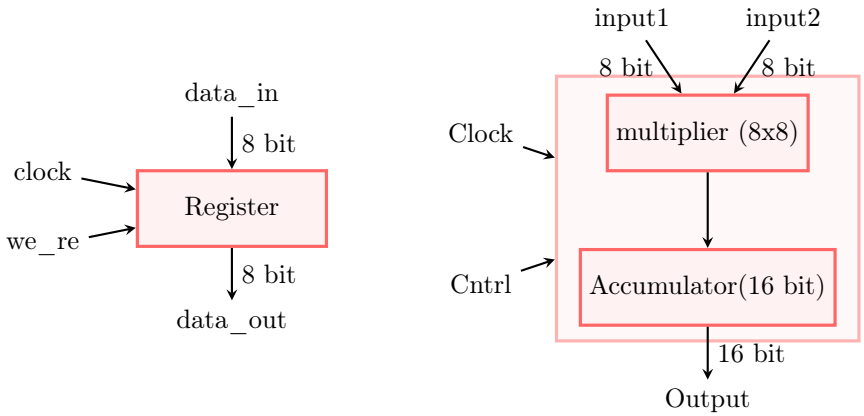


Figure 6: Register and MAC unit

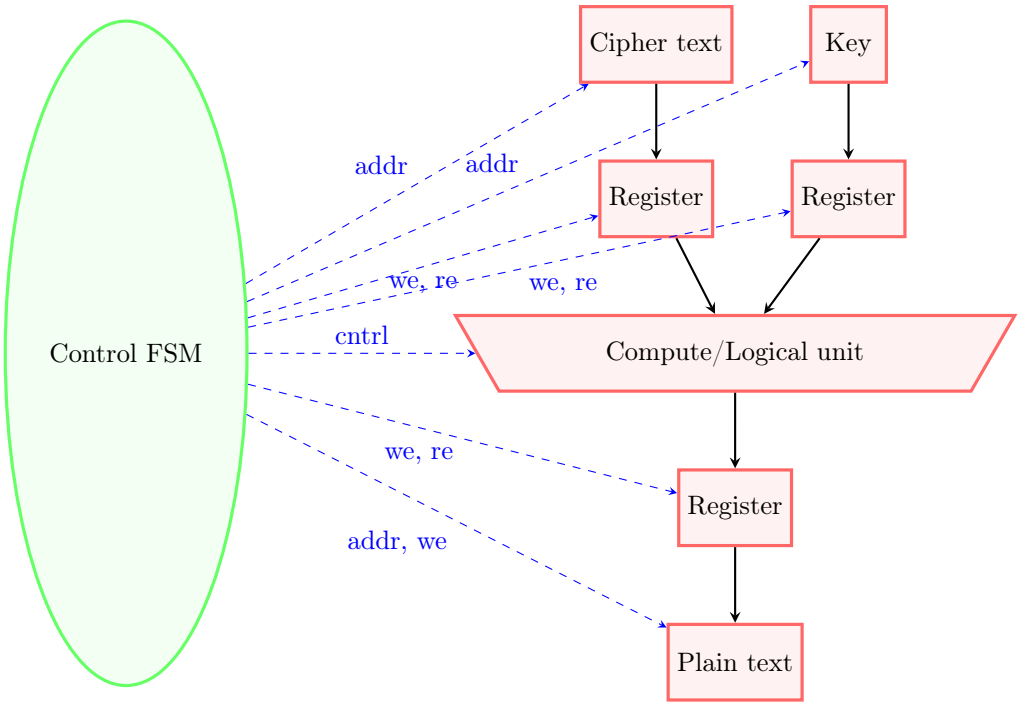


Figure 7: Overview of control path

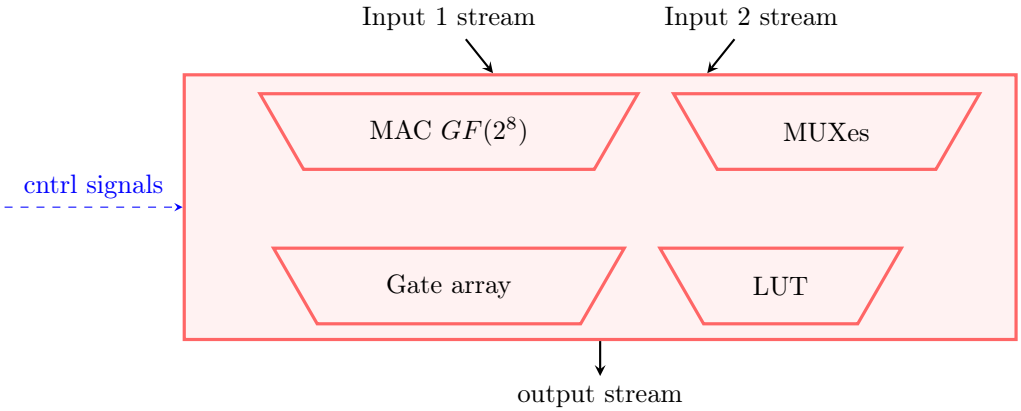


Figure 8: Details of Compute units

3.4 Read/Write timing from memory

With respect to FPGAs, memory will be synthesized using a Block RAM. Therefore, the read/write access will happen based on the clock edge. In general, write operation takes more clock cycles than read operation.

To determine **"clock cycles"** for read/write operations, need to understand the concept of setup/hold of read address line before clock and the data read access time of the given RAM. This can be determined using the given datasheet:

- Timing diagram for read and write in block RAM : Block RAM Timing Characteristics Page 43
- Timing values for the FPGA BRAM: Block RAM and FIFO Switching Characteristics Page 30

Based on clock cycles, you will need to add WAIT state (not the wait statement) in FSM for proper read/write operations.

3.5 Creating Finite State Machines in VHDL

For the control path, you will need to implement an FSM to control the compute unit and perform data transfers between ROM/RAM and the local registers. In this section, we illustrate how to implement an FSM in VHDL.

An example FSM shown in figure 9. Its job is to control the operation of a programmable adder/subtractor by sending a signal that instructs the component to ADD or SUBTRACT. The FSM consists of 2 states, ADD and SUB, along with input flags M (to change the operation) and DONE (to know when the current operation is completed); and output flag `cntrl_add_sub` to indicate the operation in the programmable adder/subtractor.

The FSM, specified in VHDL, consists of two parts:

1. Sequential block: In this block, the state is updated on the clock edge or initialized to a default state if the reset flag is set HIGH.
2. Combinational Block: This consists of the logic to update the next state based on the flag values and the current state.

The following is the VHDL template for implementing an FSM. You can see that the state of the system is captured in the Sequential block and updated every positive clock edge, and the work performed in one clock cycle is indicated/controlled in the Combinational block.

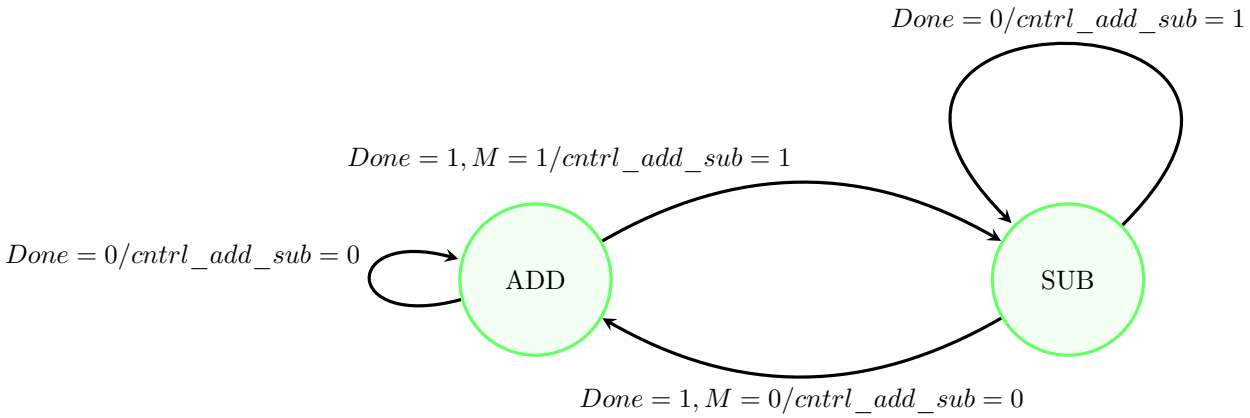


Figure 9: FSM for adder/subtractor

```
entity FSM IS
    Port ( M : in STD_LOGIC;
           Done : in STD_LOGIC;
           clk    : in  STD_LOGIC;
           reset : in STD_LOGIC;
           cntrl_add_sub : out STD_LOGIC;
         );
end FSM;
architecture machine of FSM IS
    type state_type is (ADD, SUB);
```

```
    signal   cur_state          : state_type := ADD;
    signal   next_state         : state_type := ADD;
begin
   --Sequential block
   process (clk, reset)
   begin
      if (reset = '1') then
         cur_state <= ADD;
      elsif (clk'EVENT AND clk = '1') then
         cur_state <= next_state;
      end if;
    end process;
    --Combinational block
    process (cur_state, M, Done)
    begin
        next_state <= cur_state;

        case cur_state is
            when ADD =>
                if Done = '1' and M = '1' then
                    next_state <= SUB;
                    cntrl_add_sub <= '1';
                elsif Done = '0' then
                    next_state <= ADD;
                    cntrl_add_sub <= '0';
                end if;
            when SUB =>
                if Done = '1' and M = '0' then
                    next_state <= ADD;
                    cntrl_add_sub <= '0';
                elsif Done = '0' then
                    next_state <= SUB;
                    cntrl_add_sub <= '1';
                end if;
        end case;
    end process;
end machine;
```

4 Displaying the plaintext

To display the final text, you will be using the seven segment display of the basys 3 board. You are allowed to reuse the VHDL code from HW2 assignment. The following points should be kept in mind:

- After the final round, you need to convert the hexa-decimal value to the corresponding character using ASCII Table.
- We will restrict the characters to be displayed in the range 0-F. For any out of the range character display a default value "-" as shown in Figure 10. Plaintext will contain blank space as well.
- No case sensitivity that is display capital F and small f as F on the display.
- Only four character can be displayed at a time, your code should ensure that text keep scrolling across the display in cyclic manner.

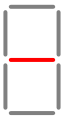


Figure 10: Output for out of range character

5 Submission and Demo Instructions

Since this assignment will span over multiple weeks, your progress in each week will be evaluated as described below.

1. **Make sure that you have a working system first before optimizing it. Another golden rule: Always ensure that a simulation is working correctly before testing on the FPGA board.**
2. **PART 2. Week 3-4 (27th October - 10th November 2024): Implement FSM and integrate it with the hardware design.**
3. You are required to submit a zip file containing the following via gradescope:
    - VHDL files for all the designed modules.
    - Simulated waveforms with test bench of each subcomponent and complete design.
    - Include the FSM state diagram in the report. Simulation waveform showing change in signals with the state.
    - A short report (1-2 pages) explaining your approach. Include block diagram depicting the modules. And short summary about functionality of each module.

6 Resources references

- IEEE document: https://ieeexplore.ieee.org/document/8938196
- Basys 3 board reference manual: https://digilent.com/reference/_media/basys3:basys3_rm.pdf
- Online VHDL simulator: https://www.edaplayground.com/x/A4