The Advanced Encryption Standard (AES) on an FPGA

Lecture 14

Josh Brake
Harvey Mudd College

Outline

- AES Overview
- AES Implementation Details
  - Block diagram
  - Embedded Block RAMs
  - Timing

Learning Objectives

By the end of this lecture you will…

- Have an operational understanding of the fundamental mathematics used in AES
- Understand the basic process of AES

The Advanced Encryption Standard

AES Overview

From the spec:

> The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

128-bit AES

- The cipher works by taking a plain text message and scrambling it into a ciphertext message in a way that is hard to reverse.
- Scrambling is done 10 times to make it hard. The key changes from round to round.

AES Data Organization

128-bit message is organized as a matrix of 16 bytes (4x4).

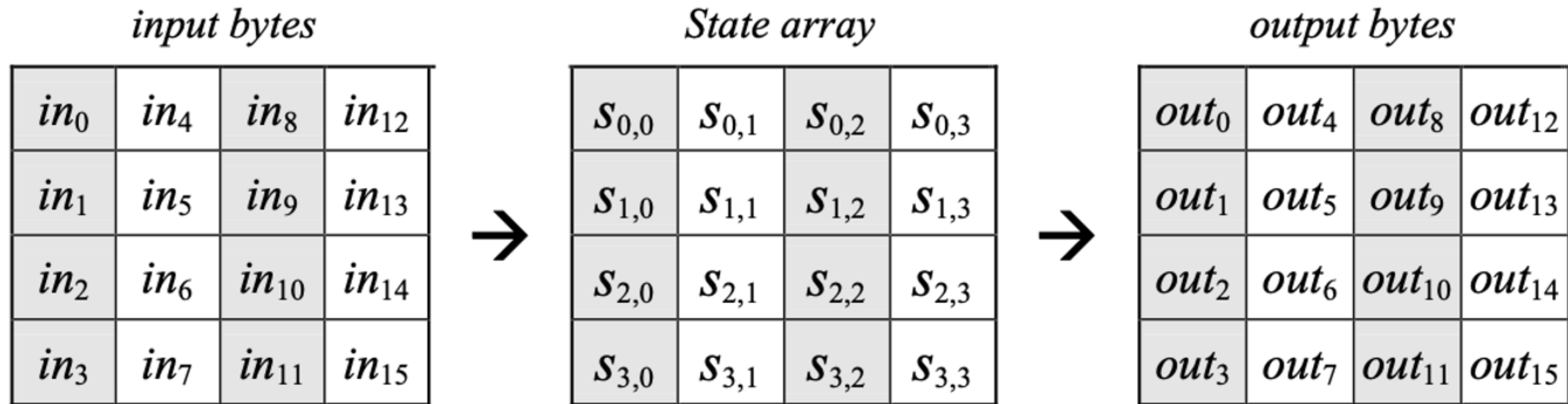

Figure 3. State array input and output.

AES Cipher Process

Each step of the cipher involves the following steps

1. `SubBytes`: take each byte and replace it with a different byte using a randomish lookup table.
2. `ShiftRows`: move bytes around in the rows
3. `MixColumns`: funky Galois multiplication on elements of the columns
4. `AddRoundKey`: XOR with the key for the current round.

AES Cipher Pseudocode

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])            // See Sec. 5.1.4

   for round = 1 step 1 to Nr–1
      SubBytes(state)                        // See Sec. 5.1.1
      ShiftRows(state)                       // See Sec. 5.1.2
      MixColumns(state)                      // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 5. Pseudo Code for the Cipher.[1]

AES Implementation

Design Process

Design the AES algorithm in the following way:

1. Think about each individual transformation as a separate module. Is the operation of the module combinational or sequential?
2. What are the inputs?
3. What are the outputs?
4. Write testbenches for each individual module and test each module individually.
5. If there is a sequential element, design an FSM. (Hint: the structure from a processor of controller and datapath is very helpful.)

AES Block Diagram

Spend a few moments thinking through one round of the AES cipher. Draw a block diagram which shows each step of the cipher (e.g., function call in the pseudo code) as an individual block. Consider the size of the inputs/outputs.

As you design, consider the following questions:

- How many rounds are required for AES-128?
- How does the final round differ from the all the others?
- How would you prevent a certain operation from applying in a certain round?
- Can the entire process be implemented as combinational logic?
- Can the cipher be implemented as a sequential process?
- What tradeoffs exist between a fully combinational vs. a sequential design?

State matrix

How do we transform the 128-bit input into the state array?

It's easier to manipulate the data using the array notation as described in the spec, so let's define a new module that restructures the 128-bit key as a 4x4 array of bytes.

```
// AES state type definition
typedef logic [0:3][0:3] [7:0] aes_state_t;
```

Input2State Verilog

```
// AES state type definition
typedef logic [0:3][0:3] [7:0] aes_state_t;

`include "AESTypes.svh"

module Input2State(
  input   logic [127:0] in,
  output  aes_state_t aes_state
);

  // Takes in the 1d in and transforms it into the state array
  assign aes_state[0][0] = in[127:120];
  assign aes_state[1][0] = in[119:112];
  assign aes_state[2][0] = in[111:104];
  assign aes_state[3][0] = in[103:96];

  ...

  assign aes_state[0][3] = in[31:24];
  assign aes_state[1][3] = in[23:16];
  assign aes_state[2][3] = in[15:8];
  assign aes_state[3][3] = in[7:0];

endmodule
```

SubBytes

What are the inputs?

A byte.

What are the outputs?

A byte mapped through the sbox substitution lookup table.

What is this module doing (in words)?

Replacing one byte with another byte.

SubBytes Verilog

What does the Verilog look like to accomplish this?

```
module SubstitutionBox (
  input   logic [7:0] a,
  output  logic [7:0] y
);

  // Signal to store entries for byte substitution
  logic [7:0] sbox_lut [255:0];

  // Intialize the RAM with the values for the byte substitution
  initial $readmemh("sbox.txt", sbox_lut);

  // Combinationally assign output (no clock)
  assign y = sbox_lut[a];

endmodule
```

ShiftRows

What are the inputs?

A row of state (32 bits, 4 bytes, 1 word).

What are the outputs?

A shifted row (32 bits, 4 bytes, 1 word).

What is this module doing (in words)?

Circular left shift each row based on the row number (e.g., row 0 unshifted, row 1 shifted to the left by one element, etc.)

MixColumns

What are the inputs?

A column of the state matrix (32 bits, 4 bytes, 1 word).

What are the outputs?

A new of the state matrix (32 bits, 4 bytes, 1 word).

What is this module doing (in words)?

Transforming the column according to a matrix multiplication. This can be instantiated as a somewhat tricky-looking Galois multiplication.

AddRoundKey

What are the inputs?

- State matrix.
- Previous round key (128-bits).

What are the outputs?

New state (128-bits).

What is this module doing (in words)?

Computing a new round key based on the previous one.

`GenerateRoundkey`

What are the inputs?

The key.

What are the outputs?

A key schedule (set of 128-bit keys)

What is this module doing (in words)?

Create the key schedule based on the initial key and a set of other operations.

Embedded Block RAMs

3.1.5. sysMEM Embedded Block RAM Memory

Larger iCE40 UltraPlus device includes multiple high-speed synchronous sysMEM Embedded Block RAMs (EBRs), each 4 kbit in size. This memory can be used for a wide variety of purposes including data buffering and FIFO.

sysMEM Memory Block

The sysMEM block can implement single port, pseudo dual port, or FIFO memories with programmable logic resources. Each block can be used in a variety of depths and widths as listed in Table 3.4.

Table 3.4. sysMEM Block Configurations

Block RAM Configuration	Block RAM Configuration and Size	WADDR Port Size (Bits)	WDATA Port Size (Bits)	RADDR Port Size (Bits)	RDATA Port Size (Bits)	MASK Port Size (Bits)
SB_RAM256x16 SB_RAM256x16NR SB_RAM256x16NW SB_RAM256x16NRNW	256x16 (4 k)	8 [7:0]	16 [15:0]	8 [7:0]	16 [15:0]	16 [15:0]
SB_RAM512x8 SB_RAM512x8NR SB_RAM512x8NW SB_RAM512x8NRNW	512x8 (4 k)	9 [8:0]	8 [7:0]	9 [8:0]	8 [7:0]	No Mask Port
SB_RAM1024x4 SB_RAM1024x4NR SB_RAM1024x4NW SB_RAM1024x4NRNW	1024x4 (4 k)	10 [9:0]	4 [3:0]	10 [9:0]	4 [3:0]	No Mask Port
SB_RAM2048x2 SB_RAM2048x2NR SB_RAM2048x2NW SB_RAM2048x2NRNW	2048x2 (4 k)	11 [10:0]	2 [1:0]	11 [10:0]	2 [1:0]	No Mask Port

Note: For iCE40 UltraPlus, the primitive name without "Nxx" uses rising-edge Read and Write clocks. "NR" uses rising-edge Write clock and falling-edge Read clock. "NW" uses falling-edge Write clock and rising-edge Read clock. "NRNW" uses failing-edge clocks on both Read and Write.

Embedded Block RAM: Block Diagram

RAM4k Block

Figure 3.4 shows the 256x16 memory configurations and their input/output names. In all the sysMEM RAM modes, the input data and addresses for the ports are registered at the input of the memory array.

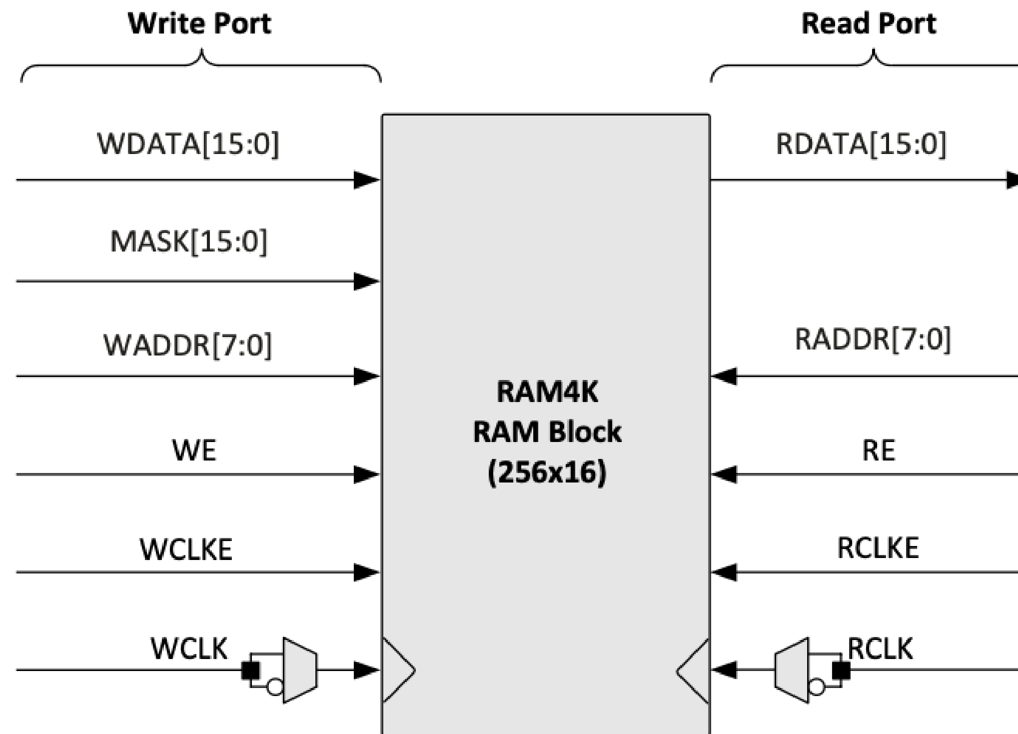

Figure 3.4. sysMEM Memory Primitives

Cascading Memories

4.3.1. Address Cascading (or Depth Cascading)

Address/Depth cascading is useful when the memories are required to have the capacity of storing *more* words while keeping the data width the same. In this case additional user logic is needed to decode the address.

Figure 4.1 shows an example of the depth cascading of a 32K x 16 SPRAM. Additional logic is required that guides the data to the correct memory block using Muxes and Demuxes. The rest of the signals (that are not shown), should be connected to both the memory blocks without any other logic requirements.

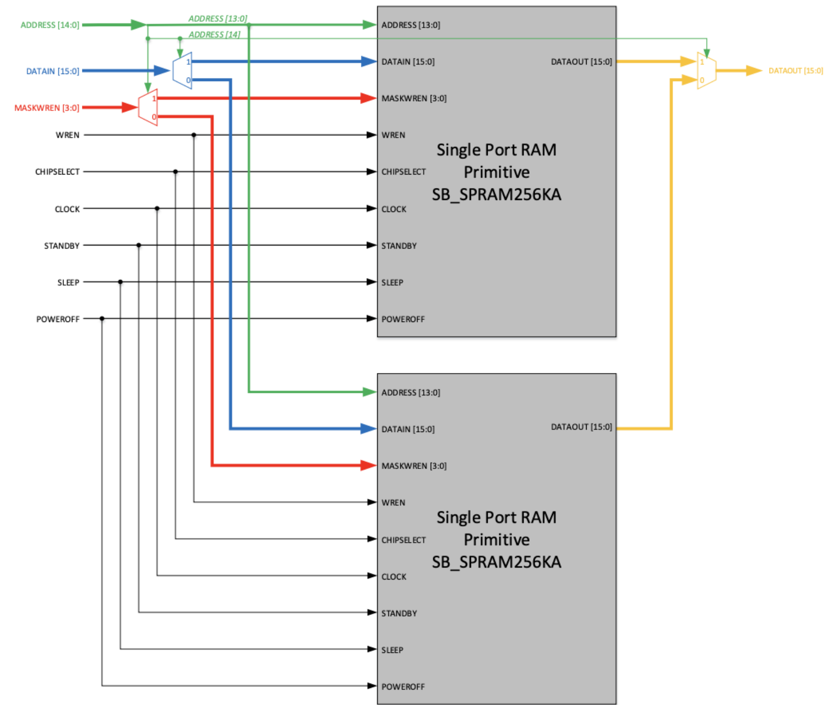

Figure 4.1. Address/Depth Cascading Example for 32K x 16 SPRAM using Primitive