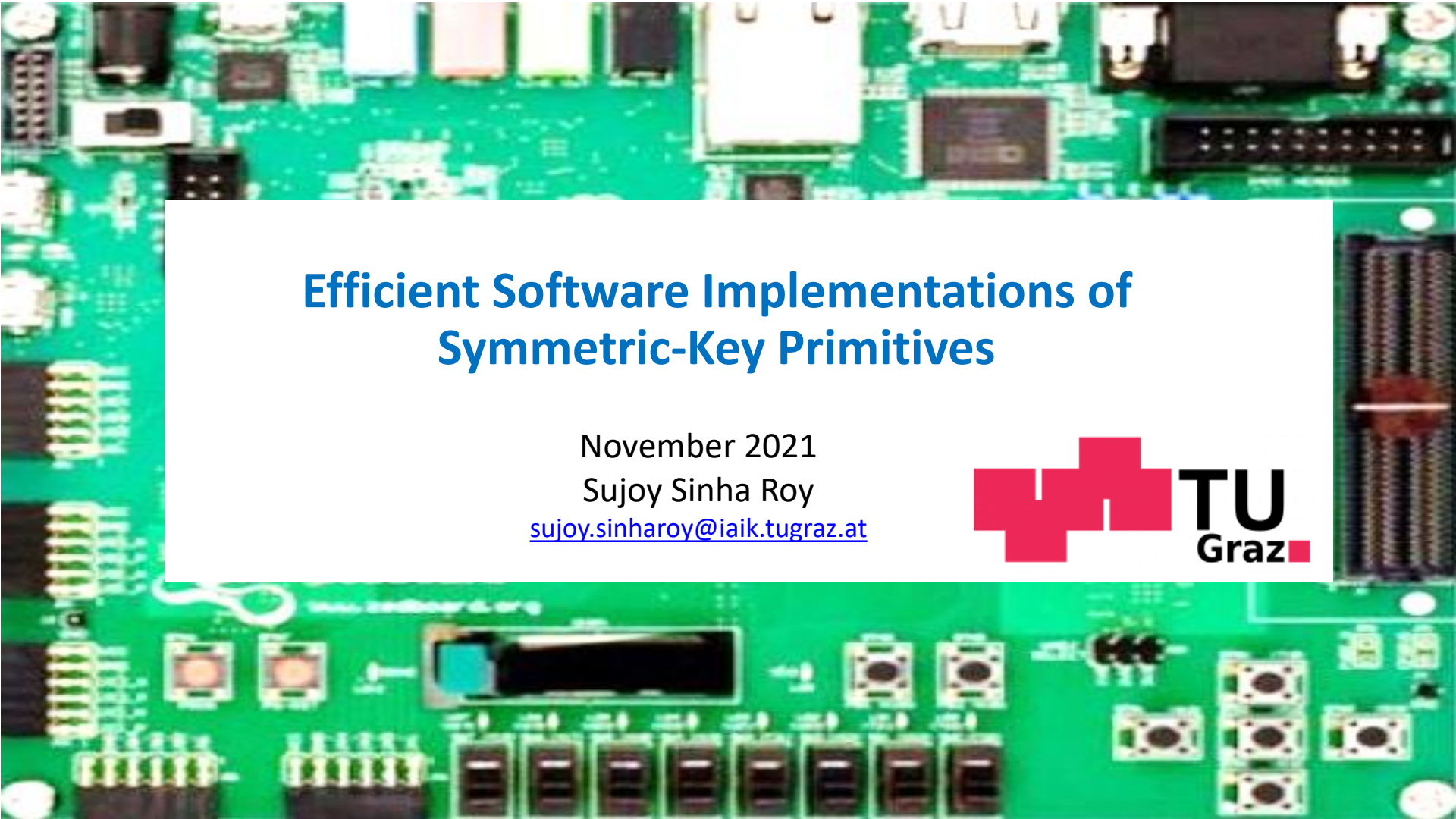


# Efficient Software Implementations of Symmetric-Key Primitives

November 2021

Sujoy Sinha Roy
sujoy.sinharoy@iaik.tugraz.at

TU Graz

# Exercise: Compute Hamming Weight efficiently

Ex1:

```
int HammingWeight(unsigned int v)
{
        // Code for Hamming-Weight of 32-bit unsigned int v
}
```

# Hamming Weight: Solution 1 (Serial method)

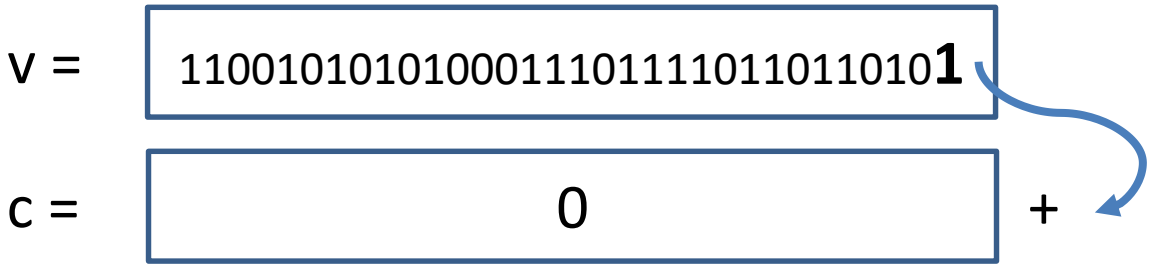
```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
        c += v&1;
  }
  return c;
}
```

v = 11001010101000111011110110110101

c = 0

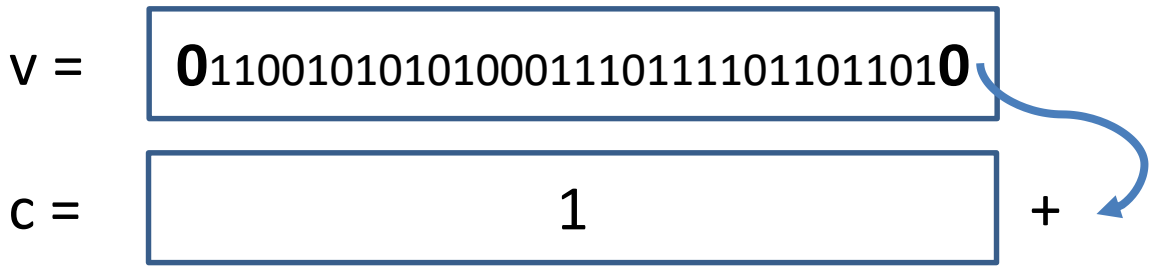
# Hamming Weight: Solution 1 (Serial method)

```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
        c += v&1;
  }
  return c;
}
```



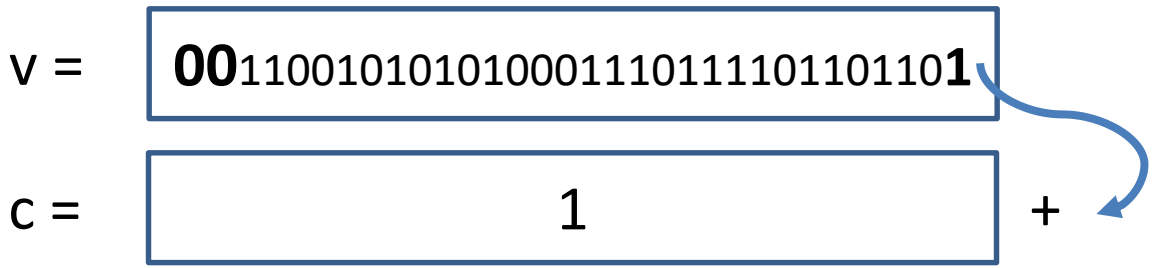
# Hamming Weight: Solution 1 (Serial method)

```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
        c += v&1;
  }
  return c;
}
```



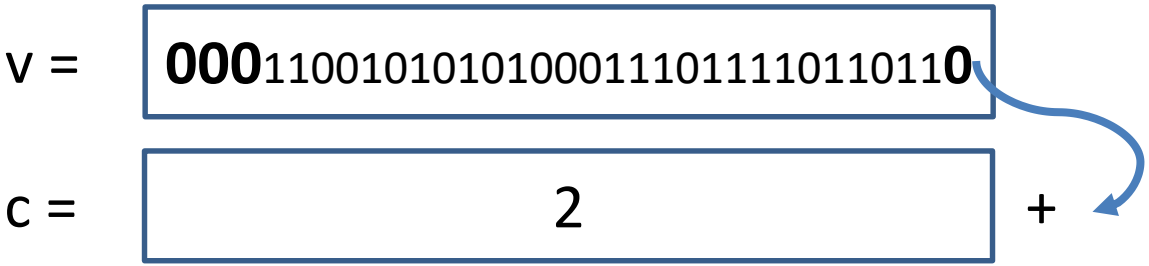
# Hamming Weight: Solution 1 (Serial method)

```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
        c += v&1;
  }
  return c;
}
```



# Hamming Weight: Solution 1 (Serial method)

```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
        c += v&1;
  }
  return c;
}
```



# Hamming Weight: Solution 1 (Serial method)

```
int HammingWeight(unsigned int v)
{
  unsigned int c;
  for(c = 0; v; v>>=1)
  {
      c += v&1;
  }
  return c;
}
```

2 operations

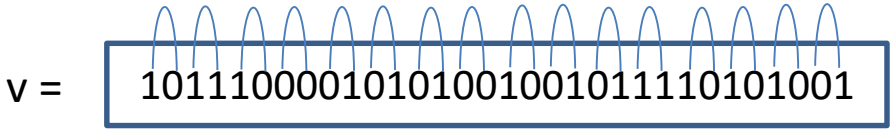
2 operations

Total cost for computing Hamming weight of 32-bit v

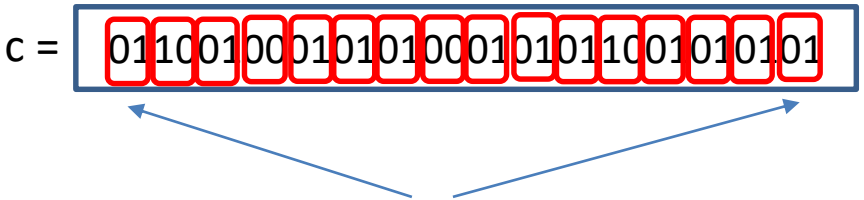
≈ 32*(2+2)

≈ 128 operations

# Hamming Weight: Solution 2 (Parallel method)




Add two consecutive bits.




We will create 16 ‘boxes’.

Each red box contains sum of two consecutive bits in it

# Hamming Weight: Solution 2 (Parallel method)

v = 10111000010101001001011110101001

Add two consecutive bits.

1. Choose only the bits in the **odd** positions and set all others to 0

$temp_1$ = v & 0x55555555

= 00010000010101000001010100000001

Note: 0x55555555 = $(0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101)_2$

# Hamming Weight: Solution 2 (Parallel method)

v = 10111000010101001001011110101001

Add two consecutive bits.

1. Choose only the bits in the **odd** positions and set all others to 0.
2. Choose only the bits in the **even** positions and set all others to 0.

$temp_2$ = (v>>1) & 0x55555555

= 01010100000000000100000101010100

# Hamming Weight: Solution 2 (Parallel method)

v = 10111000010101001001011110101001

Add two consecutive bits.

1. Choose only the bits in the **odd** positions and set all others to 0.
2. Choose only the bits in the **even** positions and set all others to 0.
3. Now compute c = $temp_1$ + $temp_2$

$temp_1$ = v & 0x55555555

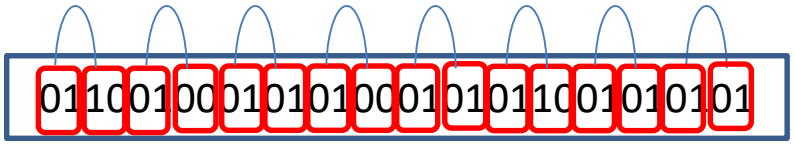
= 00010000010101000001010100000001

$temp_2$ = (v>>1) & 0x55555555

= 01010100000000000100000101010100

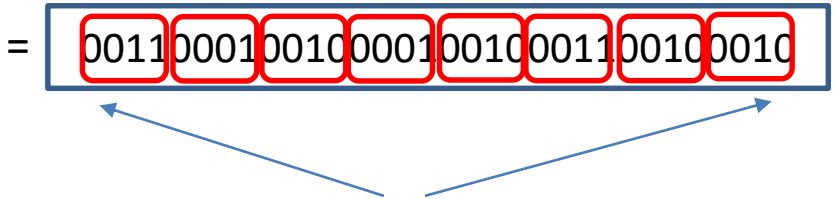
c = 01100100010101000101011001010101

# Hamming Weight: Solution 2 (Parallel method)




Next, add two consecutive boxes together.

```
c = (c>>2)&0x33333333 + c&0x33333333;
```




Each red box contains sum of four consecutive bits of v

… continue it in this manner.

# Hamming Weight: Solution 2 (Parallel method) in C

```
unsigned int HammingWeight(unsigned int v)
{
        unsigned int c; // stores Hamming weight
        static const int S[]={1, 2, 4, 8, 16};
        static const int B[]={0x55555555, 0x33333333,
                0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF};

        c = ((v >> S[0]) & B[0]) + (v & B[0]);
        c = ((c >> S[1]) & B[1]) + (c & B[1]);
        c = ((c >> S[2]) + c) & B[2];
        c = ((c >> S[3]) + c) & B[3];
        c = ((c >> S[4]) + c) & B[4];

        return c;
}
```

**Number of operations = 17**

**(Down from 128 in the sequential implementation)**

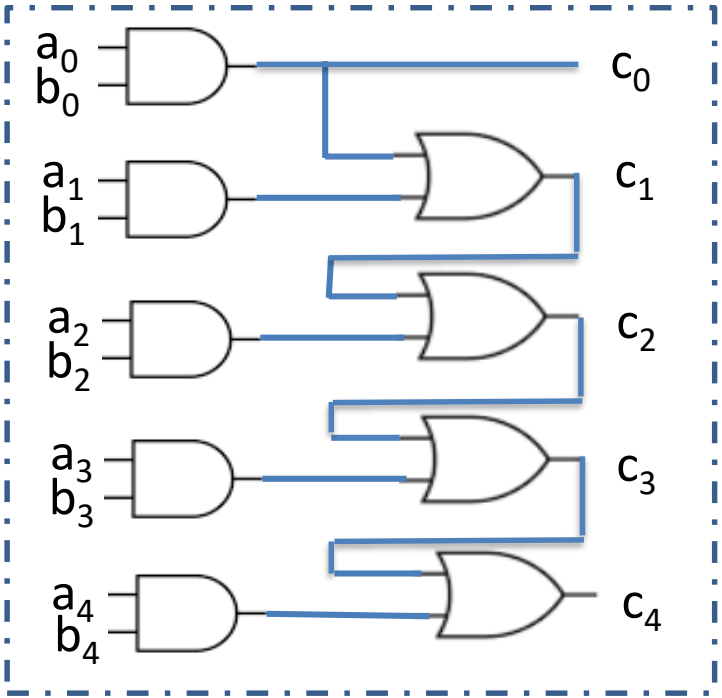
# What do we learn from the example?

1. Bit manipulations are tricky on SW platforms
   (On HW platforms they are pretty simple: just wire selections)

2. Efficiency depends greatly on the quality of the code
   (An efficient code uses word-arithmetic optimally)

# Bitslicing: A technique for data orthogonalization

# Motivating example: Evaluation of a circuit in *software*




We have a gate-level description of the above 5-bit circuit.

Goal: We want to evaluate this ‘circuit’ in SW on **many** 5-bit inputs.

# Naïve circuit evaluation in SW

**Computation cost per iteration =**

- Extracting bits from A and B
  - 2 op. per bit, total 2*10 = 20
  - Constant cost

- Circuit evaluation
  - 5 AND + 4 OR = 9 gate evaluations
  - **Increases with circuit complexity**

  (E.g. circuit with 1 million gates)

- Packing result bits
  - Constant cost

```
…
unsigned int a0, a1, a2, a3, a4;
unsigned int b0, b1, b2, b3, b4;

for(int i=0; i<NUM_INPUTS; i++){
        // Extract the bits of A[i]
        a0=A[i]&1; a1=(A[i]>>1)&1; …
        // Extract the bits of B[i]
        b0=B[i]&1; b1=(B[i]>>1)&1; …

        // Evaluate the circuit
        c0 = a0&b0;
        c1 = (a1&b1) | c0;
        c2 = (a2&b2) | c1;
        c3 = (a3&b3) | c2;
        c4 = (a4&b4) | c3;

        // Pack c-bits into C[]
        C[i] = c0|(c1<<1)|(c2<<2)|(c3<<3)|(c4<<4);
}
```

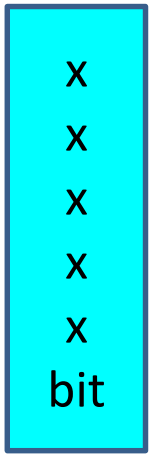
# Computational inefficiency of the code

Consider a circuit with n gates

- Each bit is represented using one **int** data-type

Let there are n gates.

```
// Evaluate the circuit
c0 = a0&b0;
c1 = (a1&b1) | c0;
c2 = (a2&b2) | c1;
c3 = (a3&b3) | c2;
c4 = (a4&b4) | c3;
```



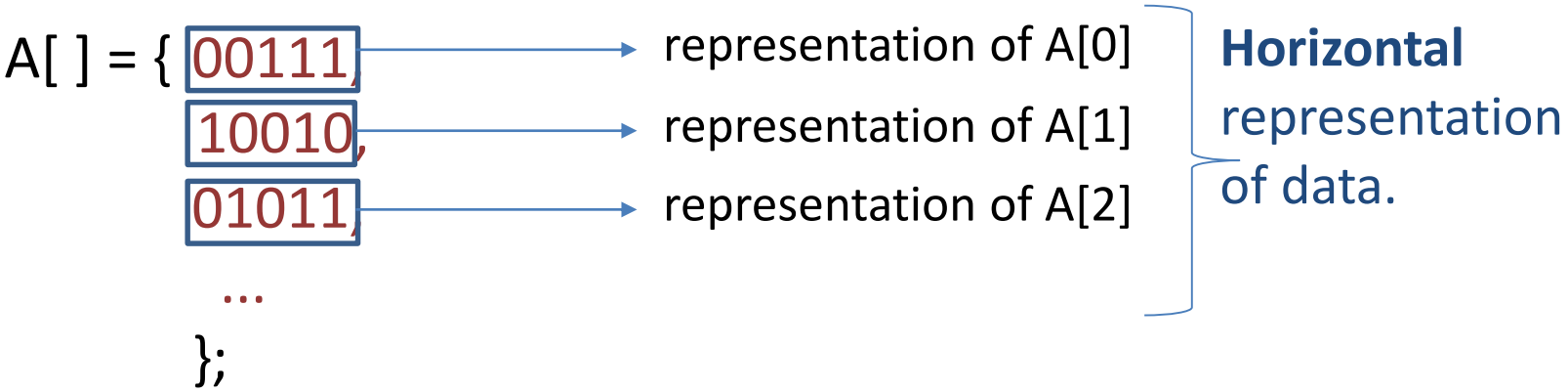

Only the least bit of unsigned int is used to keep the bit.

- Total cost ≈ n **int** operations
- Optimal cost ≈ n **bit** operations
- Computational inefficiency = Total cost/Optimal cost
  = (n **int** operations)/(n **bit** operations)
  = sizeof(**int**)

# Bitslicing: A technique for data orthogonalization to leverage from processor word length

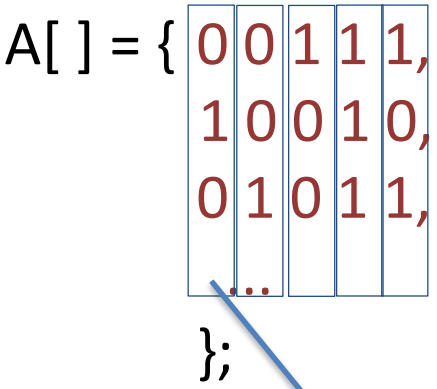
# Data is initially ‘horizontal’

Say, we have is A[ ] = { 7, 18, 11, … }
In binary this is



# How data will look like in ‘orthogonal’ representation

Say, we have is A[ ] = { 7, 18, 11, … }
In binary this is




Bits are packed vertically into integers.

Example: 32 bits from this column are packed into one uint32_t.

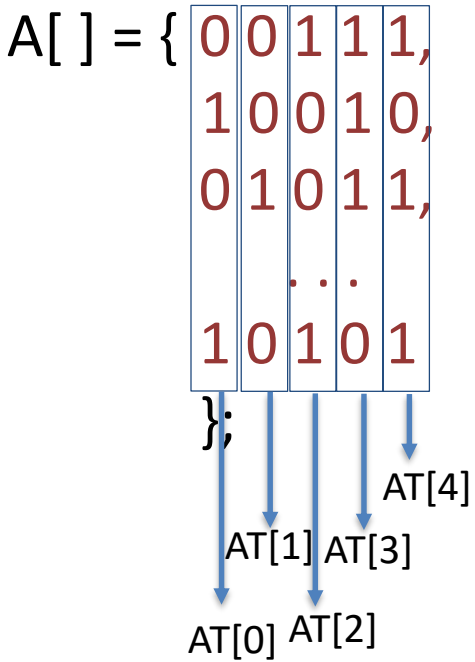
# Horizontal to Orthogonal transformation

```
unsigned int AT[5]={0,0,0,0,0};
for(i=0; i<32; i++){
  AT[0]=AT[0] + bit(A[i],0)<<i;
  AT[1]=AT[1] + bit(A[i],1)<<i;
  AT[2]=AT[2] + bit(A[i],2)<<i;
  AT[3]=AT[3] + bit(A[i],3)<<i;
  AT[4]=AT[4] + bit(A[i],4)<<i;
}
```

After orthogonalization

AT[0] = 010……1
AT[1] = 001……0
AT[2] = 100……1
AT[3] = 111……0
AT[4] = 101……1

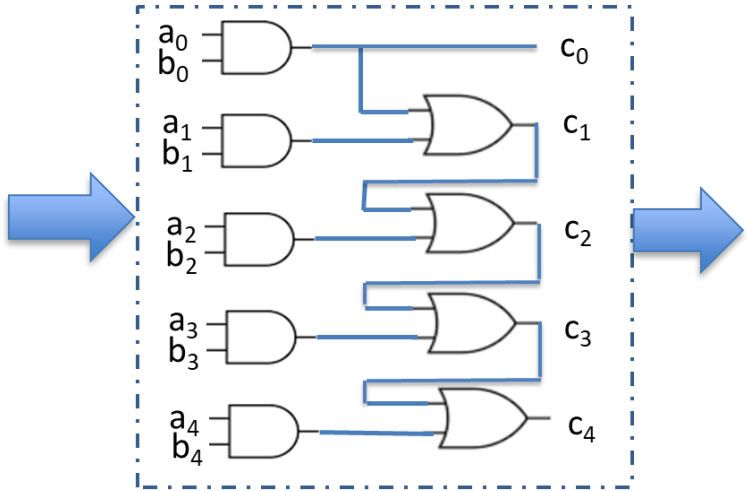
Each having 32 bits from the respective columns.



# Evaluating on Orthogonal data

AT[0], AT[1], …, AT[5]

BT[0], BT[1], …, BT[5]




CT[0], CT[1], …, CT[5]

```
// Bitsliced evaluation of the circuit
CT[0] = AT[0] & BT[0];
CT[1] = (AT[1] & BT[1]) | CT[0];
CT[2] = (AT[2] & BT[2]) | CT[1];
CT[3] = (AT[3] & BT[3]) | CT[2];
CT[4] = (AT[4] & BT[4]) | CT[3];
```

**32 bits are computed in parallel!**

# Improved computational efficiency

```
// Evaluate the circuit
c0 = a0&b0;
c1 = (a1&b1) | c0;
c2 = (a2&b2) | c1;
c3 = (a3&b3) | c2;
c4 = (a4&b4) | c3;
```

**Previous approach**

Computational inefficiency
= Total cost/Optimal cost
= (n **int** op.)/(n **bit** op.)
= sizeof(**int**)

```
// Bitsliced evaluation of the circuit
CT[0] = AT[0] & BT[0];
CT[1] = (AT[1] & BT[1]) | CT[0];
CT[2] = (AT[2] & BT[2]) | CT[1];
CT[3] = (AT[3] & BT[3]) | CT[2];
CT[4] = (AT[4] & BT[4]) | CT[3];
```

Now **data orthogonalization**

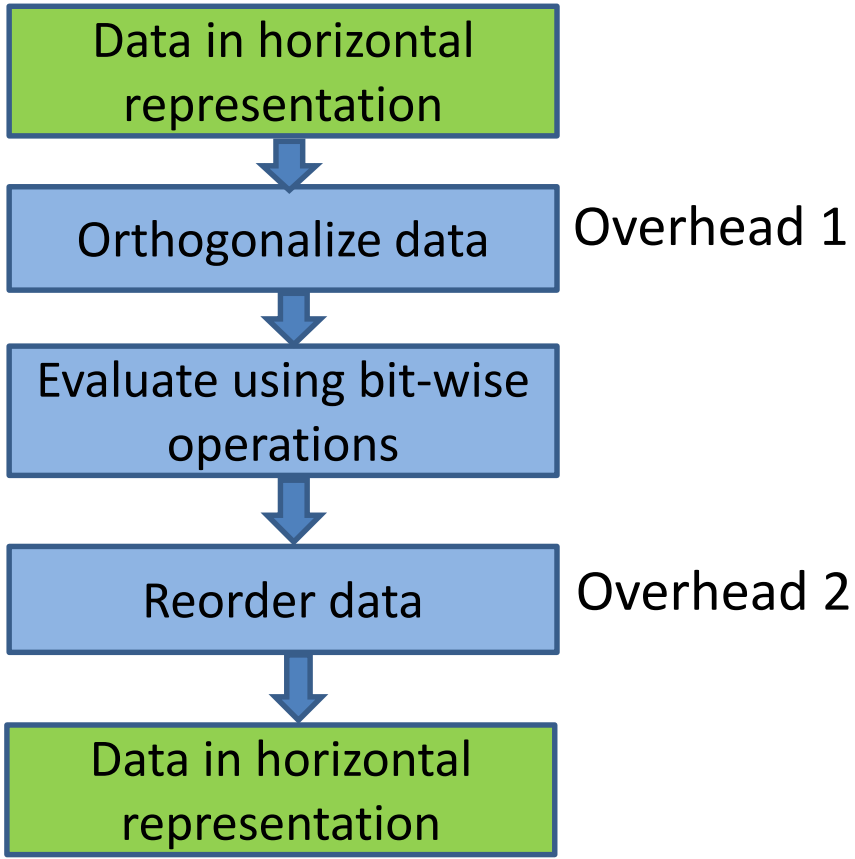
Computational inefficiency
= Total cost/Optimal cost
= (n*32 **bit** op.)/(n*32 **bit** op.)
= 1

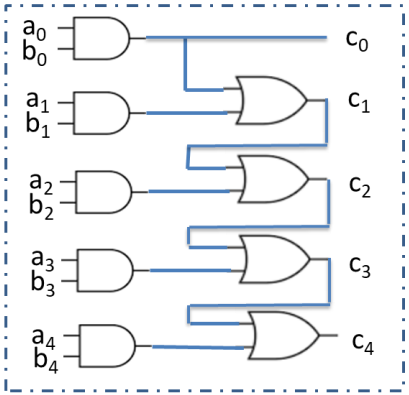
**32x speedup using 32-bit word length**
**64x speedup using 64-bit word length**

# Actual cost of bitsliced implementation

- Previous analysis didn't take data ordering into account.

- Bitslicing can be faster when
  1. Overhead 1 and 2 are small
  2. Operations are performed on a bulk of data
  3. Depending on application, theory vs practice comes in the picture

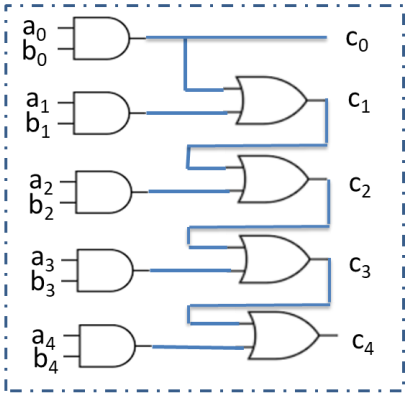





We applied bitslicing to this gate-level circuit evaluation and saw improvements.

## Can we apply bitslicing to a general algorithm?

```
for(i=0;i<L;i++)
{
   foo(…);
   …
}
```




We applied bitslicing to this gate-level circuit evaluation and saw improvements.

# Can we apply bitslicing to a general algorithm?

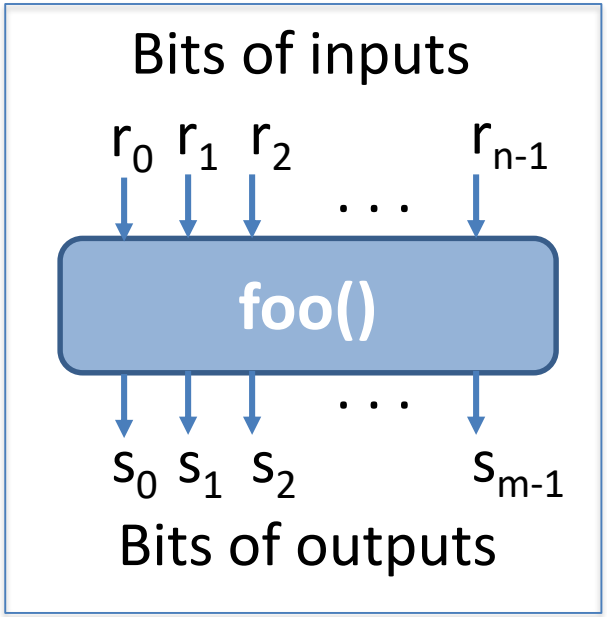
```
for(i=0;i<L;i++)
{
   foo(…);
   …
}
```

First step:
Find a ‘gate-level’ description of foo().

# Bitslicing a small function

Let us consider a generic ‘small function’
→ Small number of inputs

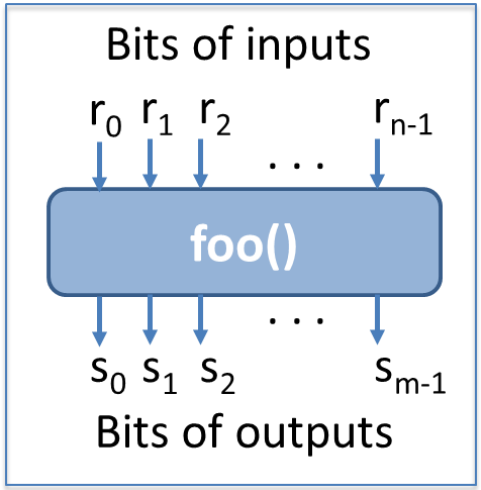
Any function can be viewed as a map: taking inputs and producing outputs




The function maps n-bit binary string $\{r_0,r_1,\ldots,r_{n-1}\}$ to m-bit binary string $\{s_0,s_1,\ldots,s_{m-1}\}$

# Bitslicing a small function

- Construct a Truth Table for foo()




| $r_0, r_1, \ldots r_{n-1}$ | $s_0, s_1, \ldots s_{m-1}$ |
|---|---|
| 0 0 … 0 | 0 1 … 1 |
| 0 0 … 1 | 1 1 … 0 |
| … | … |

- Since the number of input n is small, we can list all $2^n$ possible input cases and the corresponding outputs in a Truth table

# Bitslicing a small function

- Use logic minimization (e.g. Karnaugh Map) to minimize table

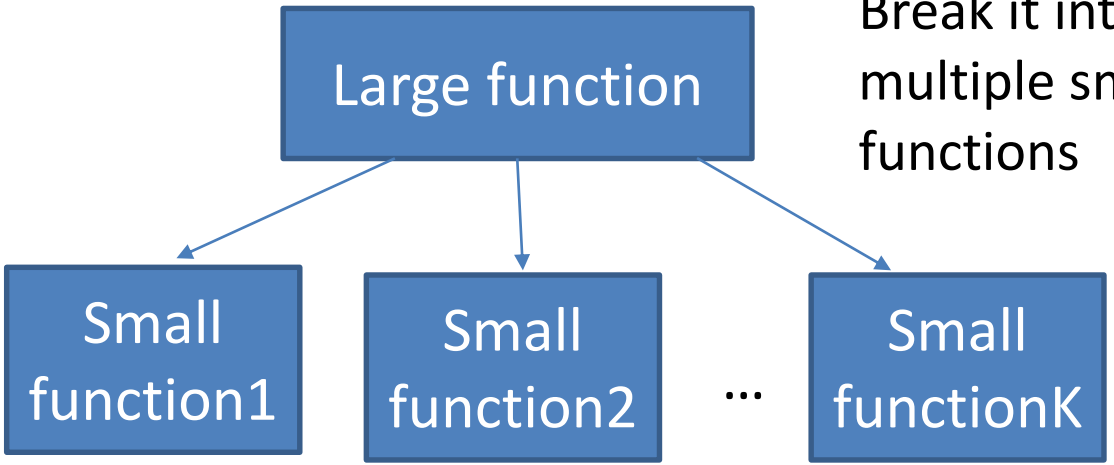
| $r_0, r_1, \ldots r_{n-1}$ | $s_0, s_1, \ldots s_{m-1}$ |
|---|---|
| 0 0 … 0 | 0 1 … 1 |
| 0 0 … 1 | 1 1 … 0 |
| … | … |

- This will generate “Sum of Products (SOP)” Boolean expressions

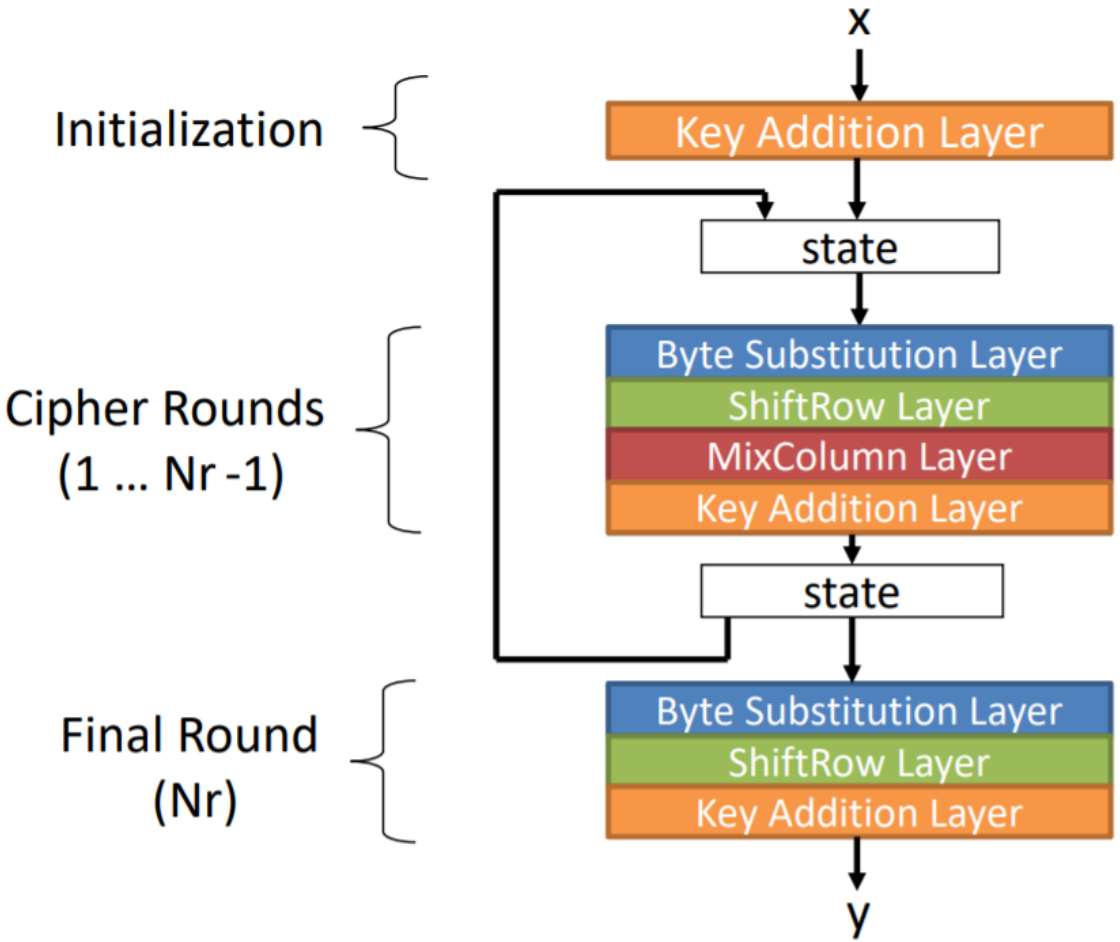
**This gives us a ‘gate-level’ description of *foo*() in AND and OR gates**

# Bitslicing a relatively large function

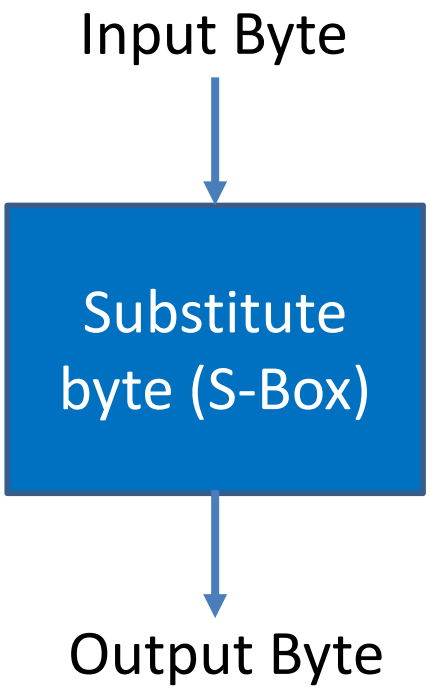
- Large → large number of input bits
- We can bitslice a large function if we can decompose it into multiple ‘small functions’ which can be bitsliced.
- This can be done in a hierarchy



# Bitslicing AES



# Bitslicing S-Box




| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| 10 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | c0 |
| 20 | b7 | fd | 93 | 26 | 36 | 3f | f7 | cc | 34 | a5 | e5 | f1 | 71 | d8 | 31 | 15 |
| 30 | 04 | c7 | 23 | c3 | 18 | 96 | 05 | 9a | 07 | 12 | 80 | e2 | eb | 27 | b2 | 75 |
| 40 | 09 | 83 | 2c | 1a | 1b | 6e | 5a | a0 | 52 | 3b | d6 | b3 | 29 | e3 | 2f | 84 |
| 50 | 53 | d1 | 00 | ed | 20 | fc | b1 | 5b | 6a | cb | be | 39 | 4a | 4c | 58 | cf |
| 60 | d0 | ef | aa | fb | 43 | 4d | 33 | 85 | 45 | f9 | 02 | 7f | 50 | 3c | 9f | a8 |
| 70 | 51 | a3 | 40 | 8f | 92 | 9d | 38 | f5 | bc | b6 | da | 21 | 10 | ff | f3 | d2 |
| 80 | cd | 0c | 13 | ec | 5f | 97 | 44 | 17 | c4 | a7 | 7e | 3d | 64 | 5d | 19 | 73 |
| 90 | 60 | 81 | 4f | dc | 22 | 2a | 90 | 88 | 46 | ee | b8 | 14 | de | 5e | 0b | db |
| a0 | e0 | 32 | 3a | 0a | 49 | 06 | 24 | 5c | c2 | d3 | ac | 62 | 91 | 95 | e4 | 79 |
| b0 | e7 | c8 | 37 | 6d | 8d | d5 | 4e | a9 | 6c | 56 | f4 | ea | 65 | 7a | ae | 08 |
| c0 | ba | 78 | 25 | 2e | 1c | a6 | b4 | c6 | e8 | dd | 74 | 1f | 4b | bd | 8b | 8a |
| d0 | 70 | 3e | b5 | 66 | 48 | 03 | f6 | 0e | 61 | 35 | 57 | b9 | 86 | c1 | 1d | 9e |
| e0 | e1 | f8 | 98 | 11 | 69 | d9 | 8e | 94 | 9b | 1e | 87 | e9 | ce | 55 | 28 | df |
| f0 | 8c | a1 | 89 | 0d | bf | e6 | 42 | 68 | 41 | 99 | 2d | 0f | b0 | 54 | bb | 16 |

# Bitslicing S-Box

Follow the steps of bitslicing a small function

1. Represent S-Box as a Truth table

| $r_0, r_1, \ldots r_7$ | $s_0, s_1, \ldots s_7$ |
|---|---|
| 0 0 … 0 | 0 1 … 1 |
| 0 0 … 1 | 1 1 … 0 |
| … | … |

8 bit to 8 bit Truth table.
There are 256 entries.

2. Now generate “Sum of Products (SOP)” Boolean expressions.

# An efficient bit-sliced AES implementation

Faster and Timing-Attack Resistant AES-GCM

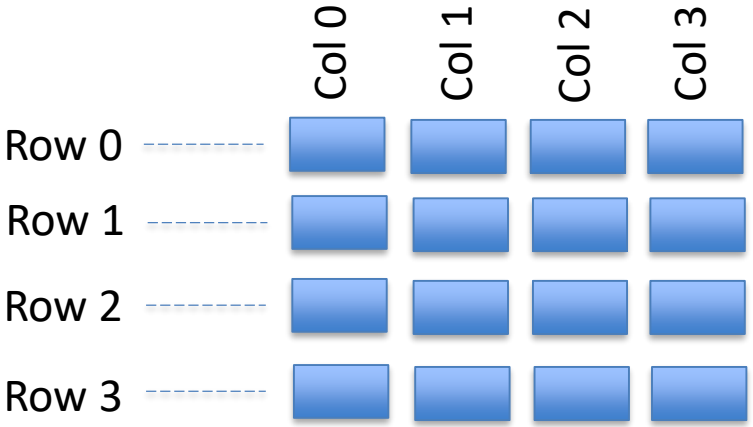
Emilia Käsper[1] and Peter Schwabe[2] ⋆

… with some simplifications for the purpose of course-lecture.

“Faster and Timing-Attack Resistant AES-GCM” by Emilia Käsper and Peter Schwabe. CHES 2009

# AES state

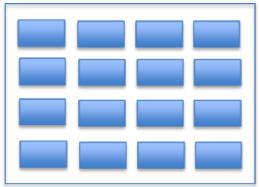
AES state consists of 16 bytes: 4 rows and 4 columns



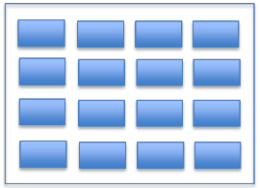
# AES state

AES state consists of 16 bytes: 4 rows and 4 columns

State of 1st block

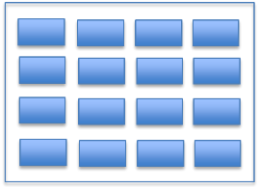


State of 2nd block



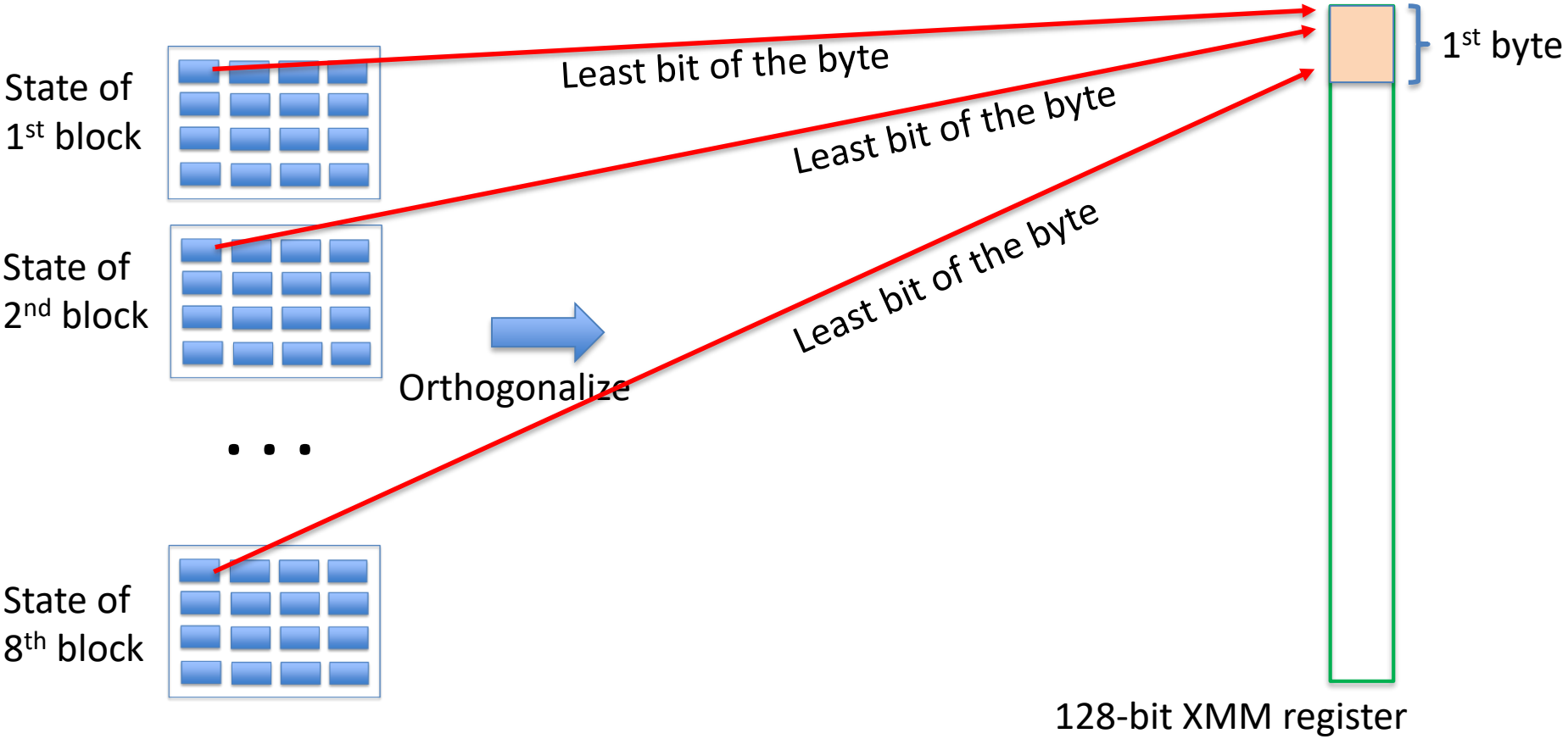
. . .

State of 8th block



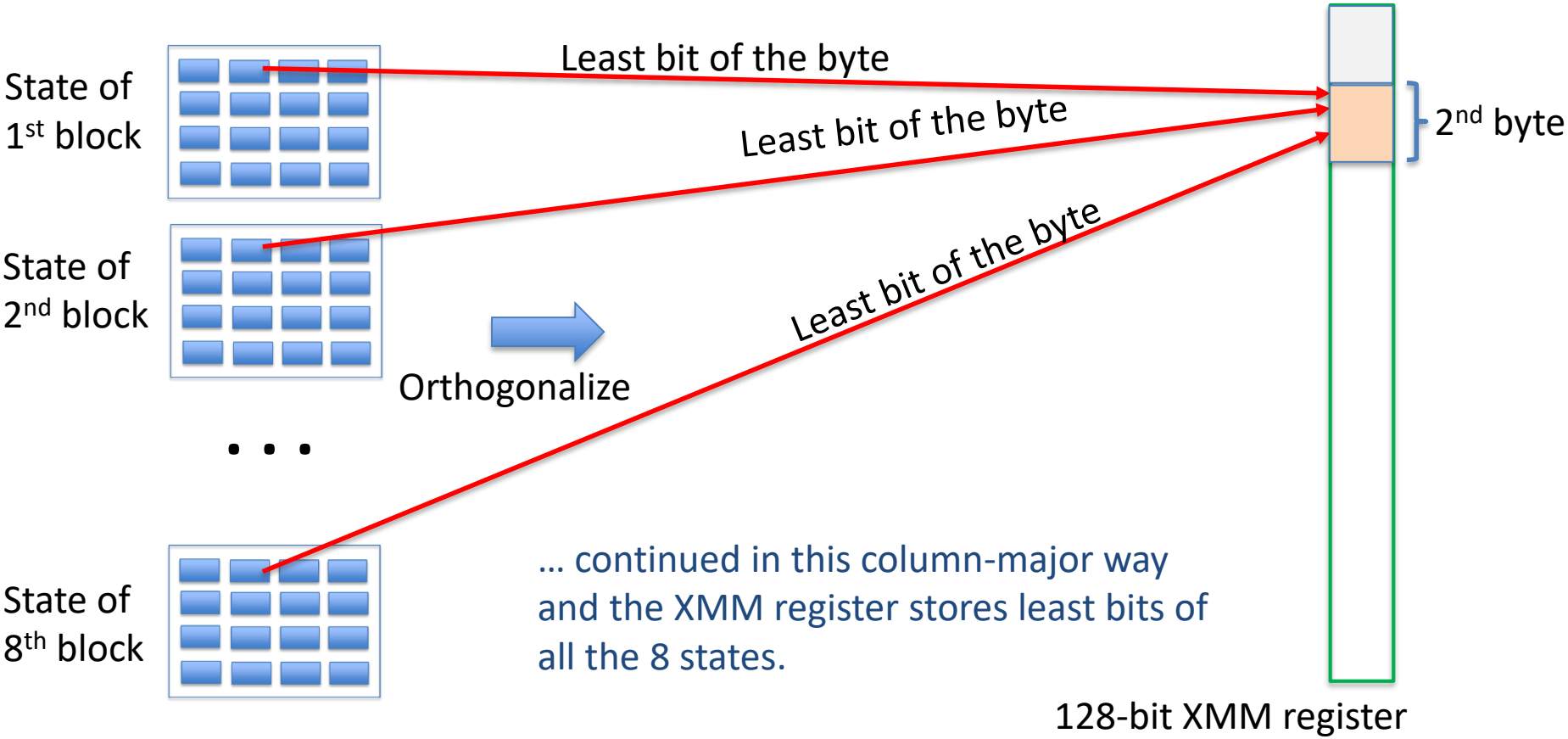
# AES state orthogonalization

AES state consists of 16 bytes: 4 rows and 4 columns



# AES state orthogonalization

AES state consists of 16 bytes: 4 rows and 4 columns

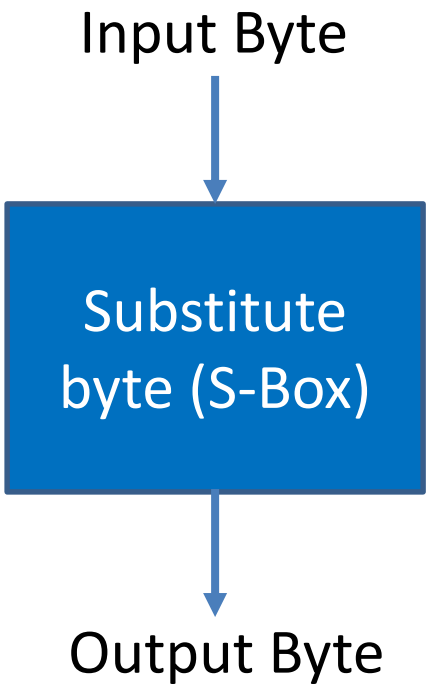


… similarly, 2nd least bit of each byte are packed in the 2nd XMM register

… similarly, 3rd least bit of each byte are packed in the 3rd XMM register

… finally, 8th bit (i.e., MSb) of each byte are packed in the 8th XMM register

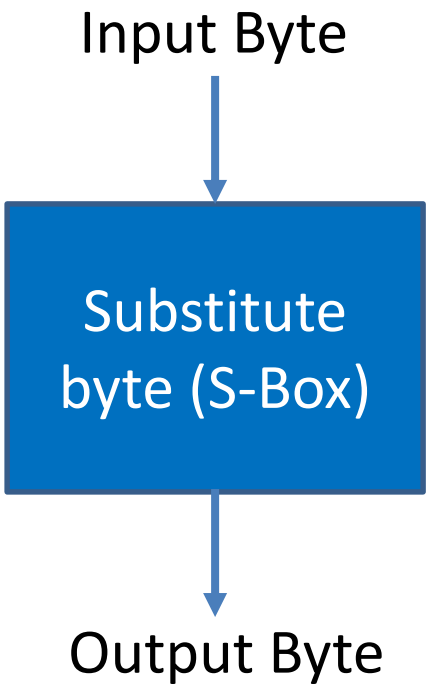
# Bit-sliced Sub-Byte




For a bit-sliced implementation we need Boolean expressions (i.e., gate-level description) for output bits.

Optimization: minimize the number of gates in the circuit description of S-Box.

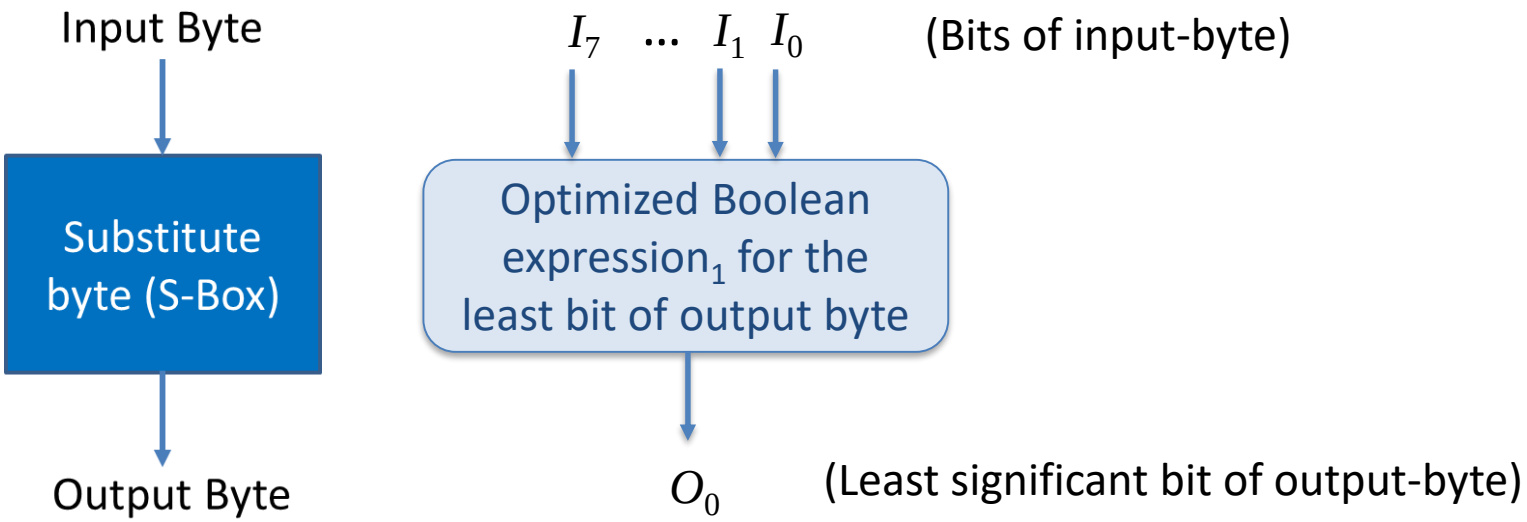
# Bit-sliced Sub-Byte

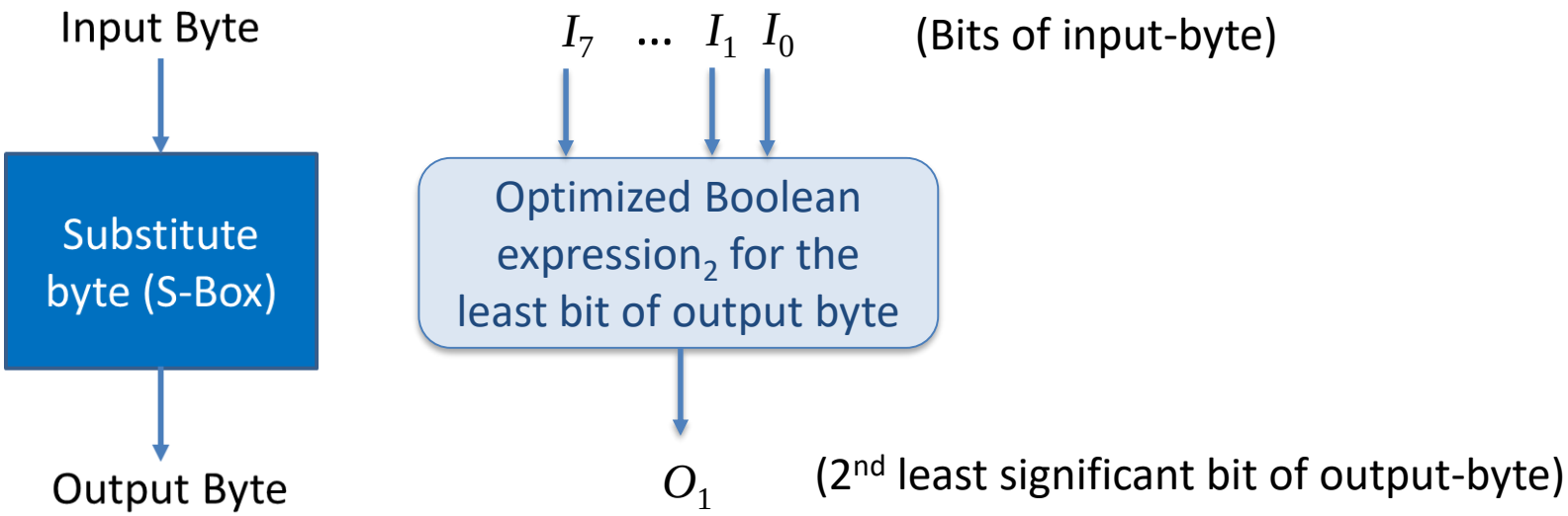



For a bit-sliced implementation we need Boolean expressions (i.e., gate-level description) for output bits.

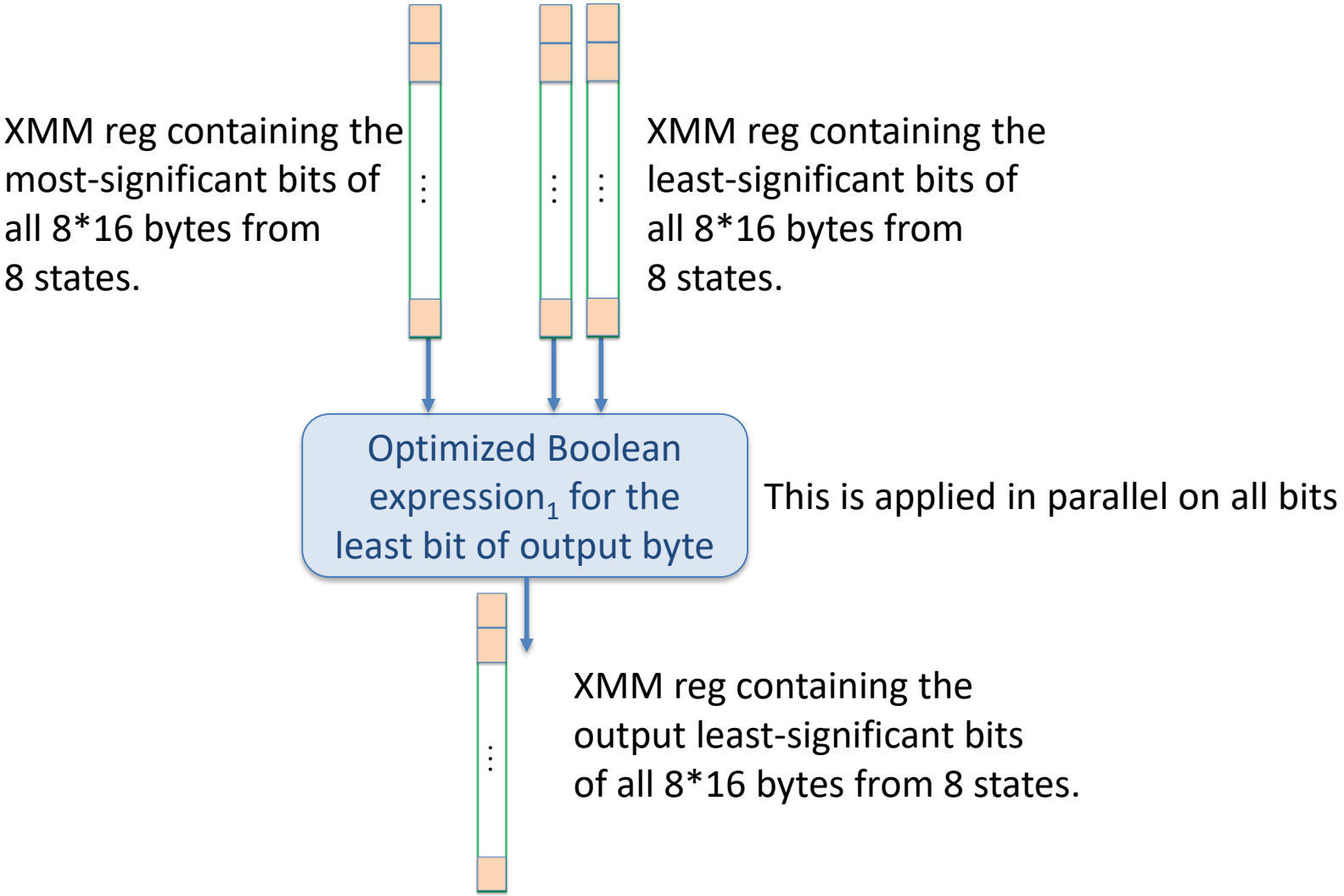
Optimization: minimize the number of gates in the circuit description of S-Box.
→ [BP2009] obtained Boolean expressions for S-Box with only 117 gates.

[BP2009] “New logic minimization techniques with applications to cryptology”. Joan Boyar and Rene Peralta. https://eprint.iacr.org/2009/191


Input Byte
Substitute byte (S-Box)
Output Byte
$I_7$ ... $I_1$ $I_0$
(Bits of input-byte)
Optimized Boolean expression$_1$ for the least bit of output byte
$O_0$
(Least significant bit of output-byte)

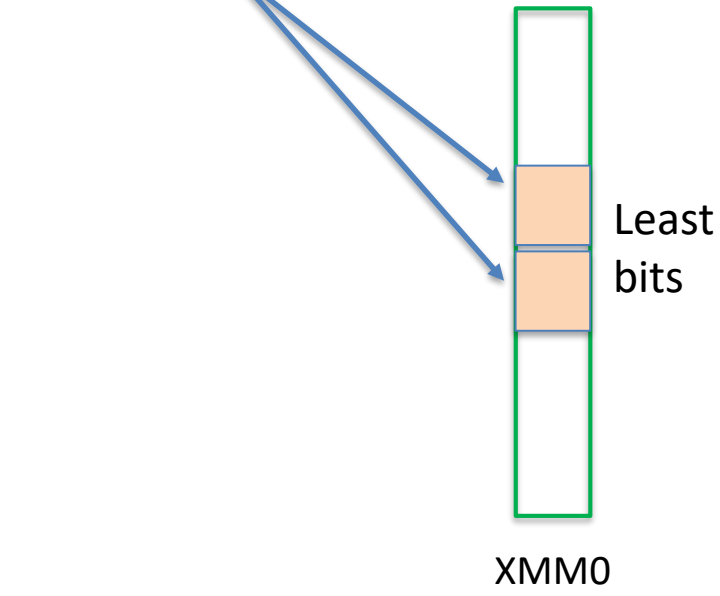



… and so on. Every output bit is written as an expression of input bits.


XMM reg containing the most-significant bits of all 8*16 bytes from 8 states.
XMM reg containing the least-significant bits of all 8*16 bytes from 8 states.
Optimized Boolean expression$_1$ for the least bit of output byte
This is applied in parallel on all bits
XMM reg containing the output least-significant bits of all 8*16 bytes from 8 states.

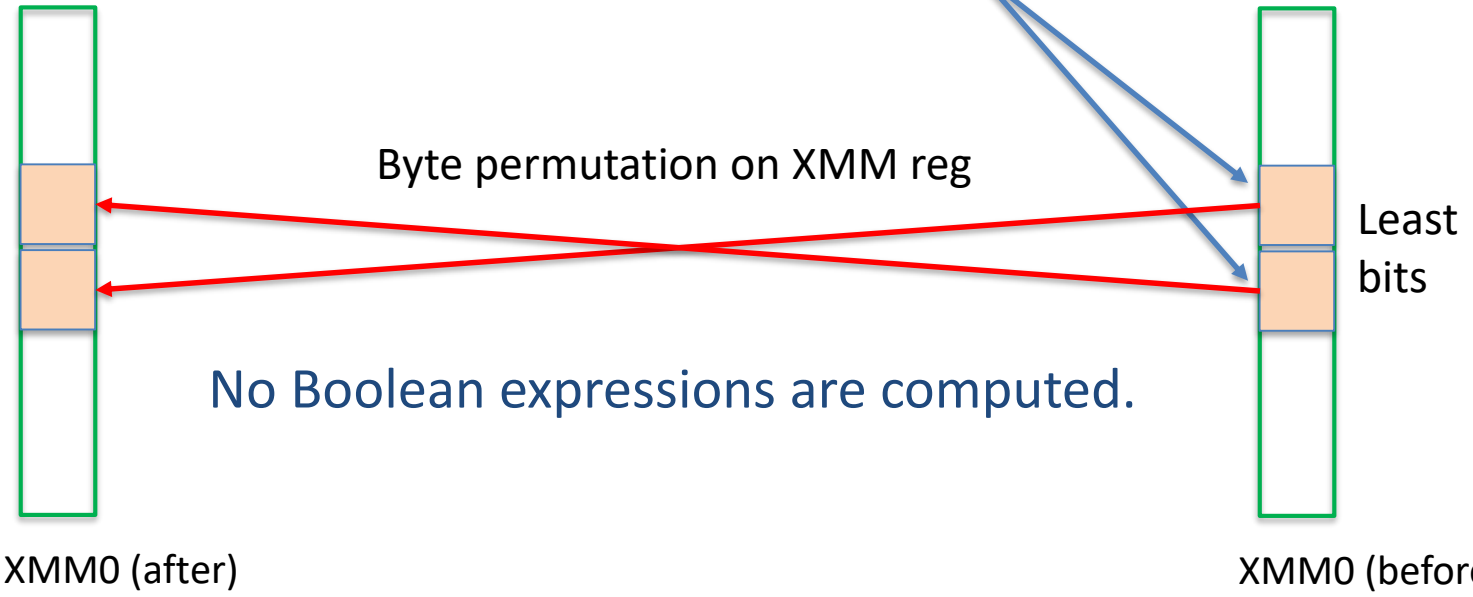
# Bit-sliced Shift-Rows

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$



# Bit-sliced Shift-Rows

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$



# Bit-sliced Mix-Columns

This step can be viewed as a matrix multiplication.

$$\begin{pmatrix} d_0 & d_4 & d_8 & d_{12} \\ d_1 & d_5 & d_9 & d_{13} \\ d_2 & d_6 & d_{10} & d_{14} \\ d_3 & d_7 & d_{11} & d_{15} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{pmatrix}$$

[KS2019] found optimized Boolean expressions.

[KS2019] “Faster and Timing-Attack Resistant AES-GCM” by Emilia Käsper and Peter Schwabe. CHES 2019