ECE 385 – Digital Systems Laboratory

Lecture 16 – Introduction to Experiment 9

Zuofu Cheng

Fall 2018

Link to Course Website

ECE ILLINOIS

ILLINOIS

Experiment 9 overview

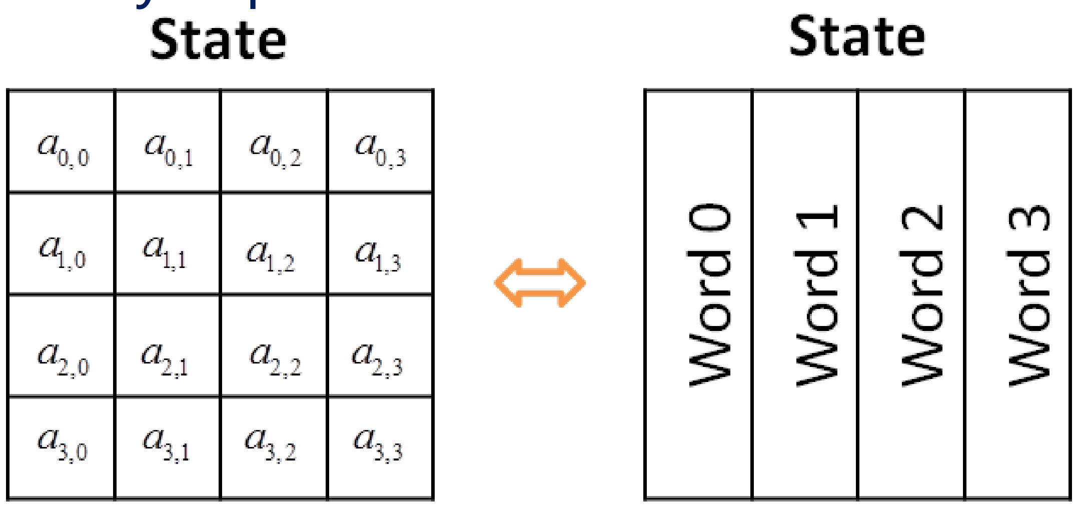
- Week 1
  - AES encryption implementation on the Nios II SoC
  - Written in C
  - Performance benchmark
  - Create HW/SW communication in preparation for week 2
- Week 2
  - AES decryption implementation in hardware
  - Written in SystemVerilog
  - Performance benchmark
- Compare relative performance of the hardware and software implementations
  - For most parts, encryption and decryption are largely symmetric

128-bit Advanced Encryption Standard (AES)

- A symmetric block cipher based on the Rijndael algorithm
- A *cypher* is an encryption process to transfer meaningful message into scrambled data for the purpose of data transfer security
  - Inputs:
    - Message, called *Plaintext*
    - A secret key, called *Cipher Key*
  - Output:
    - Scrambled text, called *Ciphertext*
- *Inverse Cipher* is the dual of the cipher
  - We will use the same key, thus the entire process of encoding/decoding is called a *symmetric-key algorithm*

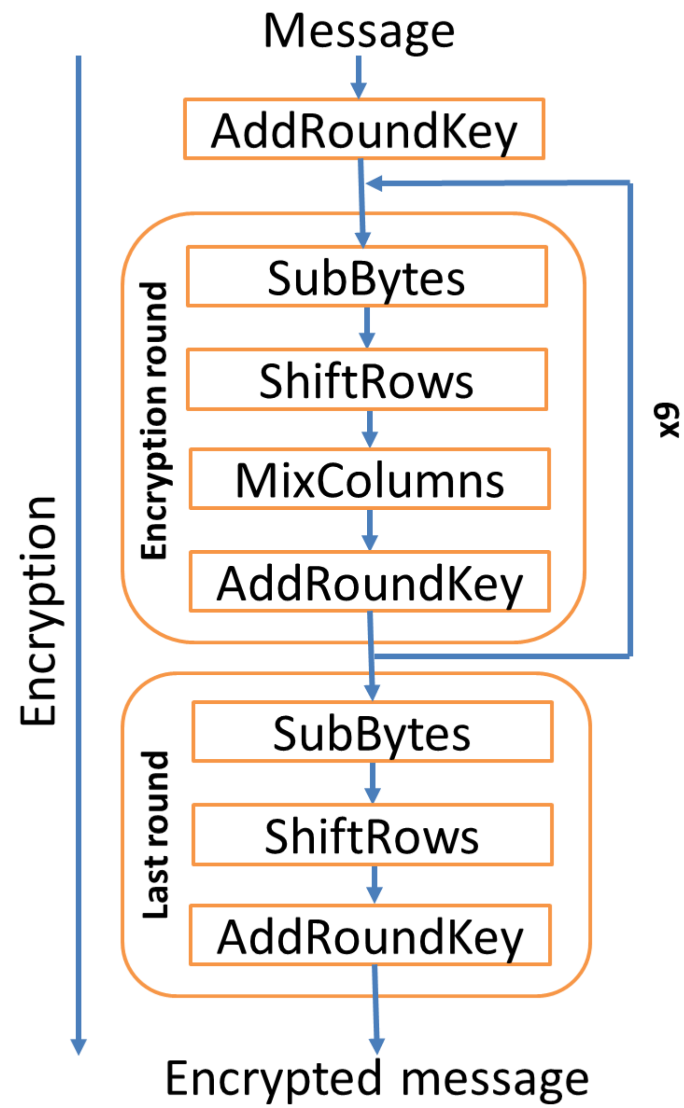
AES (128-bit)

- State – 128-bit intermediate results during the AES algorithm, arranged in column major matrix of 4x4 Bytes
- Word – the 4-Byte data from a single State column.
- Round Key – 128-bit keys derived from the Cipher Key using the Key Expansion routine. It is applied in different stages of the algorithm
- Key Schedule – 11x128-bit Round Keys derived from the Cipher Key using the Key Expansion routine



AES Encryption Algorithm

- An AES encryption goes through several "rounds"
  - Each round consists of several module routines
  - 10 rounds for 128-bit AES, with 9 full rounds and a reduced last round
  - Round Keys in AddRoundKey() are generated separately in KeyExpansion() using the Cipher Key

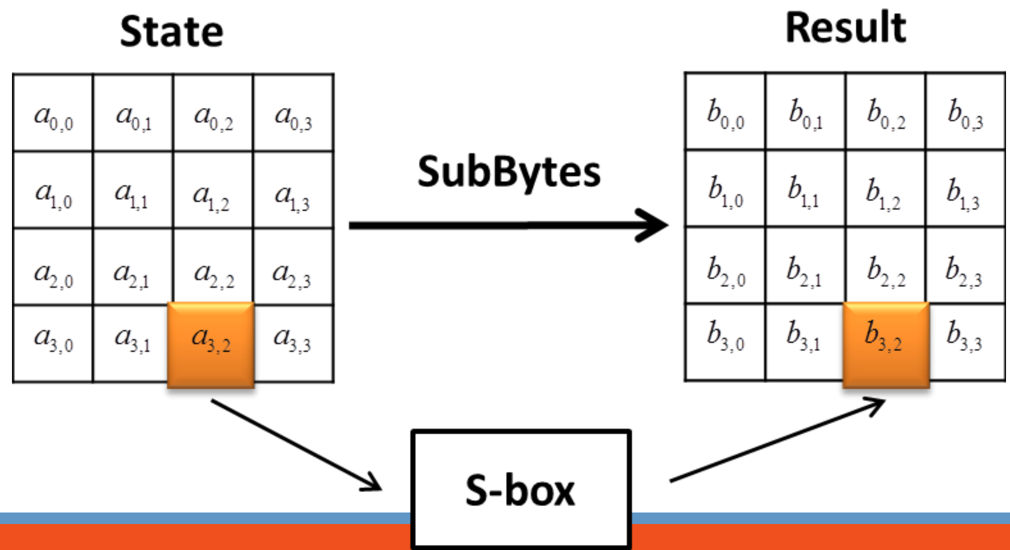


AES Encryption Algorithm Overview

- Core algorithm takes in block (16 bytes) and generates another block (16 bytes)
- Keep in mind **Column Major** ordering (e.g. FORTRAN style - different from C style)
- 10 rounds of 4 steps (only 3 in last round)
- Steps are:
  - Substitution
  - Shift
  - Mix (multiplication)
  - AddRoundKey

SubBytes()

- SubBytes performs transformation in the Rijndael's finite field
  - First find the multiplicative inverse of each Byte
  - Then use an affine transformation in GF(2^8) to obtain the final value
    - GF(2^8) stands for *Galois field*, or a field that contains a finite number of elements, 2^8 in this case
  - This process is usually pre-computed and stored in Rijndael's S-box (substitution box) as a lookup table



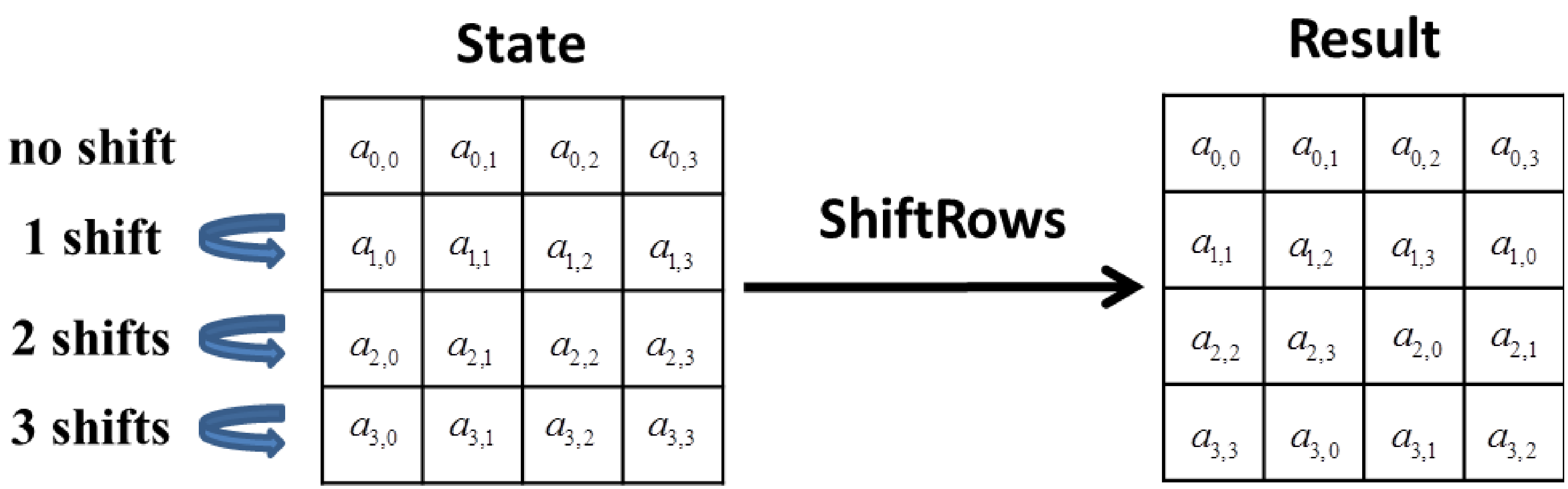
SubBytes()

- Usually we don’t compute the S-box (would be slow)
- Instead use a lookup table (`const uchar aes_sbox[16][16]`)
- This is byte-wise substitution, what is necessary size of the lookup table?

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1x	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2x	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3x	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4x	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5x	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6x	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7x	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8x	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9x	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
ax	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
bx	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
cx	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
dx	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
ex	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
fx	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

ShiftRows()

- ShiftRows performs circular left shift
  - First row remains unchanged
  - Second row is left-circularly shifted by 1 Byte
  - Third row is left-circularly shifted by 2 Bytes
  - Fourth row is left-circularly shifted by 3 Bytes



MixColumns()

- Multiply each column by matrix as shown in **GF(2^8)**

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

- Note that this is more complex than normal multiplication (and addition – as required by matrix multiplication)
  – Involves multiplication of polynomials
- Can use LUT here as well (since we only multiply by 1, 2 and 3)

Rijndael's Finite Field

- Finite field arithmetic in Galois field GF(2^8)
  - Represents binary numbers using polynomials

    e.g. $\{57\} = 01010111 = x^6 + x^4 + x^2 + x + 1$
  - Addition and subtraction in GF(2^8) are performed with XOR

    e.g. $\{57\} \oplus \{83\} = \{d4\}$
  - Multiplication in GF(2^8) is performed with the multiplication of the polynomials (XOR product terms with the same exponential), then modulo an irreducible polynomial (only divisors are one and itself) of degree 8: $(x^8 + x^4 + x^3 + x + 1)$

$$
\begin{aligned}
&\{57\} \bullet \{83\} \\
&= (x^6 + x^4 + x^2 + x + 1)(\mathrm{x}^7 + \mathrm{x} + 1) \text{ modulo } (x^8 + x^4 + x^3 + x + 1) \\
&= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \text{ modulo } (x^8 + x^4 + x^3 + x + 1) \\
&= \mathrm{x}^7 + \mathrm{x}^6 + 1 = 11000001 = \{\mathrm{c}1\}
\end{aligned}
$$

MixColumns() Continued

- MixColumns performs matrix multiplication with each Word $w_i = \{a_{0,i}, a_{1,i}, a_{2,i}, a_{3,i}\}^T$ under Rijndael's finite field
  - $(\{02\} \bullet a)$ can be implemented by bit-wise left shift then a conditional bit-wise XOR with {1b} if the 8th bit before the shift is 1
  - $\{03\} \bullet a = (\{02\} \bullet a) \oplus a$
  - It is also possible to use a pre-computed lookup table

$$
\begin{aligned}
b_{0,i} &= (\{02\} \bullet a_{0,i}) \oplus (\{03\} \bullet a_{1,i}) \oplus a_{2,i} \oplus a_{3,i} \\
b_{1,i} &= a_{0,i} \oplus (\{02\} \bullet a_{1,i}) \oplus (\{03\} \bullet a_{2,i}) \oplus a_{3,i} \\
b_{2,i} &= a_{0,i} \oplus a_{1,i} \oplus (\{02\} \bullet a_{2,i}) \oplus (\{03\} \bullet a_{3,i}) \\
b_{3,i} &= (\{03\} \bullet a_{0,i}) \oplus a_{1,i} \oplus a_{2,i} \oplus (\{02\} \bullet a_{3,i})
\end{aligned}
$$

MixColumns() Examples

- Examples:
- A) $\{01\} \bullet \{d4\} = \{d4\}$
- B) $\{02\} \bullet \{d4\} = 11010100 << 1$ (always left shift by 1)

  $= 10101000 \oplus 00011011$ (XOR with $\{1b\}$ since MSB of $\{d4\}$ is 1)

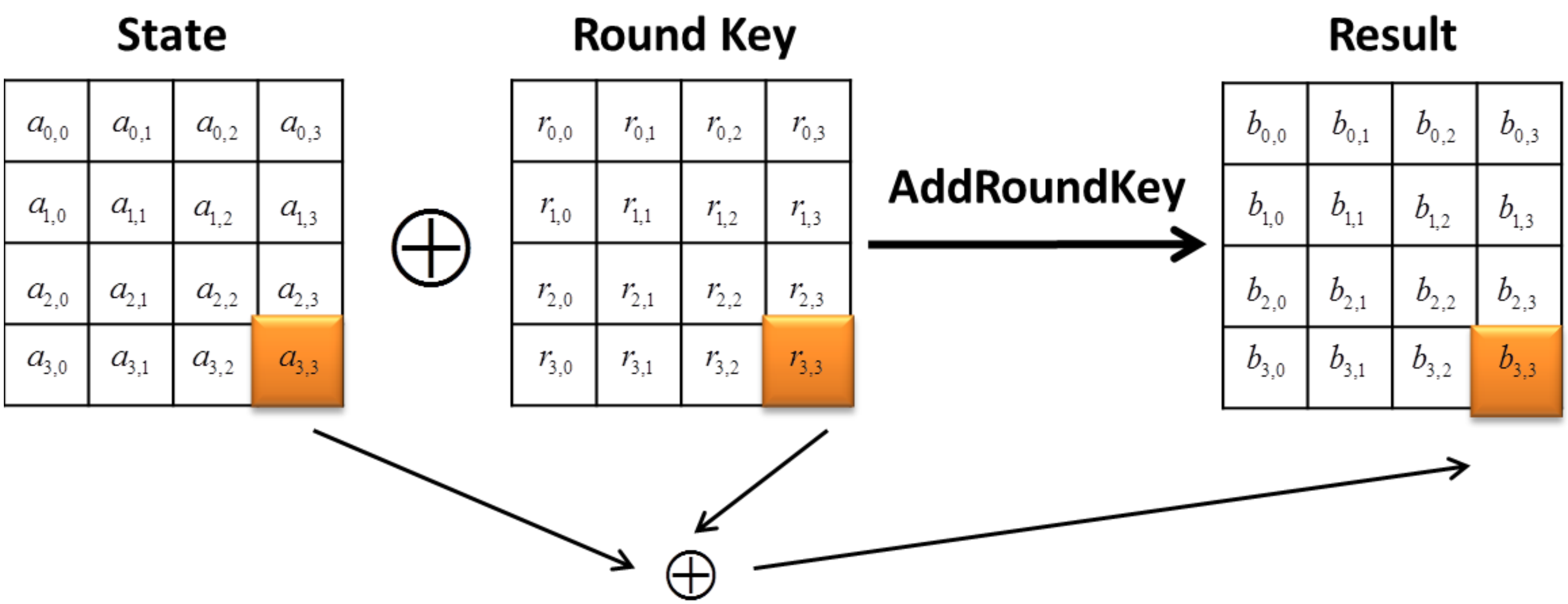
  $= 10110011 = \{b3\}$
- C) $\{03\} \bullet \{d4\} = (\{02\} \bullet \{d4\}) \oplus \{d4\}$

  $= 10110011 \oplus 11010100$

  $= 01100111 = \{67\}$

AddRoundKey ()

- XORs each Byte with the corresponding Byte from the current RoundKey
  - Each Round of the algorithm uses different RoundKeys
  - Each RoundKey is generated from the previous RoundKey
  - RoundKeys can be generated either altogether at the beginning of the AES algorithm, or during each round



KeyExpansion()

- KeyExpansion generates a RoundKey at a time based on the previous RoundKey (use the Cipher Key to generate the first RoundKey)
  - RotWord() – circularly shift each Byte in a Word up by 1 Byte
  - SubWord() – identical to SubBytes()
  - Rcon() – xor the Word with the corresponding Word from the Rcon lookup table

for every Word w_i in all n RoundKeys (*i=1,2,…,4n, n=10*)

    $w_{temp} = w_{i-1}$

    if w_i is the first Word in the current RoundKey

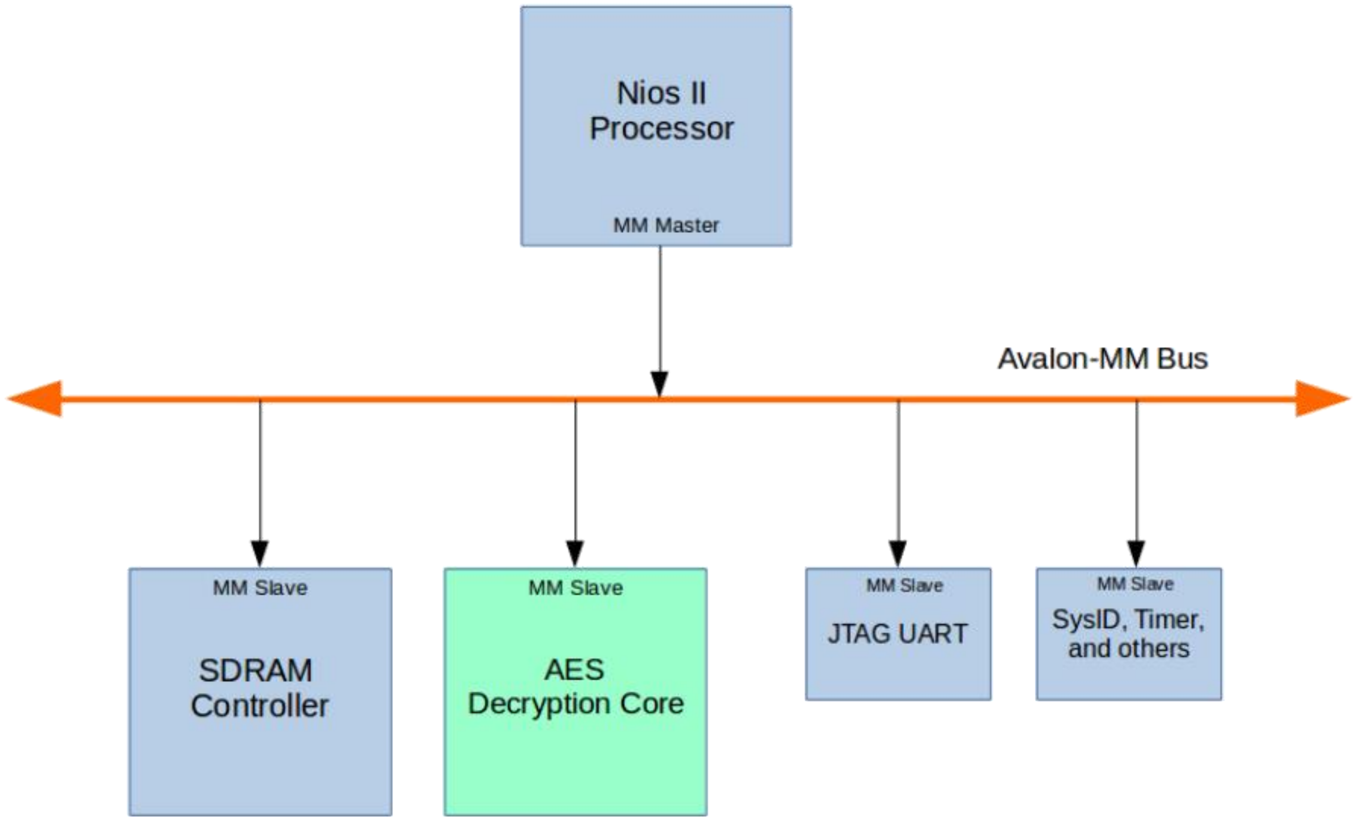
        w_{temp} = SubWord(RotWord(w_{temp})) **xor** Rcon$_n$

    for every Word in the current RoundKey, including the first Word

        $w_i = w_{i-4}$ **xor** w_{temp}

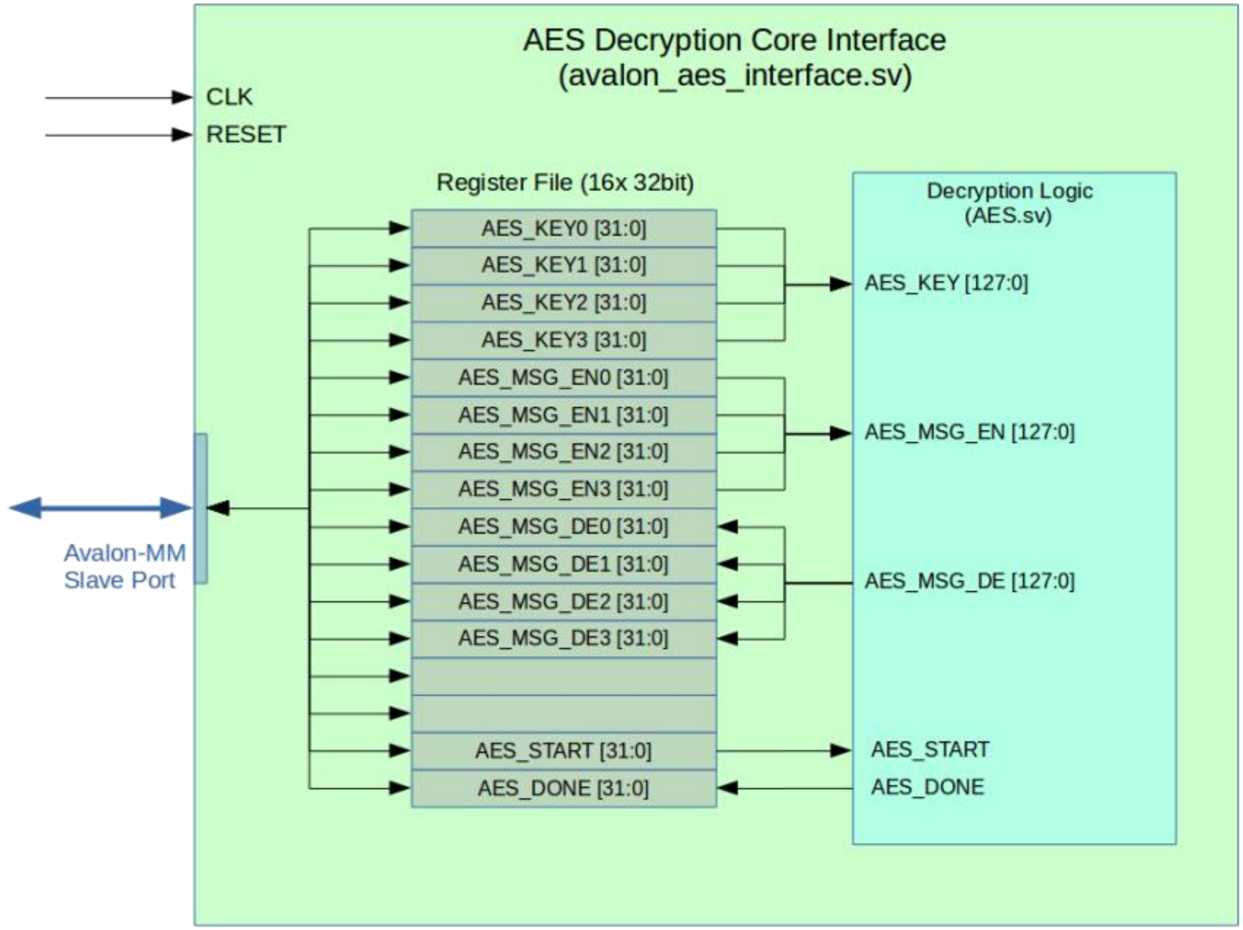
Nios II System Components in Lab 9

- Create a Avalon MM interface
- Decode 16 registers to use for HW/SW communication



AES Decryption Core Interface

- 16 Registers, each 32-bits (you need to write this)
- How many bytes on Avalon bus?



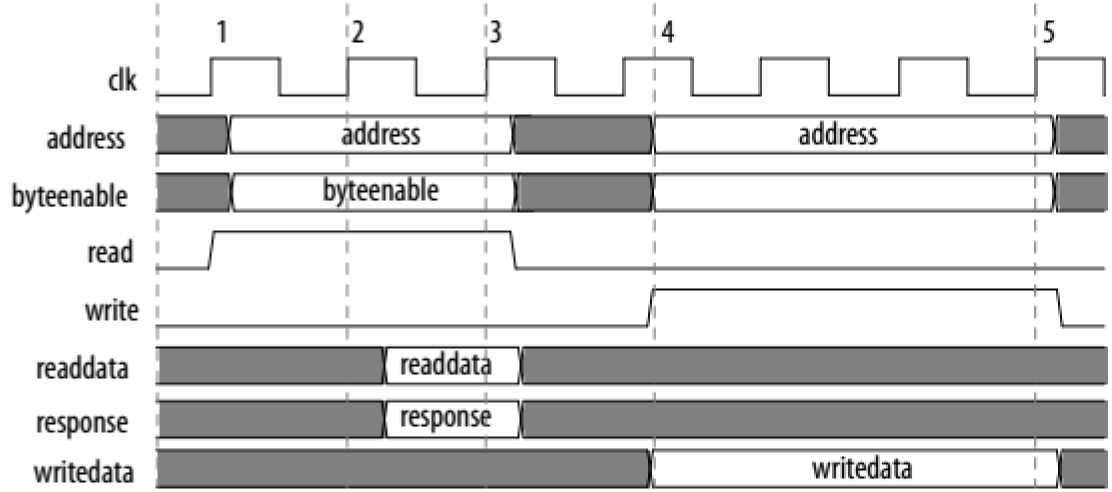
Avalon MM Interface Signals

- You will create a module which will decode Avalon bus (following chart) and place data into 16 registers
- Note address is only 4 bits, don't need to decode full 32-bit address

Name	Direction	Width	Description
read	Input	1	High when a read operation is to be performed
write	Input	1	High when a write operation is to be performed
readdata	Output	32	32-bit data to be read
writedata	Input	32	32-bit data to be written
address	Input	4	Address of the read or write operation
byteenable	Input	4	4-bit active high signal to identify which byte(s) are being written
chipselect	Input	1	High during a read or write operation

Read/Write Timing

- We will use the Avalon-MM bus with **fixed** wait-states
- Number of wait states will be 0*
- Note: timing diagram has wait state of 1! What would same thing with wait state = 0 look like?



byteenable[3:0]	Write Action
1111	Write full 32-bits
1100	Write the two upper bytes
0011	Write the two lower bytes
1000	Write byte 3 only
0100	Write byte 2 only
0010	Write byte 1 only
0001	Write byte 0 only

Components with zero wait-states are allowed. However, components with zero wait-states may decrease the achievable frequency. Zero wait-states require the component to generate the response in the same cycle that the request was presented.