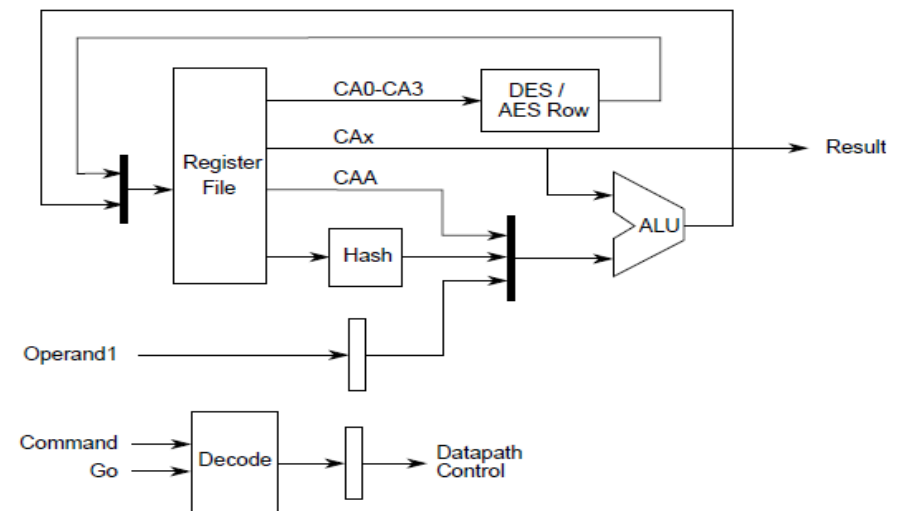
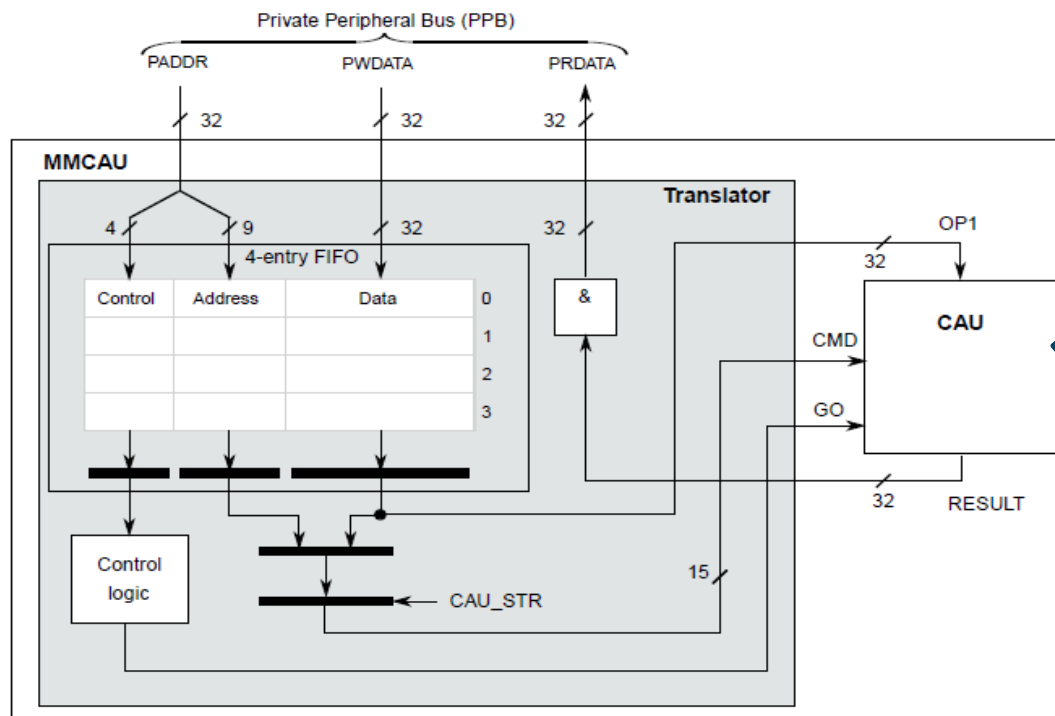


MMCAU Training Outline

1. **Module Overview**
2. On-chip interconnects and inter-module dependencies
3. Software configuration
4. Demo code explanation
5. Frequently asked question list
6. Reference material

Block Diagram

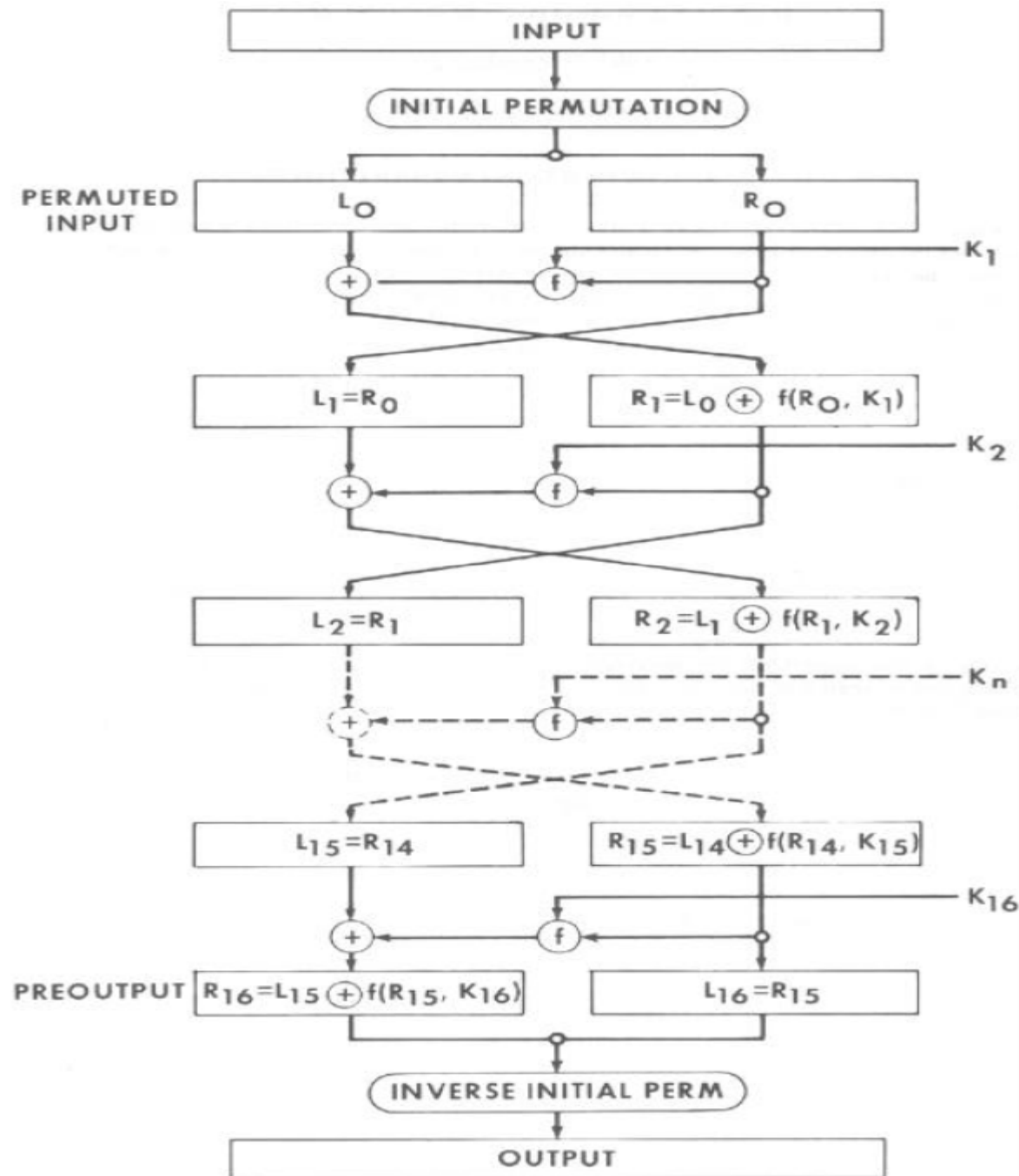


Feature list

- ▶ Supports DES, 3DES, AES, MD5, SHA-1, and SHA-256 algorithms
- ▶ Simple, flexible programming model
- ▶ Ability to send up to three commands in one data write operation

Overview of Supported Security Algorithms

DES

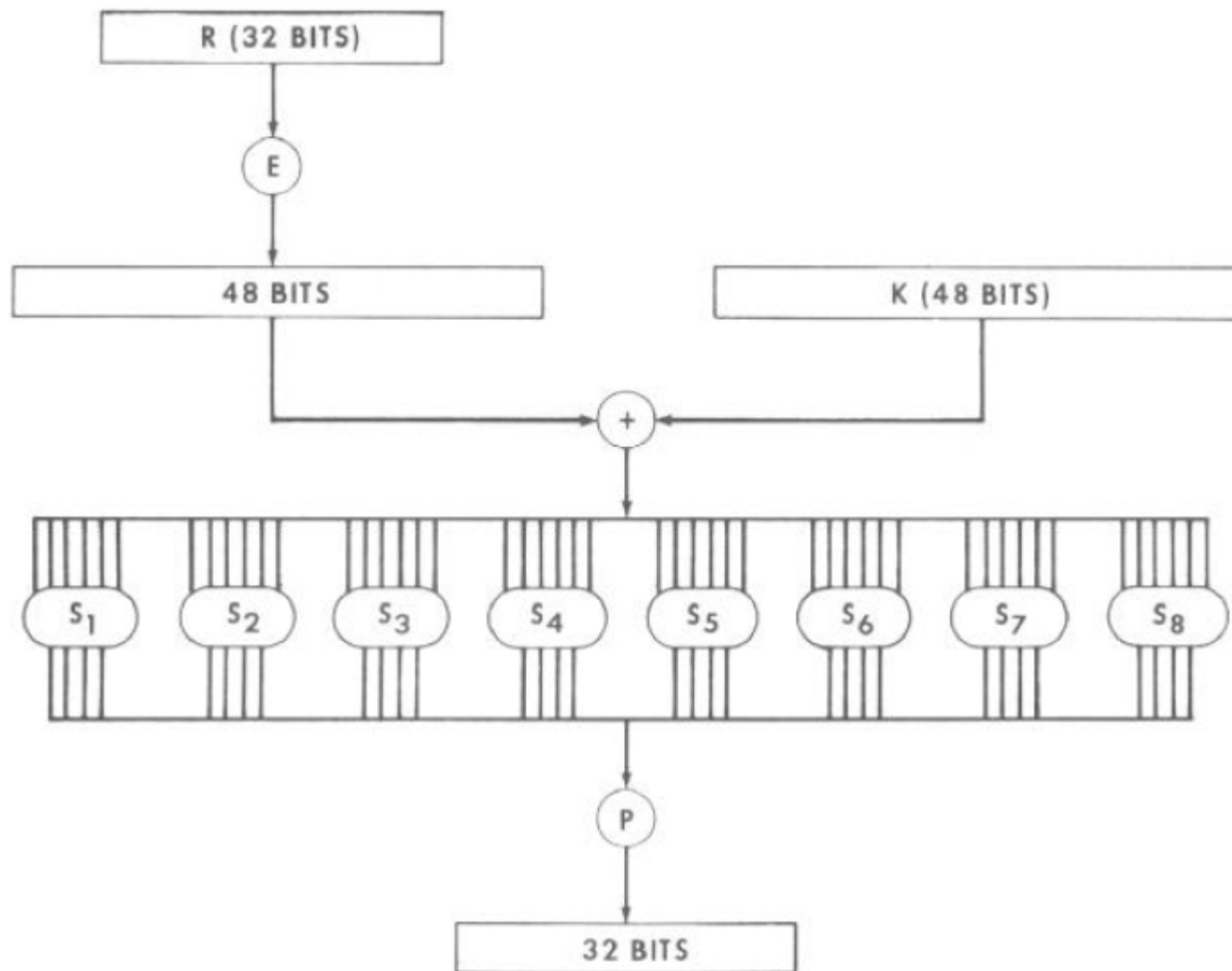


Encryption

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$

DES



**Feistel (F) function: calculation
of $f(R,K)$**

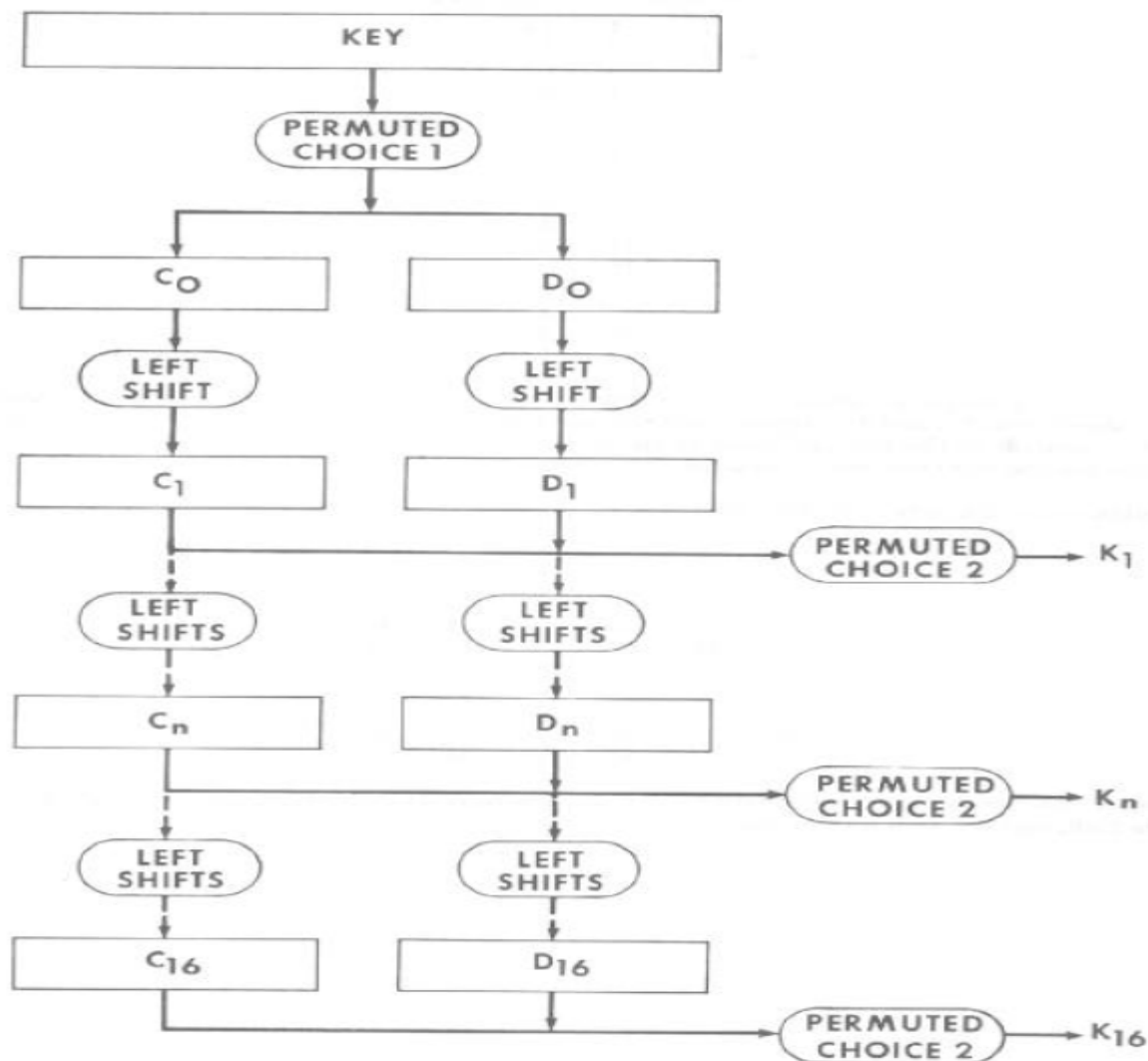
Decryption:

- Same as Encryption
- The permutation IP^{-1} applied to the preoutput block is the **inverse of the initial permutation IP** applied to the input.

$$\begin{aligned}R_{n-1} &= L_n \\L_{n-1} &= R_n \oplus f(L_n, K_n)\end{aligned}$$

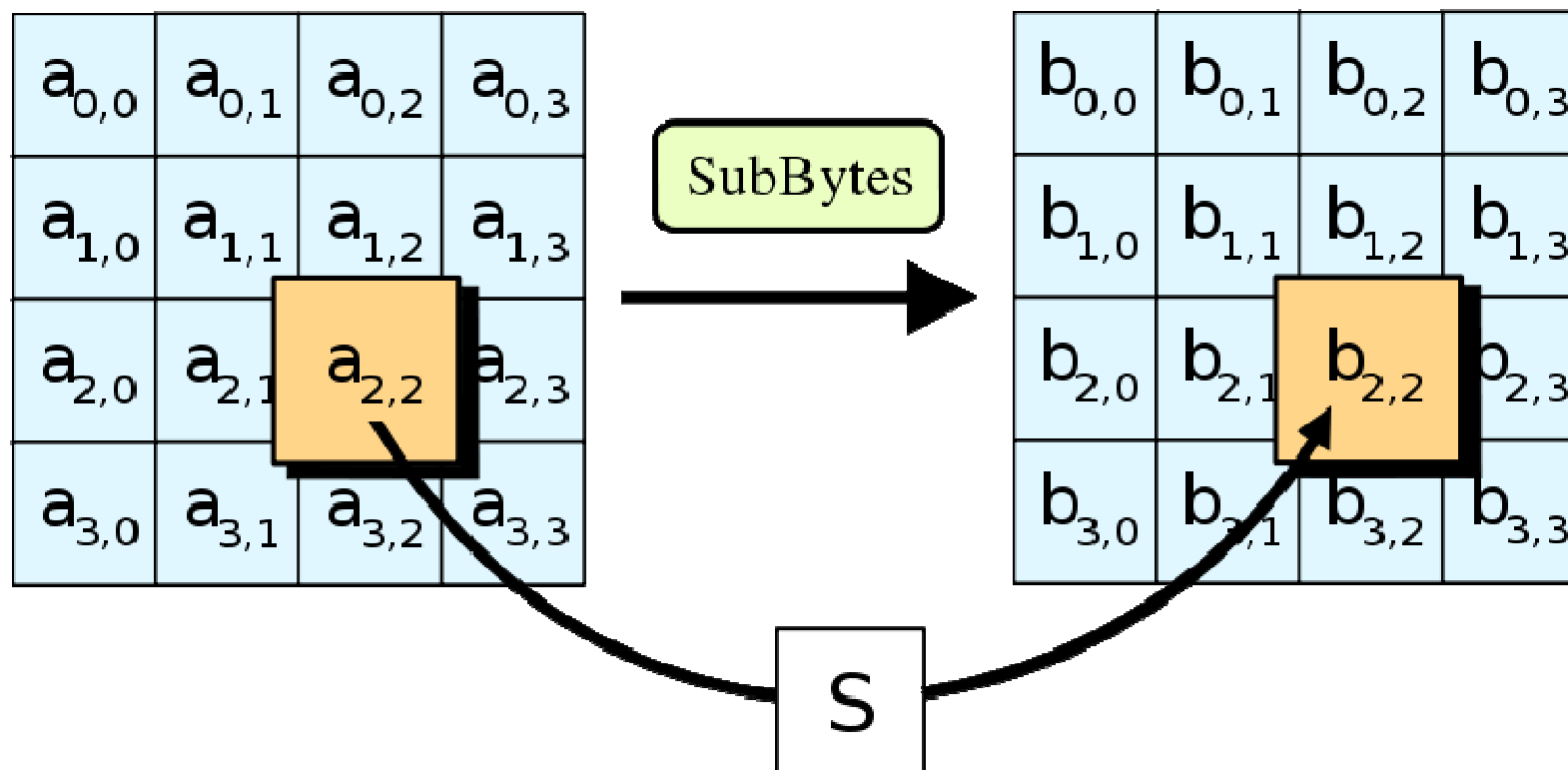
Where $R_{16}L_{16}$ is the permuted input block for deciphering calculation and L_0R_0 is the preoutput block

Key Schedule

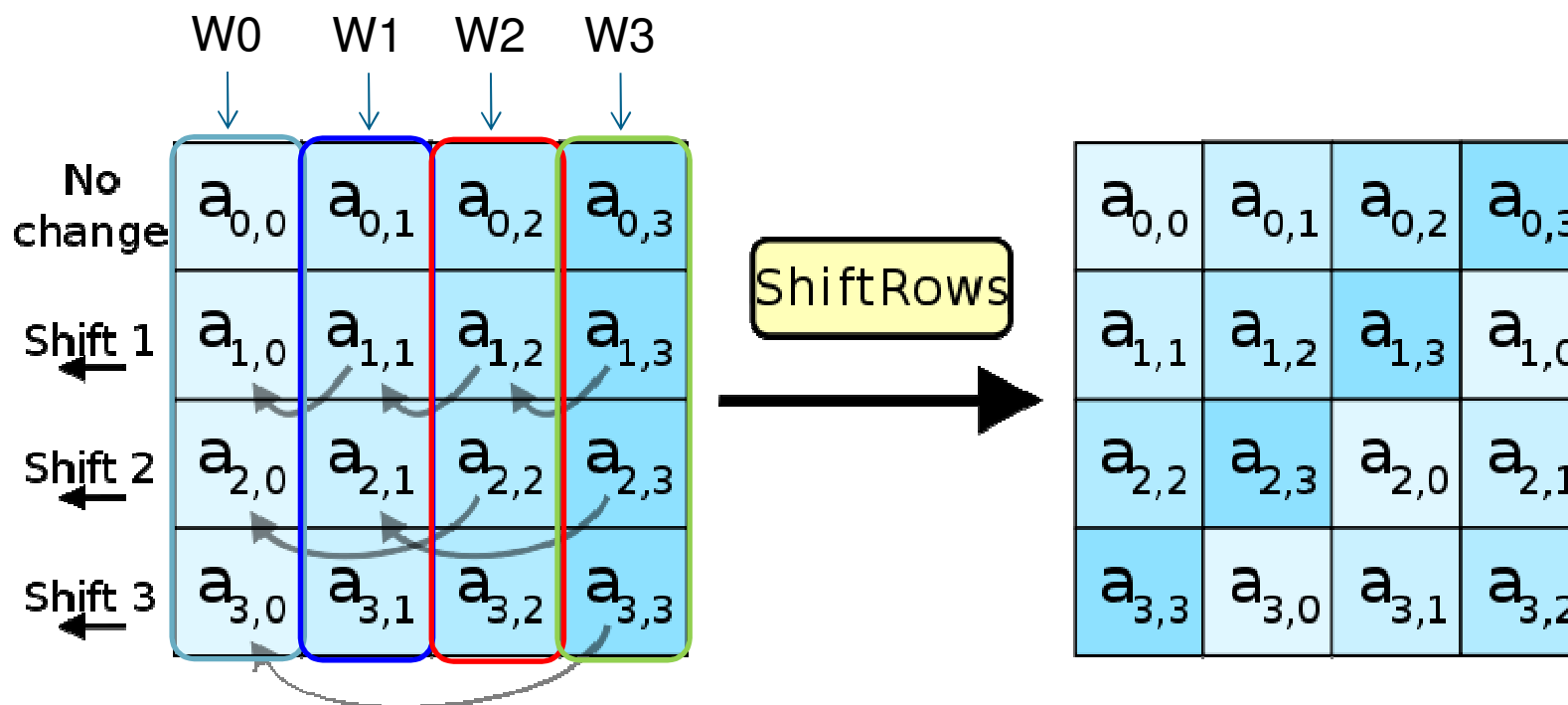


1. KeyExpansion—round keys are derived from the cipher key using Rijndael's key schedule
2. Initial Round
 - AddRoundKey—each byte of the state is combined with the round key using bitwise xor
3. Rounds
 - 1) SubBytes—a non-linear substitution step where each byte is replaced with another according to a lookup table.
 - 2) ShiftRows—a transposition step where each row of the state is shifted cyclically a certain number of steps.
 - 3) MixColumns—a mixing operation which operates on the columns of the state, combining the four bytes in each column.
 - 4) AddRoundKey
4. Final Round (no MixColumns)
 - 1) SubBytes
 - 2) ShiftRows
 - 3) AddRoundKey

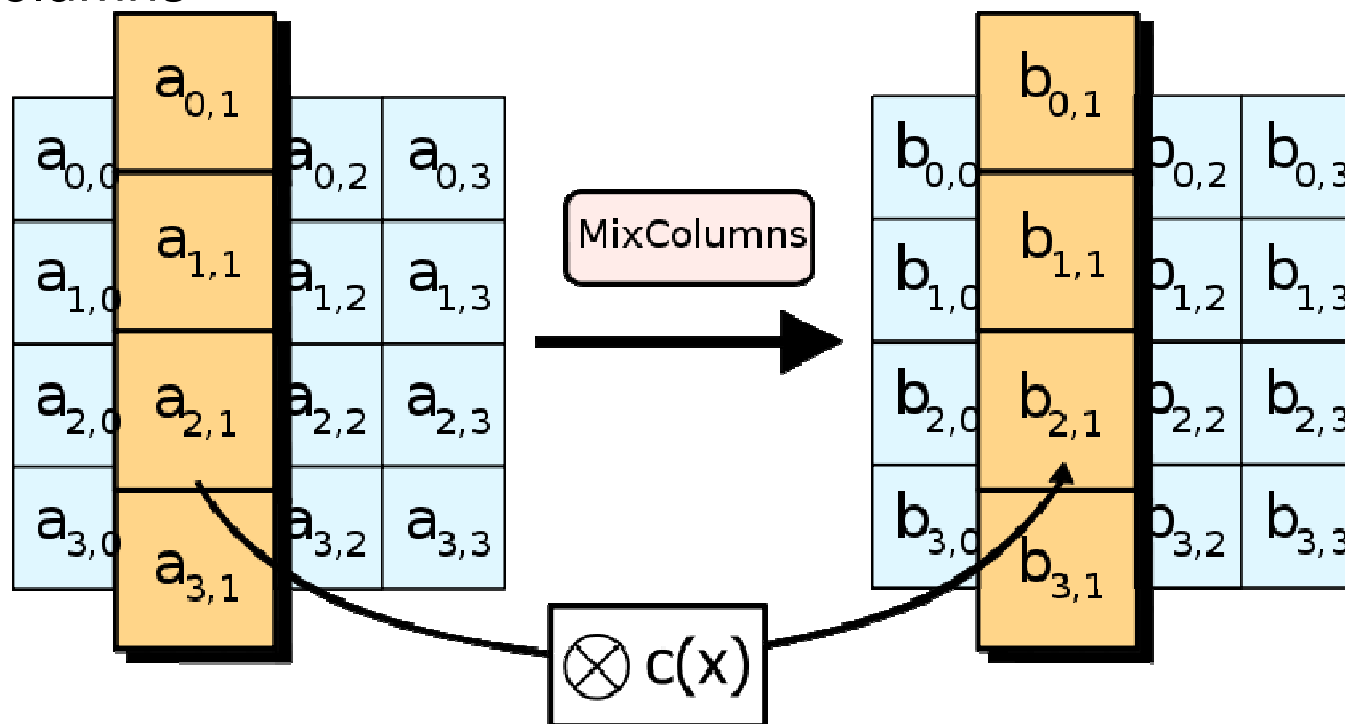
► SubBytes



► ShiftRows



► MixColumns



Each column is treated as a polynomial over $\mathbf{GF}(2^8)$ and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$;
the inverse of this polynomial is $c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$

► MixColumns

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

- $r_0 = 2a_0 + 3a_1 + 1a_2 + 1a_3$
- $r_1 = 1a_0 + 2a_1 + 3a_2 + 1a_3$
- $r_2 = 1a_0 + 1a_1 + 2a_2 + 3a_3$
- $r_3 = 3a_0 + 1a_1 + 1a_2 + 2a_3$

Note:

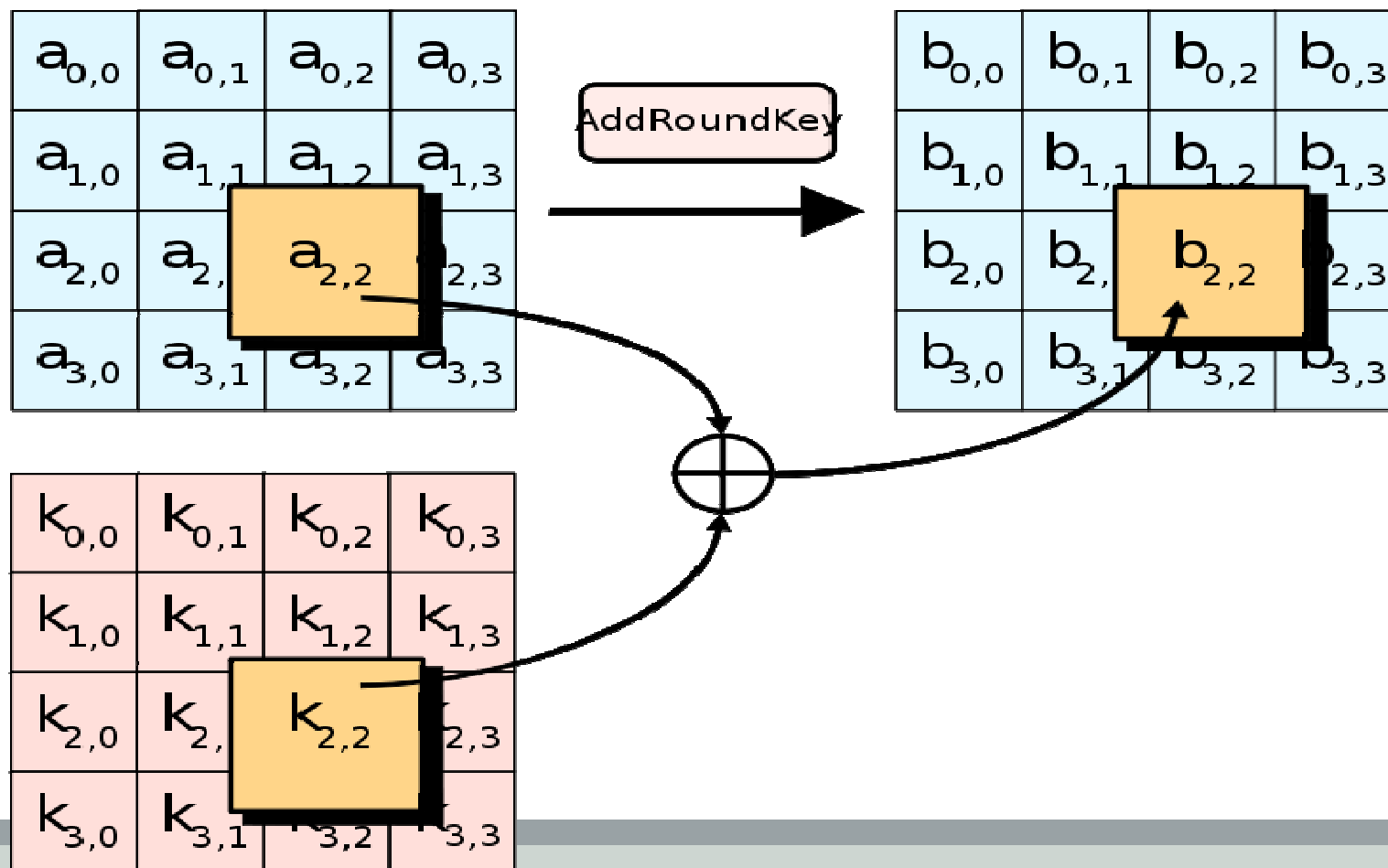
- Replace the multiply by 2 with a single shift and conditional exclusive or
- Replace a multiply by 3 with a multiply by 2 combined with an exclusive or

► InverseMixColumns

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

- $r_0 = 14a_0 + 9a_3 + 13a_2 + 11a_1$
- $r_1 = 14a_1 + 9a_0 + 13a_3 + 11a_2$
- $r_2 = 14a_2 + 9a_1 + 13a_0 + 11a_3$
- $r_3 = 14a_3 + 9a_2 + 13a_1 + 11a_0$

► AddRoundKey



► Step1: Append padding bits

- The message is padded to 448 mod 512 by adding single “1” bit and then “0” bits. At least one bit and at most 512 bits are appended.

► Step2: Append Length

- 64-bit representation of the message length is appended. Resulting message is divided into N blocks, each of 512 bits: M[0]...M[N-1]

► Step3: Initialize MD buffer

- 4-word buffer (A,B,C,D) used, each 32-bit register, initialized with the following values in hex:

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

► Step4: Process Message in 16-word blocks

```
/* Process each 16-word block. */
For i = 0 to N/16-1 do

    /* Copy block i into X. */
    For j = 0 to 15 do
        Set X[j] to M[i*16+j].
    end /* of loop on j */

    /* Save A as AA, B as BB, C as CC, and D as DD. */
    AA = A
    BB = B
```

MD5

```

CC = C
DD = D

/* Round 1. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

/* Round 2. */
/* Let [abcd k s i] denote the operation
   a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

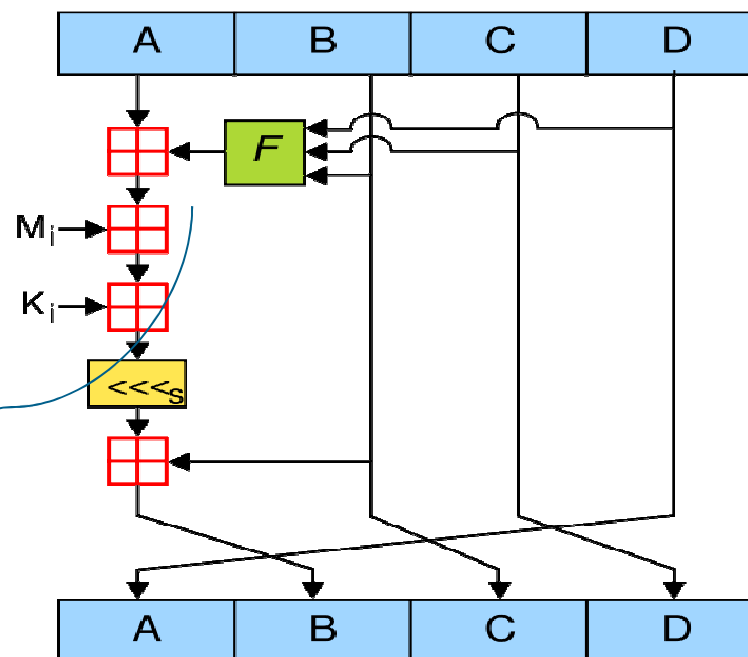
/* Round 3. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

/* Round 4. */
/* Let [abcd k s t] denote the operation
   a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

/* Then perform the following additions. (That is increment each
   of the four registers by the value it had before this block
   was started.) */
A = A + AA
B = B + BB
C = C + CC
D = D + DD

end /* of loop on i */

```



$$\begin{aligned}
 & \textcircled{1} F(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z) & \textcircled{3} H(X, Y, Z) = X \oplus Y \oplus Z \\
 & \textcircled{2} G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \neg Z) & \textcircled{4} I(X, Y, Z) = Y \oplus (X \vee \neg Z)
 \end{aligned}$$

► Step5: Output

- The message digest produced as output is A,B,C,D, beginning with lower-order byte of A, and end with the high-order byte of D.

► Preprocessing

- Step1: Pad the message, M similar to MD5
- Step2: Parse the padded message into N 512-bit message blocks:
 $M^{(1)} \dots M^{(N)}$
- Step3: Set the initial hash value, $H^{(0)}$

► Hash Computation

- After preprocessing, each message block is processed in order

For $i=1$ to N :

{

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leq t \leq 79 \end{cases}$$

- 2. Initialize the five working variables, ***a***, ***b***, ***c***, ***d***, and ***e***, with the (i-1)st hash value:

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$$c = H_2^{(i-1)}$$

$$d = H_3^{(i-1)}$$

$$e = H_4^{(i-1)}$$

- 3. For t=0 to 79:

$$T = ROTL^5(a) + f_t(b, c, d) + e + K_t + W_t$$

$$e = d$$

$$d = c$$

$$c = ROTL^{30}(b)$$

$$b = a$$

$$a = T$$

$$f_t(x, y, z) = \begin{cases} Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) & 0 \leq t \leq 19 \\ Parity(x, y, z) = x \oplus y \oplus z & 20 \leq t \leq 39 \\ Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \leq t \leq 59 \\ Parity(x, y, z) = x \oplus y \oplus z & 60 \leq t \leq 79. \end{cases}$$

- 4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

- Resulting 160-bit message digest of the message M is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)}$$

- ▶ A message schedule of 64 32-bit words, W_0, W_1, \dots, W_{63}
- ▶ 8 working variables of 32-bits each, labeled ***a, b, c, d, e, f, g*** and ***h***.
- ▶ The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \dots, H_7^{(i)}$
- ▶ Two temporary words T1 and T2
- ▶ Final result is 256-bit message digest
- ▶ SHA-256 Preprocessing similar to SHA1
- ▶ SHA-256 Hash Computation
 - For $i=1$ to N :
 - {
 - 1. Prepare the message schedule $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases}$$

$$\begin{aligned} \sigma_0^{\{256\}}(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\ \sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \end{aligned}$$

- 2. Initialize the eight working variables, a,b,c,d,e,f,g and h with the (i-1)st hash value:

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$$c = H_2^{(i-1)}$$

$$d = H_3^{(i-1)}$$

$$e = H_4^{(i-1)}$$

$$f = H_5^{(i-1)}$$

$$g = H_6^{(i-1)}$$

$$h = H_7^{(i-1)}$$

- 3. For $t = 0$ to 63

- {

$$T_1 = h + \sum_1^{\{256\}} (e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$$

$$T_2 = \sum_0^{\{256\}} (a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

- }

- 4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

$$H_5^{(i)} = f + H_5^{(i-1)}$$

$$H_6^{(i)} = g + H_6^{(i-1)}$$

$$H_7^{(i)} = h + H_7^{(i-1)}$$

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

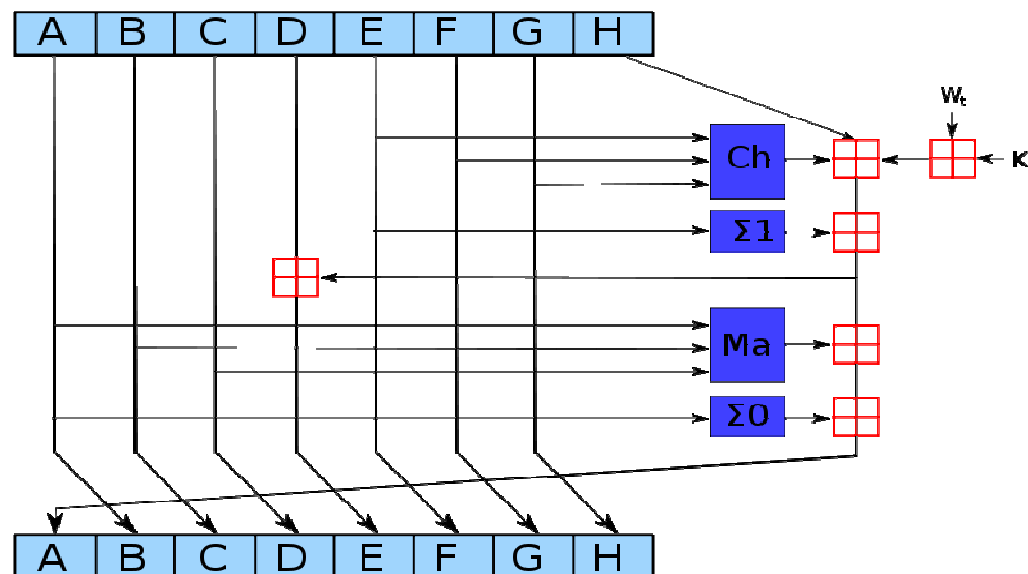
$$\sum_0^{\{256\}} (x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$

$$\sum_1^{\{256\}} (x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

- After repeating steps one through four a total of N times, the resulting 256-bit message digest of the message, M is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

► Core operation



$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Parity(x, y, z) = x \oplus y \oplus z$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

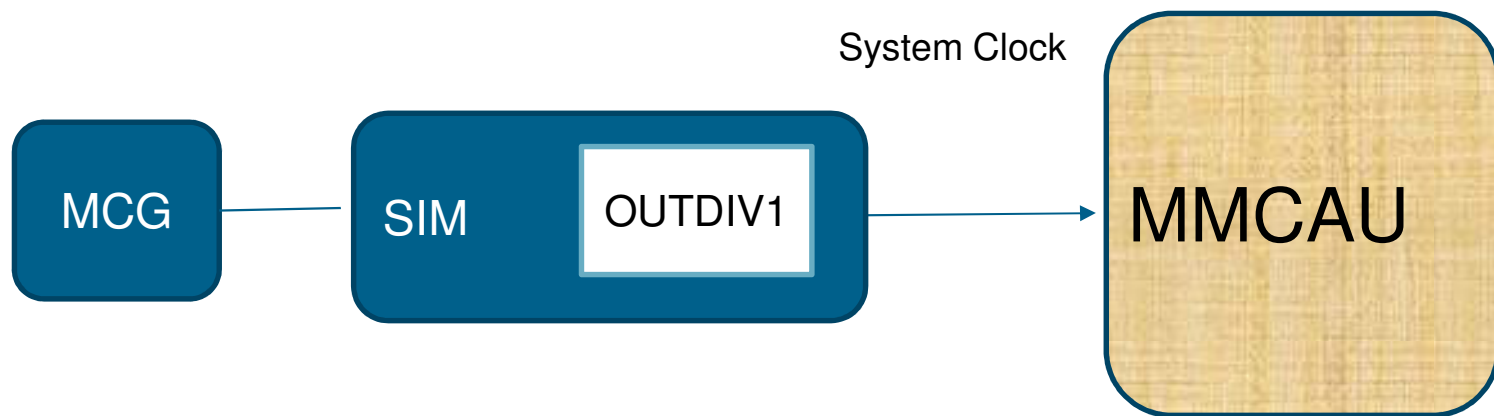
$$\Sigma 1(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$

$$\Sigma 0(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

MMCAU Training Outline

1. Module Overview
2. **On-chip interconnects and inter-module dependencies**
3. Software configuration
4. Demo code explanation
5. Frequently asked question list
6. Reference material

SoC interconnect diagram



Module dependencies

- ▶ *Clock source setup*
 - *No clock gating is required*
- ▶ *I/O configuration*
 - *No I/O*

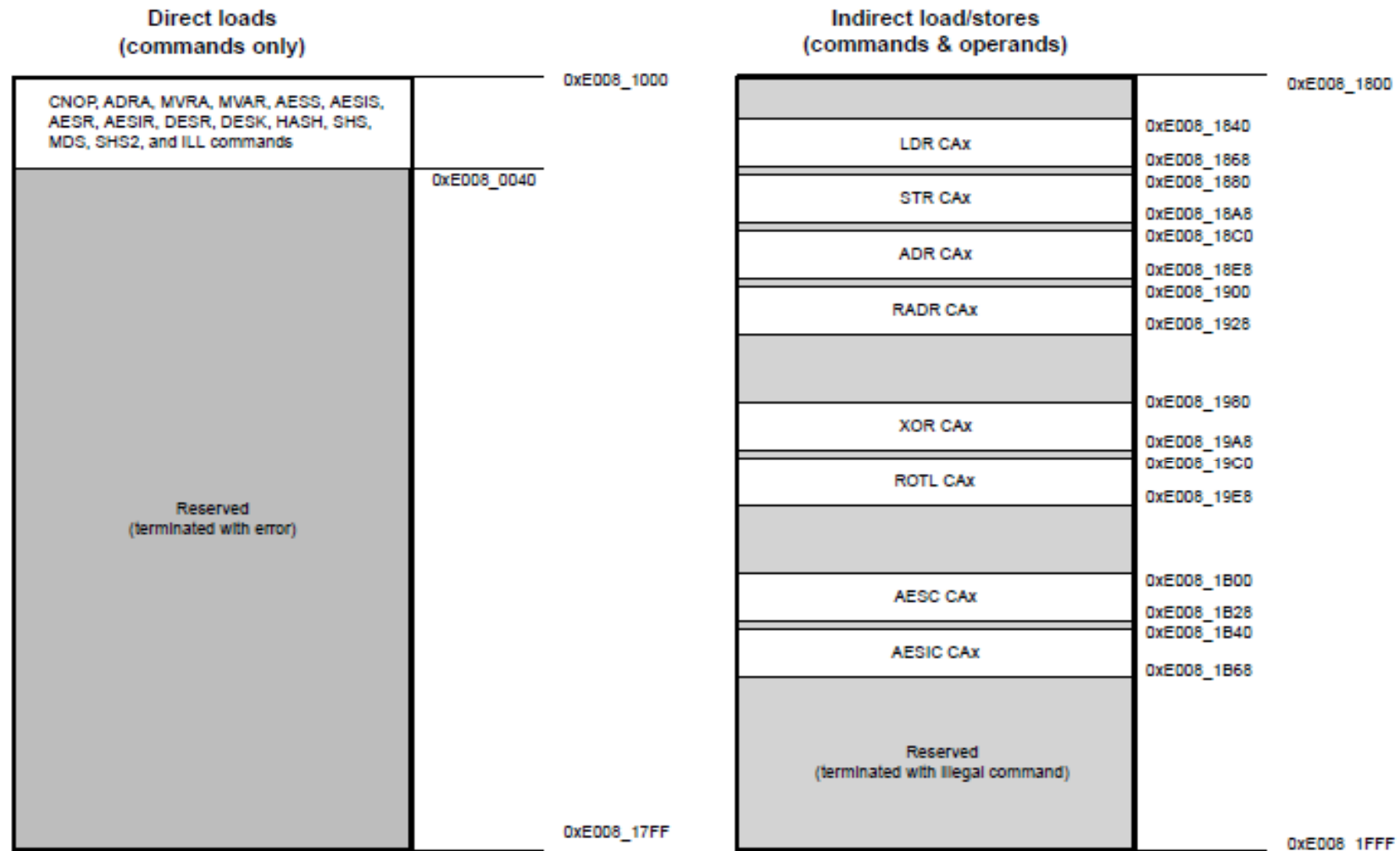
MMCAU Training Outline

1. Module Overview
2. On-chip interconnects and inter-module dependencies
3. **Software configuration**
4. Demo code explanation
5. Frequently asked question list
6. Reference material

Memory Map

Address offset (hex)	Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
0	E008_1000	Status Register (CAU_CASR)	32	R/W	2000_0000h	32.5.1/720
1	E008_1001	Accumulator (CAU_CAA)	32	R/W	0000_0000h	32.5.2/721
2	E008_1002	General Purpose Register (CAU_CA0)	32	R/W	0000_0000h	32.5.3/721
3	E008_1003	General Purpose Register (CAU_CA1)	32	R/W	0000_0000h	32.5.3/721
4	E008_1004	General Purpose Register (CAU_CA2)	32	R/W	0000_0000h	32.5.3/721
5	E008_1005	General Purpose Register (CAU_CA3)	32	R/W	0000_0000h	32.5.3/721
6	E008_1006	General Purpose Register (CAU_CA4)	32	R/W	0000_0000h	32.5.3/721
7	E008_1007	General Purpose Register (CAU_CA5)	32	R/W	0000_0000h	32.5.3/721
8	E008_1008	General Purpose Register (CAU_CA6)	32	R/W	0000_0000h	32.5.3/721
9	E008_1009	General Purpose Register (CAU_CA7)	32	R/W	0000_0000h	32.5.3/721
A	E008_100A	General Purpose Register (CAU_CA8)	32	R/W	0000_0000h	32.5.3/721

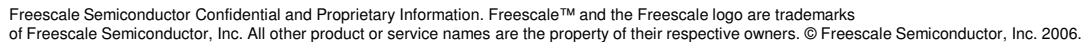
Memory Map



Directly write only commands for CAU load operation

Send both commands and operands for CAU load operation

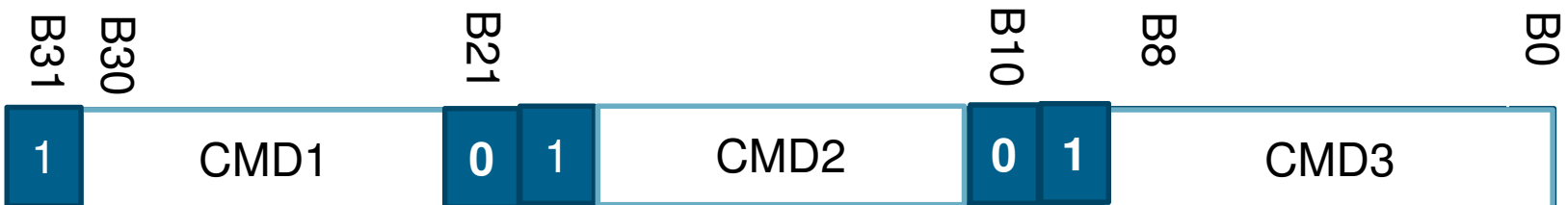
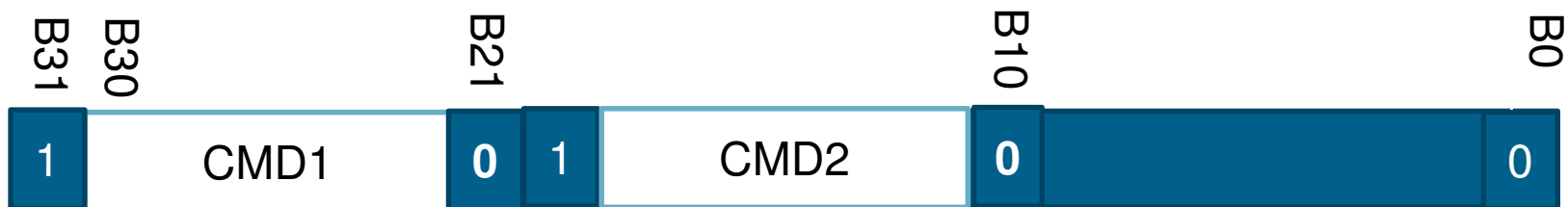
- ▶ CPU data write on PPB initiates CAU load operands
- ▶ CPU data read on PPB initiates CAU store register operations
- ▶ CAU requires 15-bit command (+32-bit operand) for each CAU load (CPU data write):



Programming Model

► Direct Loads

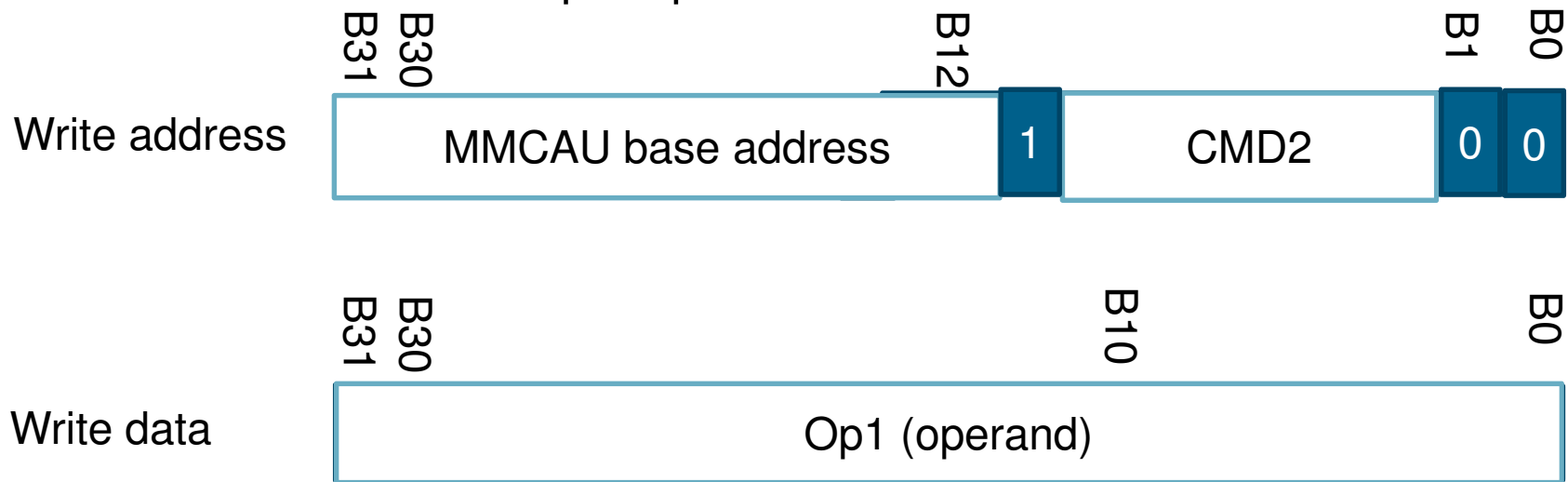
- Support writing up to 3 commands in each 32-bit direct write operation



Programming Model

► Indirect Loads

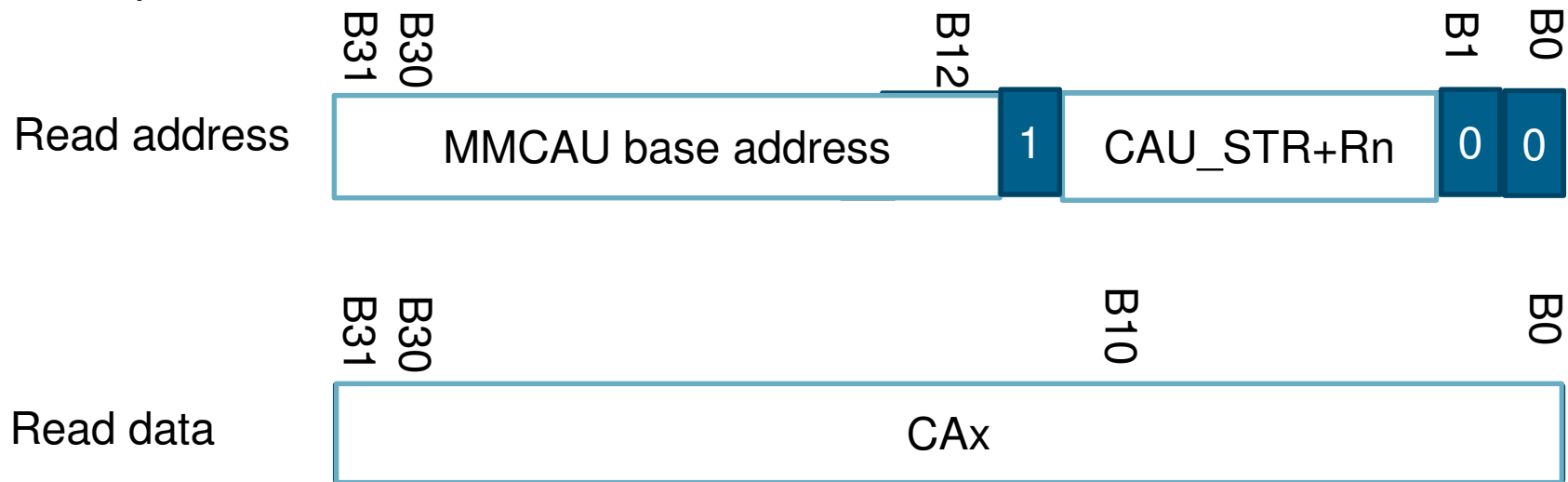
- Write address consists of MMCAU base address and CAU command
- write data is 32-bit input operand



Programming Model

► Indirect Stores

- Read address consists of MMCAU base address and CAU store register opcode and data will be read from CAx



NOTE:

CAU store register opcode:

CAU_STR+Rn = Addr[11:2] = 100010xxxx

CAU Register Coding

► CAU register coding

Code	Register	DES	AES	MD5	SHA-1	SHA-256
0	CAU status register (CASR)	—	—	—	—	—
1	CAU accumulator (CAA)	—	—	a	T	T
2	General purpose register 0 (CA0)	C	W0	—	A	A
3	General purpose register 1 (CA1)	D	W1	b	B	B
4	General purpose register 2 (CA2)	L	W2	c	C	C
5	General purpose register 3 (CA3)	R	W3	d	D	D
6	General purpose register 4 (CA4)	—	—	—	E	E
7	General purpose register 5 (CA5)	—	—	—	W	F
8	General purpose register 6 (CA6)	—	—	—	—	G
9	General purpose register 7 (CA7)	—	—	—	—	H
10	General purpose register 8 (CA8)	—	—	—	—	W/T ₁

CAU Commands

Type	Command Name	Description	CMD								Operation
			8	7	6	5	4	3	2	1	0
Direct load	CNOP	No Operation	0x000								—
Indirect load	LDR	Load Reg	0x01				CAx				Op1 → CAx
Indirect store	STR	Store Reg	0x02				CAx				CAx → Result
Indirect load	ADR	Add	0x03				CAx				CAx + Op1 → CAx
Indirect load	RADR	Reverse and Add	0x04				CAx				CAx + ByteRev(Op1) → CAx
Direct load	ADRA	Add Reg to Acc	0x05				CAx				CAx + CAA → CAA
Indirect load	XOR	Exclusive Or	0x06				CAx				CAx ^ Op1 → CAx
Indirect load	ROTL	Rotate Left	0x07				CAx				(CAx <<< (Op1 % 32)) (CAx >>> (32 - (Op1 % 32))) → CAx
Direct load	MVRA	Move Reg to Acc	0x08				CAx				CAx → CAA

CAU Commands

Type	Command Name	Description	CMD										Operation
			8	7	6	5	4	3	2	1	0		
Direct load	MVAR	Move Acc to Reg	0x09					CAx					CAA → CAx
Direct load	AESS	AES Sub Bytes	0x0A					CAx					SubBytes(CAx) → CAx
Direct load	AESIS	AES Inv Sub Bytes	0x0B					CAx					InvSubBytes(CAx) → CAx
Indirect load	AESC	AES Column Op	0x0C					CAx					MixColumns(CAx)^Op1→CAx
Indirect load	AESIC	AES Inv Column Op	0x0D					CAx					InvMixColumns(CAx^Op1) → CAx
Direct load	AESR	AES Shift Rows	0x0E0										ShiftRows(CA0-CA3) → CA0-CA3
Direct load	AESIR	AES Inv Shift Rows	0x0F0										InvShiftRows(CA0-CA3)→CA0-CA3
Direct load	DESR	DES Round	0x10					IP	FP	KS[1:0]			DES Round(CA0-CA3)→CA0-CA3
Direct load	DESK	DES Key Setup	0x11					0	0	CP	DC		DES Key Op(CA0-CA1)→CA0-CA1 Key Parity Error & CP → CASR[1]
Direct load	HASH	Hash Function	0x12					0	HF[2:0]			Hash Func(CA1-CA3)+CAA→CAA	
Direct load	SHS	Secure Hash Shift	0x130										CAA <<< 5→CAA, CAA→CA0, CA0→CA1, CA1 <<< 30 → CA2, CA2→CA3, CA3→CA4
Direct load	MDS	Message Digest Shift	0x140										CA3→CAA, CAA→CA1, CA1→CA2, CA2→CA3,
Direct load	SHS2	Secure Hash Shift 2	0x150										CAA→CA0, CA0→CA1, CA1 → CA2, CA2→CA3, CA3 + CA8 →CA4, CA4 → CA5, CA5 → CA6, CA6 → CA7
Direct load	ILL	Illegal Command	0x1F0										0x1→CASR[IC]

CAU Commands vs Security Algorithms

CAU Commands	Algorithm Function	Structure
AES Substitution	SubBytes:	CA0=W0;CA1=W1;CA2=W2;CA3=W3
AES Column Operation	MixColumns:	CA0=W0;CA1=W1;CA2=W2;CA3=W3
AES Shift Rows	ShiftRows:	CA0=W0;CA1=W1;CA2=W2;CA3=W3
DES Round	One round of the DES + a key schedule update	CA0=C;CA1=D;CA2=L;CA3=R
DES Key Setup	Permuted Choice1 (+ left shift for decrypt)	CA0=C;CA1=D;
Hash Command HFF	MD5 F(X,Y,Z) : (X & Y) (~X & Z)	X=CA1,Y=CA2,Z=CA3

CAU Commands vs Security Algorithms

CAU Commands	Algorithm Function	Structure
Hash Command HFF	MD5 $F(X,Y,Z) :$ $(X \& Y) (\sim X \& Z)$	$X=CA1, Y=CA2, Z=CA3$
Hash Command HFG	MD5 $G(X,Y,Z) :$ $(X \& Z) (Y \& \sim Z)$	$X=CA1, Y=CA2, Z=CA3$
Hash Command HFH	MD5 $H(X,Y,Z) :$ $(X \wedge Y \wedge Z)$	$X=CA1, Y=CA2, Z=CA3$
Hash Command HFI	MD5 $I(X,Y,Z) :$ $Y \wedge (X \sim Z)$	$X=CA1, Y=CA2, Z=CA3$
Hash Command HFC	SHA $Ch(b,c,d) :$ $(b \& c) \wedge (\sim b \& d)$	$T=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, W=CA5$
Hash Command HFM	SHA $Maj(b,c,d) :$ $(b \& c) \wedge (b \& d) \wedge (c \& d)$	$T=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, W=CA5$

CAU Commands vs Security Algorithms

CAU Commands	Algorithm Function	Structure
Hash Command HF2C	SHA -256 $Ch(e,f,g) :$ $(e \& f) \wedge (\sim e \& g)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6,$ $h=CA7, T1/W= CA8$
Hash Command HF2M	SHA-256 $Maj(a,b,c) :$ $(a \& b) \wedge (a \& c) \wedge (b \& c)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6, h=$ $CA7, T1/W= CA8$
Hash Command HF2S	SHA-256 $\sum 0(a) :$ $ROTR^2(a) \wedge ROTR^{13}(a) \wedge ROTR^{22}(a)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6, h=$ $CA7, T1/W= CA8$
Hash Command HF2T	SHA-256 $\sum 1(e) :$ $ROTR^6(e) \wedge ROTR^{11}(e) \wedge ROTR^{25}(e)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6, h=$ $CA7, T1/W= CA8$
Hash Command HF2U	SHA-256 $\sigma 0(e) :$ $ROTR^7(e) \wedge ROTR^{18}(e) \wedge ROTR^3(e)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6, h=$ $CA7, T1/W= CA8$
Hash Command HF2V	SHA-256 $\sigma 1(W) :$ $ROTR^6(W) \wedge ROTR^{11}(W) \wedge$ $ROTR^{25}(W)$	$T2=CAA, a=CA0, b=CA1, c=CA2,$ $d=CA3, e=CA4, f=CA5, g=CA6, h=$ $CA7, T1/W= CA8$

CAU Commands vs Security Algorithms

CAU Commands	Algorithm Function	Structure
Secure Hash Shift	SHA -1: parallel register-to-register move and shift operation	T=CAA,a=CA0,b=CA1,c=CA2,d=CA3,e=CA4,W=CA5
Message Digest Shift	MD5: parallel register-to-register move operations	a=CAA,b=CA1,c=CA2,d=CA3
Secure Hash Shift2	SHA-256: perform an addition (+) and a set of register-to-register moves in parallel	T2=CAA,a=CA0,b=CA1,c=CA2,d=CA3,e=CA4,f=CA5,g=CA6,h=CA7,T1/W= CA8

Register	Value prior to command	Value after command executes
CA4	CA4	CA3
CA3	CA3	CA2
CA2	CA2	CA1<<<30
CA1	CA1	CA0
CA0	CA0	CAA
CAA	CAA	CAA<<<5

Register	Value prior to command	Value after command executes
CA3	CA3	CA2
CA2	CA2	CA1
CA1	CA1	CAA
CAA	CAA	CA3

Register	Value prior to command	Value after command executes
CA7	CA7	CA6
CA6	CA6	CA5
CA5	CA5	CA4
CA4	CA4	CA3+CA8
CA3	CA3	CA2
CA2	CA2	CA1
CA1	CA1	CA0
CA0	CA0	CAA

MMCAU Training Outline

1. Module Overview
2. On-chip interconnects and inter-module dependencies
3. Software configuration
4. **Demo code explanation**
5. Frequently asked question list
6. Reference material

Demo Code Explanation

```
//*****
//
// AES: Encrypts a single 16-byte block
// arguments
//   *in      pointer to 16-byte block of input plaintext
//   *key_sch pointer to key schedule (44, 52, 60 longwords)
//   nr       number of AES rounds (10, 12, 14 = f(key_schedule))
//   *out     pointer to 16-byte block of output ciphertext
//
// NOTE   Input and output blocks may overlap
//
// calling convention
// int  cau_aes_encrypt (const unsigned char *in,
//                       const unsigned char *key_sch,
//                       const int          nr,
//                       unsigned char      *out)

void mmcau_aes_encrypt (unsigned int in[], unsigned int key_sch[], int nr, unsigned int
out[])
{
    int i,j;
// load the 4 plain test bytes into the CAU's CA0 - CA3 registers
    *(MMCAU_PPB_INDIRECT + (LDR+CA0)) = in[0]; // load in[0] -> CA0
    *(MMCAU_PPB_INDIRECT + (LDR+CA1)) = in[1]; // load in[1] -> CA1
    *(MMCAU_PPB_INDIRECT + (LDR+CA2)) = in[2]; // load in[2] -> CA2
    *(MMCAU_PPB_INDIRECT + (LDR+CA3)) = in[3]; // load in[3] -> CA3
```

Demo Code Explanation

```
// XOR the first 4 keys into the CAU's CA0 - CA3 registers
*(MMCAU_PPB_INDIRECT + (XOR+CA0)) = key_sch[0]; // XOR keys
*(MMCAU_PPB_INDIRECT + (XOR+CA1)) = key_sch[1];
*(MMCAU_PPB_INDIRECT + (XOR+CA2)) = key_sch[2];
*(MMCAU_PPB_INDIRECT + (XOR+CA3)) = key_sch[3];

// send a series of cau commands to perform the encryption
for (i = 0, j = 4; i < nr - 1 ; i++, j+=4)
{
    *(MMCAU_PPB_DIRECT) = mmcau_3_cmds(AESS+CA0,AESS+CA1,AESS+CA2); // SubBytes
    *(MMCAU_PPB_DIRECT) = mmcau_2_cmds(AESS+CA3,AESR);           // SubBytes, ShiftRows
    *(MMCAU_PPB_INDIRECT + (AESC+CA0)) = key_sch[j];             // MixColumns
    *(MMCAU_PPB_INDIRECT + (AESC+CA1)) = key_sch[j+1];
    *(MMCAU_PPB_INDIRECT + (AESC+CA2)) = key_sch[j+2];
    *(MMCAU_PPB_INDIRECT + (AESC+CA3)) = key_sch[j+3];
}

*(MMCAU_PPB_DIRECT) = mmcau_3_cmds(AESS+CA0,AESS+CA1,AESS+CA2); // SubBytes
*(MMCAU_PPB_DIRECT) = mmcau_2_cmds(AESS+CA3,AESR);           // SubBytes, ShiftRows
*(MMCAU_PPB_INDIRECT + (XOR+CA0)) = key_sch[j];             // XOR keys
*(MMCAU_PPB_INDIRECT + (XOR+CA1)) = key_sch[j+1];
*(MMCAU_PPB_INDIRECT + (XOR+CA2)) = key_sch[j+2];
*(MMCAU_PPB_INDIRECT + (XOR+CA3)) = key_sch[j+3];

// store the 16-byte ciphertext output block into memory
out[0] = *(MMCAU_PPB_INDIRECT + (STR+CA0)); // store 1st 4 bytes of ciphertext
out[1] = *(MMCAU_PPB_INDIRECT + (STR+CA1)); // store 2nd 4 bytes of ciphertext
out[2] = *(MMCAU_PPB_INDIRECT + (STR+CA2)); // store 3rd 4 bytes of ciphertext
out[3] = *(MMCAU_PPB_INDIRECT + (STR+CA3)); // store 4th 4 bytes of ciphertext
}
```

Demo Code Explanation

► Some macro definitions

```
#define MMCAU_PPB_DIRECT      (volatile uint32*) 0xE0081000 // CAU Starting Address
#define MMCAU_PPB_INDIRECT   (volatile uint32*) 0xE0081800

/* CAU commands, for CMD[8:4] */
/* Load Reg */
#define LDR    (0x010)

/* Store Reg */
#define STR    (0x020)

/* registers */
#define CASR    0x0
#define CAA     0x1
#define CA0     0x2
#define CA1     0x3
#define CA2     0x4
#define CA3     0x5
#define CA4     0x6
#define CA5     0x7
#define CA6     0x8
#define CA7     0x9
#define CA8     0xa
```

MMCAU Training Outline

1. Module Overview
2. On-chip interconnects and inter-module dependencies
3. Software configuration
4. Demo code explanation
5. **Frequently asked question list**
6. Reference material

MMCAU Training Outline

1. Module Overview
2. On-chip interconnects and inter-module dependencies
3. Software configuration
4. Demo code explanation
5. Frequently asked question list
6. **Reference material**