Usuba
Optimizing ~~& Trustworthy~~ Bitslicing Compiler

Darius Mercadier Pierre-Évariste Dagand

LIP6 – CNRS – INRIA
Sorbonne Université

February 19, 2019

Vintage cryptography

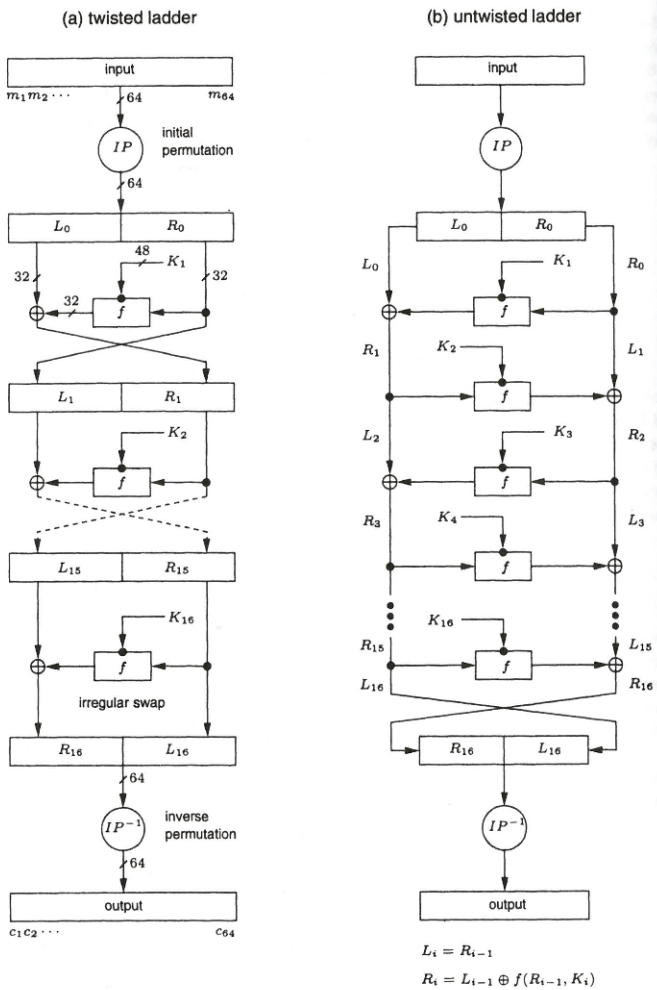


Figure 7.9: *DES computation path.*

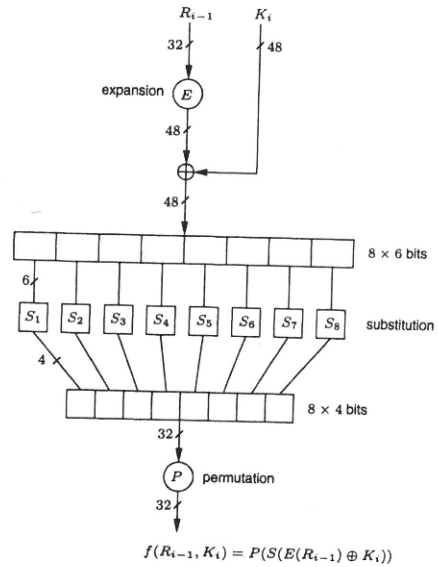


Figure 7.10: *DES inner function f.*

7.83 Algorithm DES key schedule

INPUT: 64-bit key $K = k_1 \dots k_{64}$ (including 8 odd-parity bits).
OUTPUT: sixteen 48-bit keys K_i, $1 \le i \le 16$.

1. Define v_i, $1 \le i \le 16$ as follows: $v_i = 1$ for $i \in \{1, 2, 9, 16\}$; $v_i = 2$ otherwise. (These are left-shift values for 28-bit circular rotations below.)
2. $T \leftarrow \text{PC1}(K)$; represent T as 28-bit halves (C_0, D_0). (Use PC1 in Table 7.4 to select bits from K: $C_0 = k_{57}k_{49}\dots k_{36}$, $D_0 = k_{63}k_{55}\dots k_4$.)
3. For i from 1 to 16, compute K_i as follows: $C_i \leftarrow (C_{i-1} \hookleftarrow v_i)$, $D_i \leftarrow (D_{i-1} \hookleftarrow v_i)$, $K_i \leftarrow \text{PC2}(C_i, D_i)$. (Use PC2 in Table 7.4 to select 48 bits from the concatenation $b_1 b_2 \dots b_{56}$ of C_i and D_i: $K_i = b_{14} b_{17} \dots b_{32}$. '$\hookleftarrow$' denotes left circular shift.)

If decryption is designed as a simple variation of the encryption function, savings result in hardware or software code size. DES achieves this as outlined in Note 7.84.

7.84 Note (*DES decryption*) DES decryption consists of the encryption algorithm with the same key but reversed key schedule, using in order $K_{16}, K_{15}, \dots, K_1$ (see Note 7.85). This works as follows (refer to Figure 7.9). The effect of IP^{-1} is cancelled by IP in decryption, leaving (R_{16}, L_{16}); consider applying round 1 to this input. The operation on the left

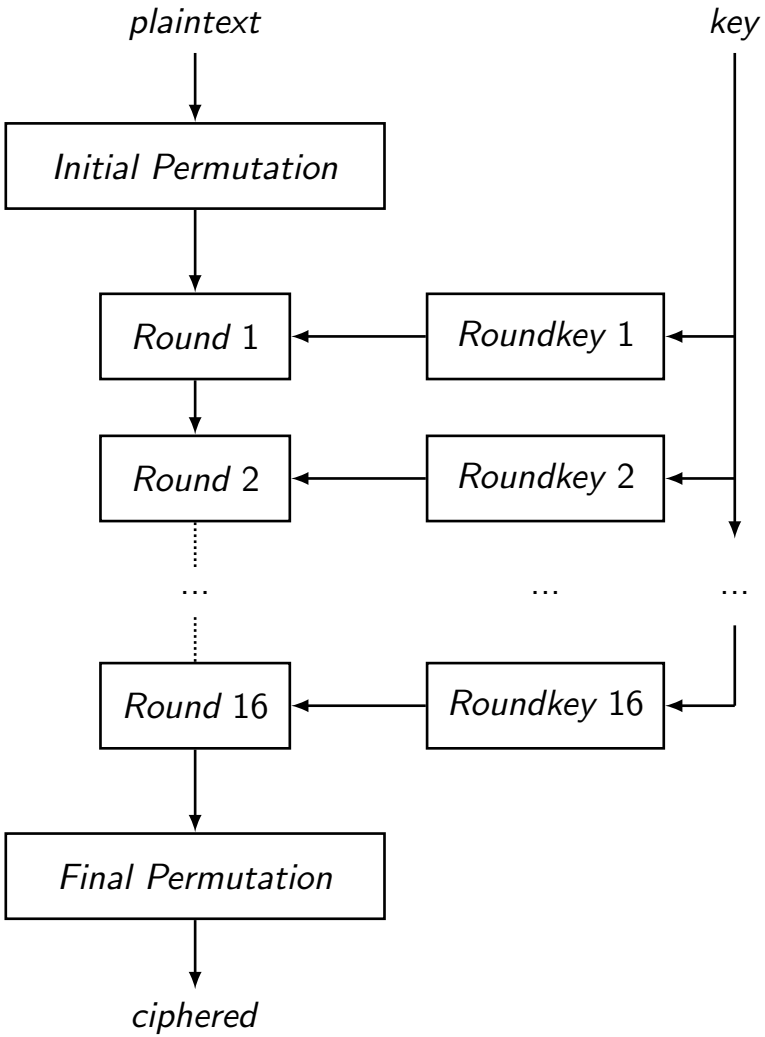
DES in C

```
s1 (r31 ^ k[47], r0 ^ k[11], r1 ^ k[26], r2 ^ k[3], r3 ^ k[13],
    r4 ^ k[41], &l8, &l16, &l22, &l30);
s2 (r3 ^ k[27], r4 ^ k[6], r5 ^ k[54], r6 ^ k[48], r7 ^ k[39],
    r8 ^ k[19], &l12, &l27, &l1, &l17);
s3 (r7 ^ k[53], r8 ^ k[25], r9 ^ k[33], r10 ^ k[34], r11 ^ k[17],
    r12 ^ k[5], &l23, &l15, &l29, &l5);
s4 (r11 ^ k[4], r12 ^ k[55], r13 ^ k[24], r14 ^ k[32], r15 ^ k[40],
    r16 ^ k[20], &l25, &l19, &l9, &l0);
s5 (r15 ^ k[36], r16 ^ k[31], r17 ^ k[21], r18 ^ k[8], r19 ^ k[23],
    r20 ^ k[52], &l7, &l13, &l24, &l2);
s6 (r19 ^ k[14], r20 ^ k[29], r21 ^ k[51], r22 ^ k[9], r23 ^ k[35],
    r24 ^ k[30], &l3, &l28, &l10, &l18);
s7 (r23 ^ k[2], r24 ^ k[37], r25 ^ k[22], r26 ^ k[0], r27 ^ k[42],
    r28 ^ k[38], &l31, &l11, &l21, &l6);
s8 (r27 ^ k[16], r28 ^ k[43], r29 ^ k[44], r30 ^ k[1], r31 ^ k[7],
    r0 ^ k[28], &l4, &l26, &l14, &l20);
...
```

(credits: Matthew Kwan)

Enters Usuba

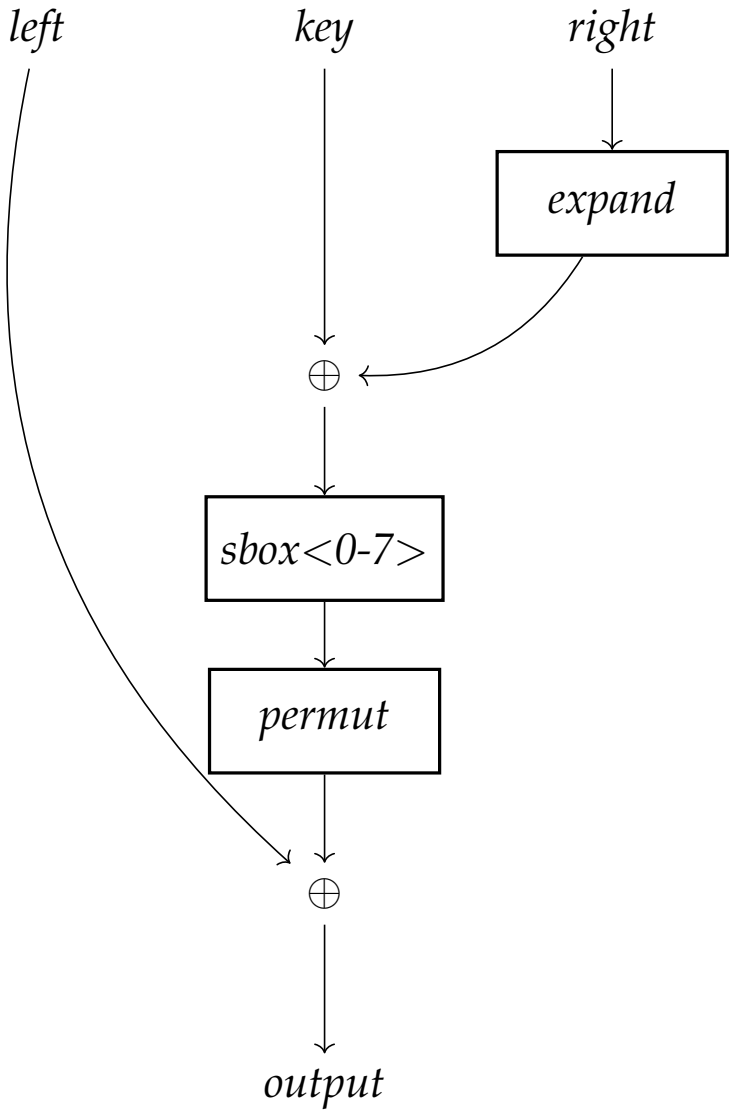
DES




```
node des ( plaintext : u64, key : u64 )
     returns ( ciphered : u64 )
vars
    des :u64[17],
    left:u32, right:u32
let
    // Initial permutation
    des[0] = init_p(plaintext);

    // Computing the 16 rounds
    forall i in [1,16],
      des[i] = round(des[i-1],
                     roundkey<i-1>(key));

    // Final permutation
    (left,right) = des[16];
    ciphered = final_p(right,left)
tel
```

Enters Usuba

Single round

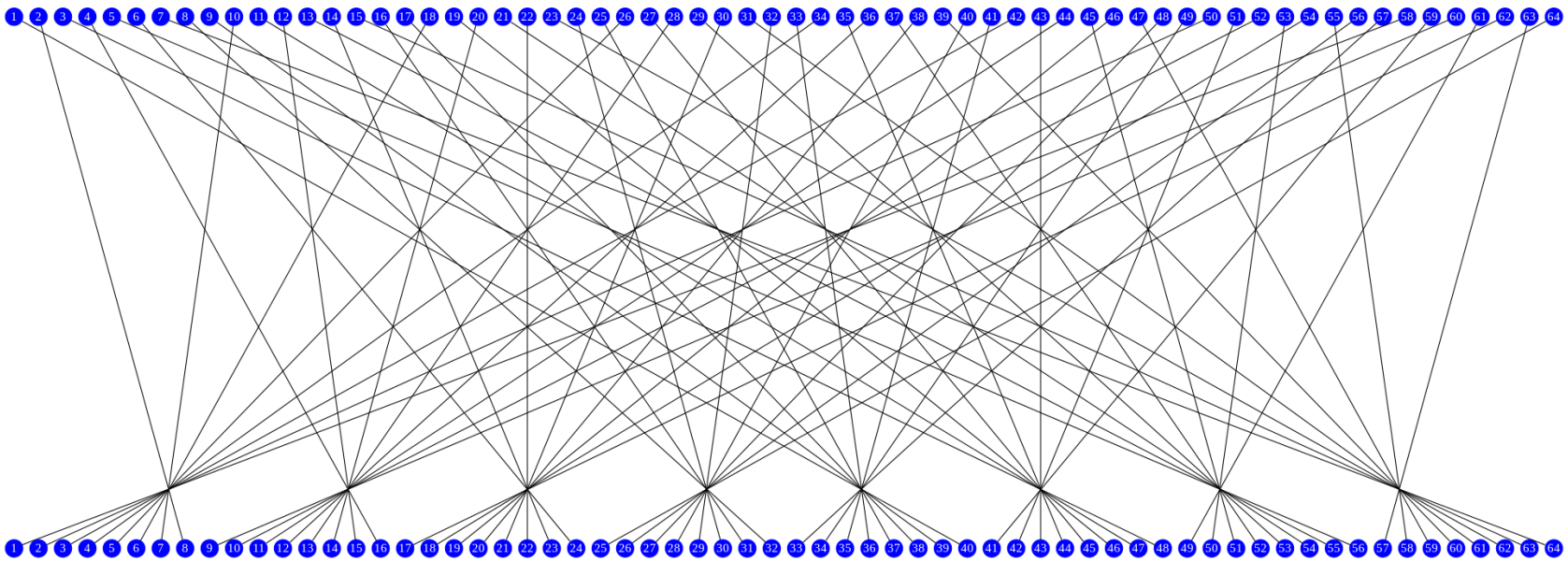



```
node round ( left   : u32,
             right  : u32,
             key    : u48 )
       returns ( output : u64 )
vars
    sbox_in : u6[8], sbox_out : u4[8]
let
    // Expansion and xor with the key
    sbox_in = expand(right) ^ key;

    // Computing the S-boxes
    forall i in [0,7] {
      sbox_out[i] = sbox<i>(sbox_in[i])
    }

    // linear permutation and final xor
    output = (right,
              left ^ permut(sbox_out))
tel
```

Enters Usuba

Permutation

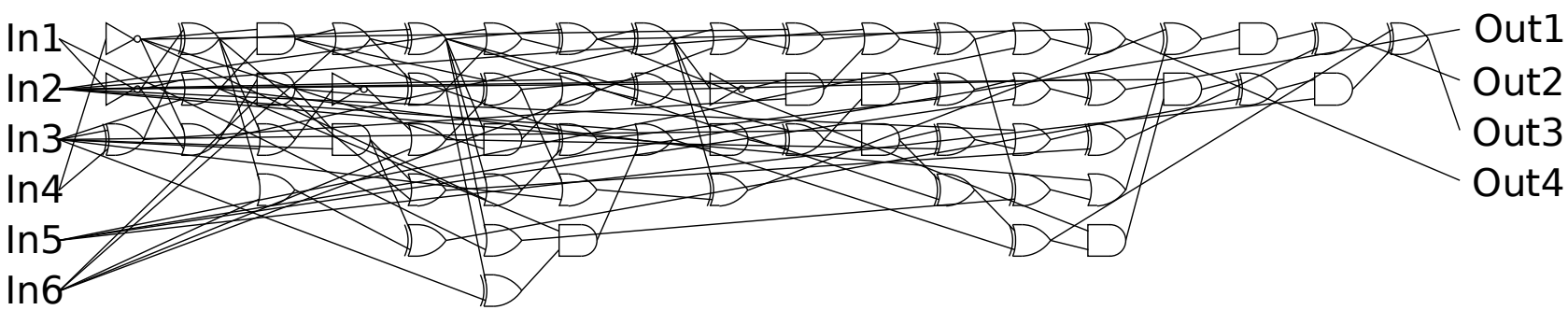
```
perm init_p ( input : u64 ) returns ( out : u64 ) {
    58, 50, 42, 34, 26, 18, 10,  2, 60, 52, 44, 36, 28,
    20, 12,  4, 62, 54, 46, 38, 30, 22, 14,  6, 64, 56,
    48, 40, 32, 24, 16,  8, 57, 49, 41, 33, 25, 17,  9,
    1 , 59, 51, 43, 35, 27, 19, 11,  3, 61, 53, 45, 37,
    29, 21, 13,  5, 63, 55, 47, 39, 31, 23, 15,  7
}
```



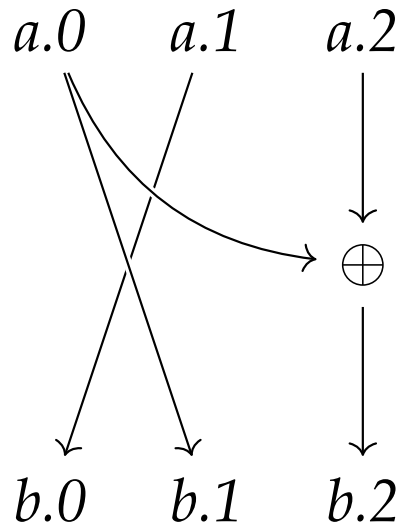
Enters Usuba

S-box / lookup table

```
table sbox_1 ( input : u6 ) returns ( out : u4 ) {
    14,  0,  4, 15, 13,  7,  1,  4,  2, 14, 15,  2, 11,
    13,  8,  1,  3, 10, 10,  6,  6, 12, 12, 11,  5,  9,
     9,  5,  0,  3,  7,  8,  4, 15,  1, 12, 14,  8,  8,
     2, 13,  4,  6,  9,  2,  1, 11,  7, 15,  5, 12, 11,
     9,  3,  7, 14,  3, 10, 10,  0,  5,  6,  0, 13
}
```



Combinational circuits in software



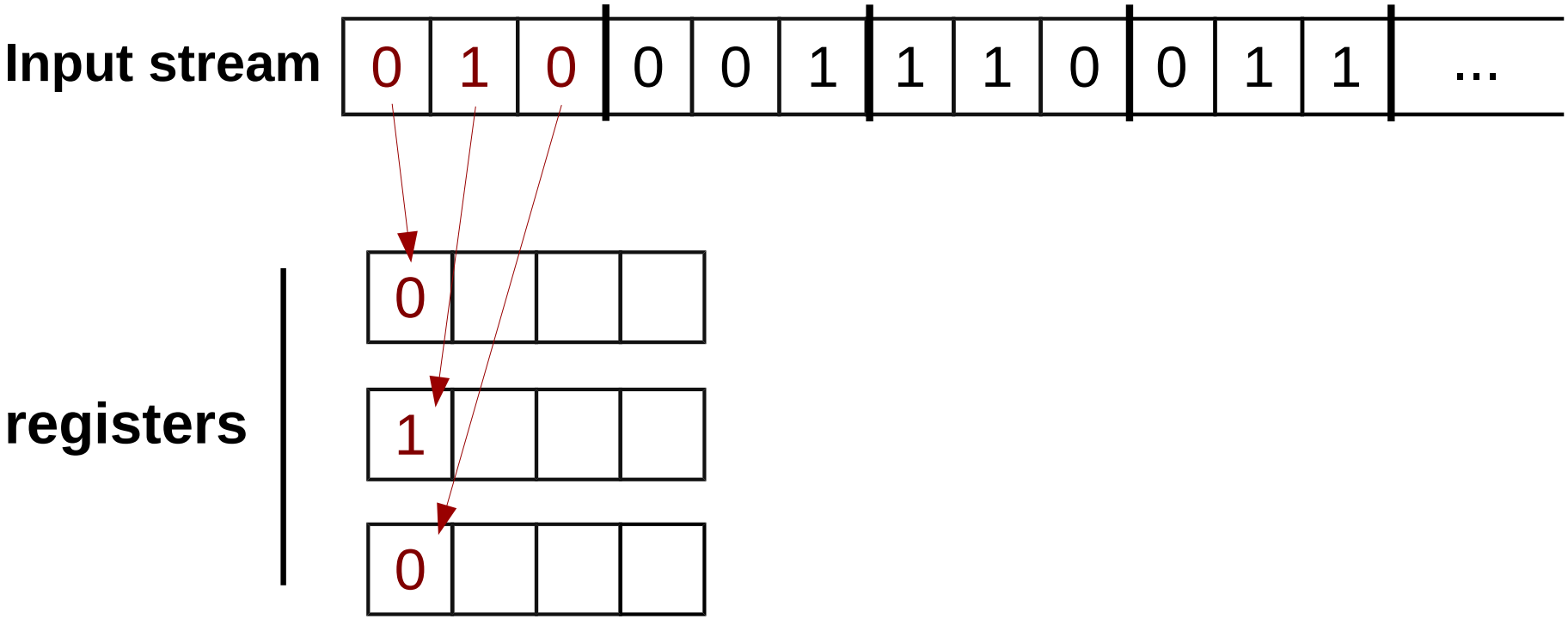
Usuba

```
node example ( a : u3 )
     returns ( b : u3 )
let
  b.0 = a.1;
  b.1 = a.0;
  b.2 = a.0 ^ a.2;
tel
```

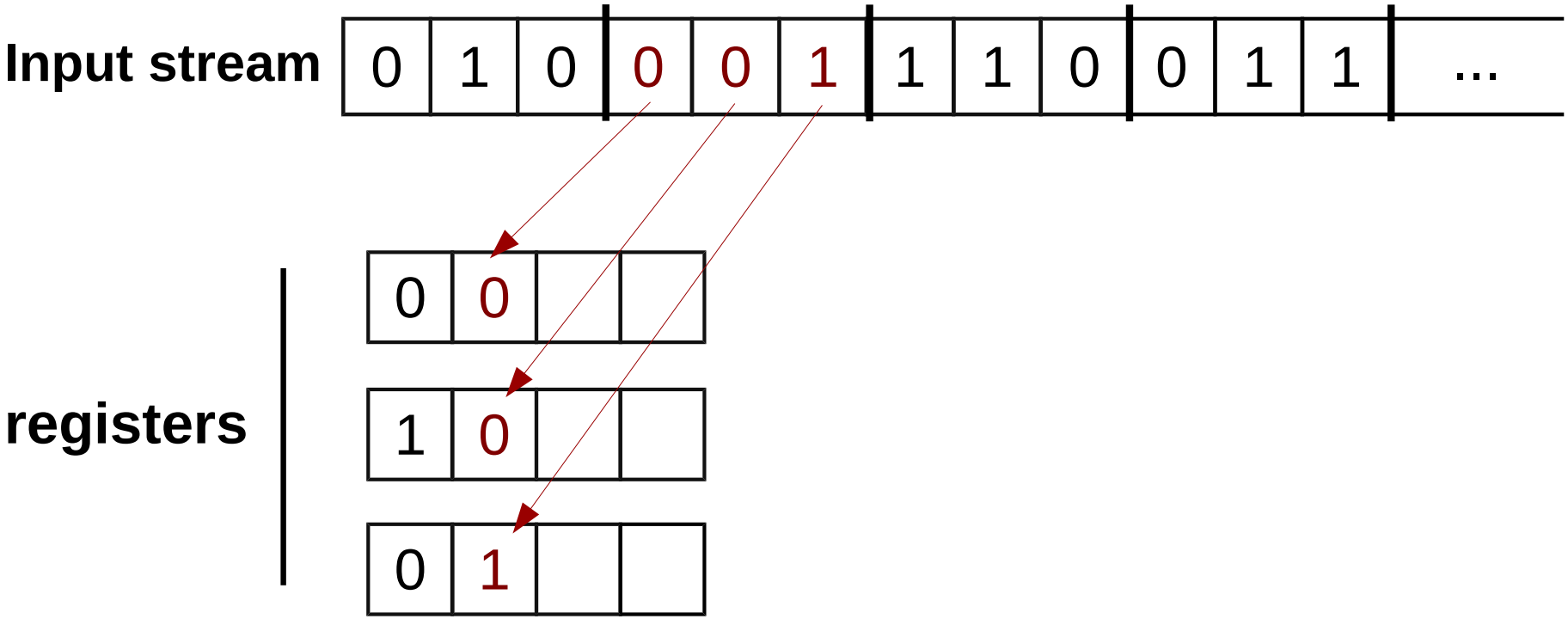
C

```
unsigned char example
        (unsigned char a) {
  return
      (a >> 1 & 1) |
      (a << 1 & 2) |
      ((a << 2) ^ a & 4);
}
```

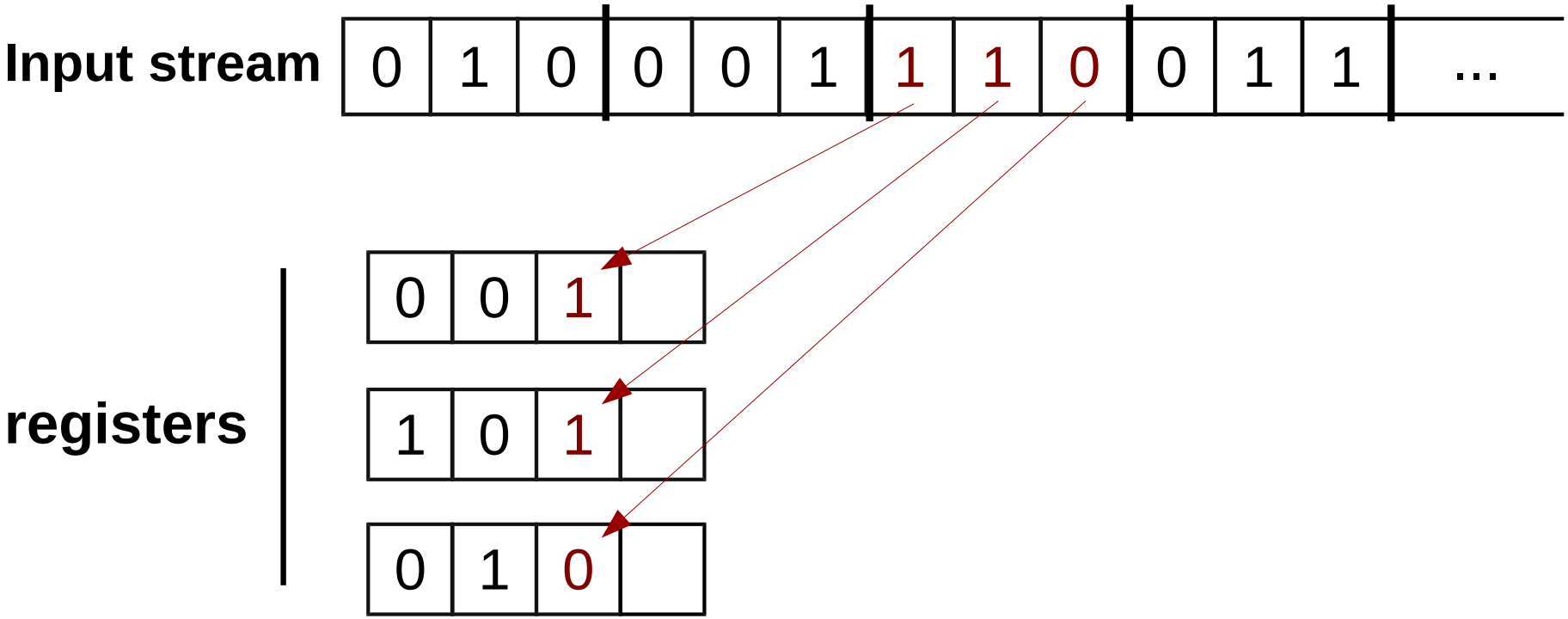
Bit-level parallelism



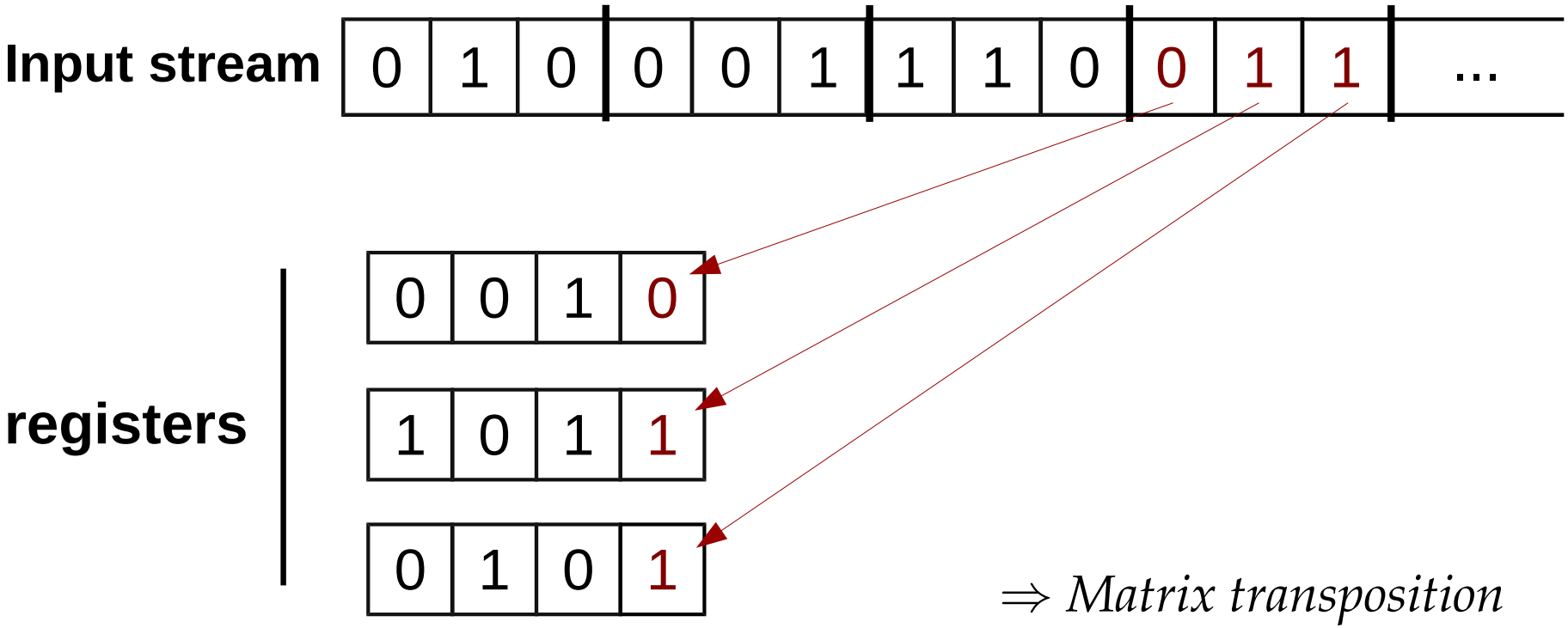
Bit-level parallelism



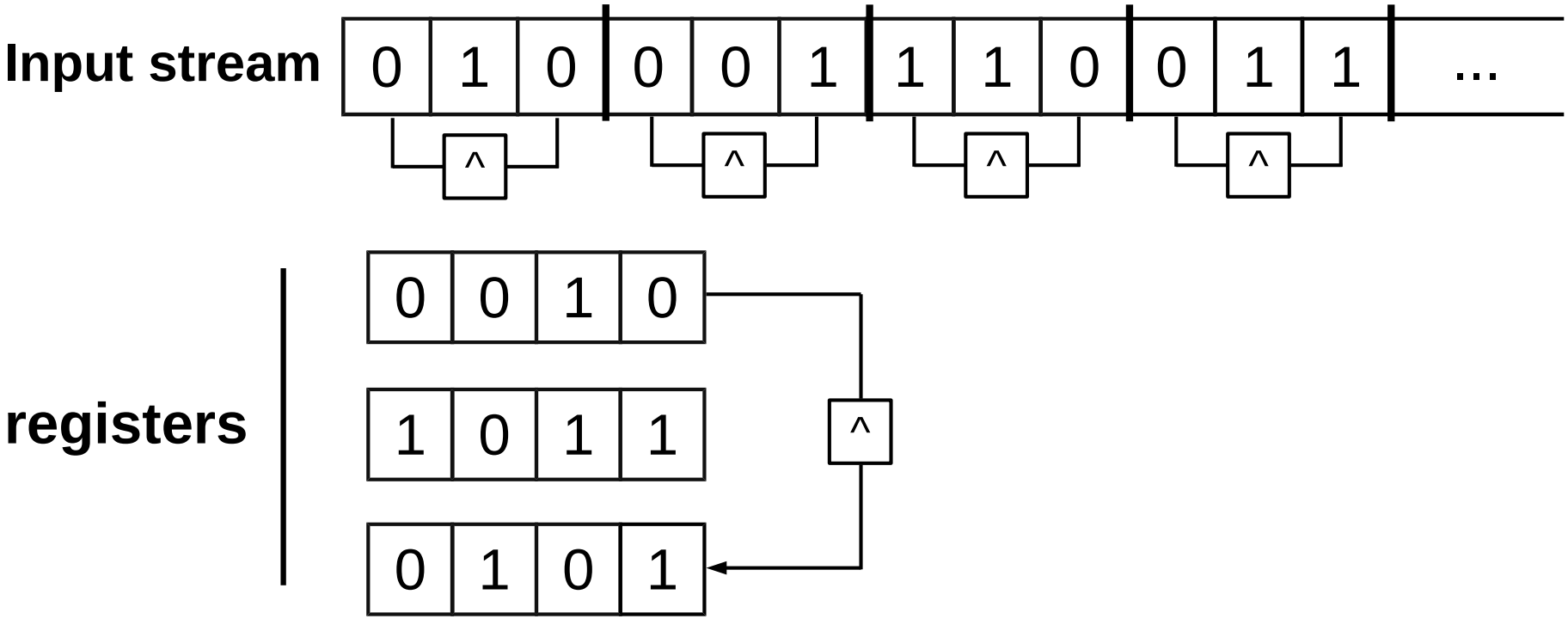
Bit-level parallelism



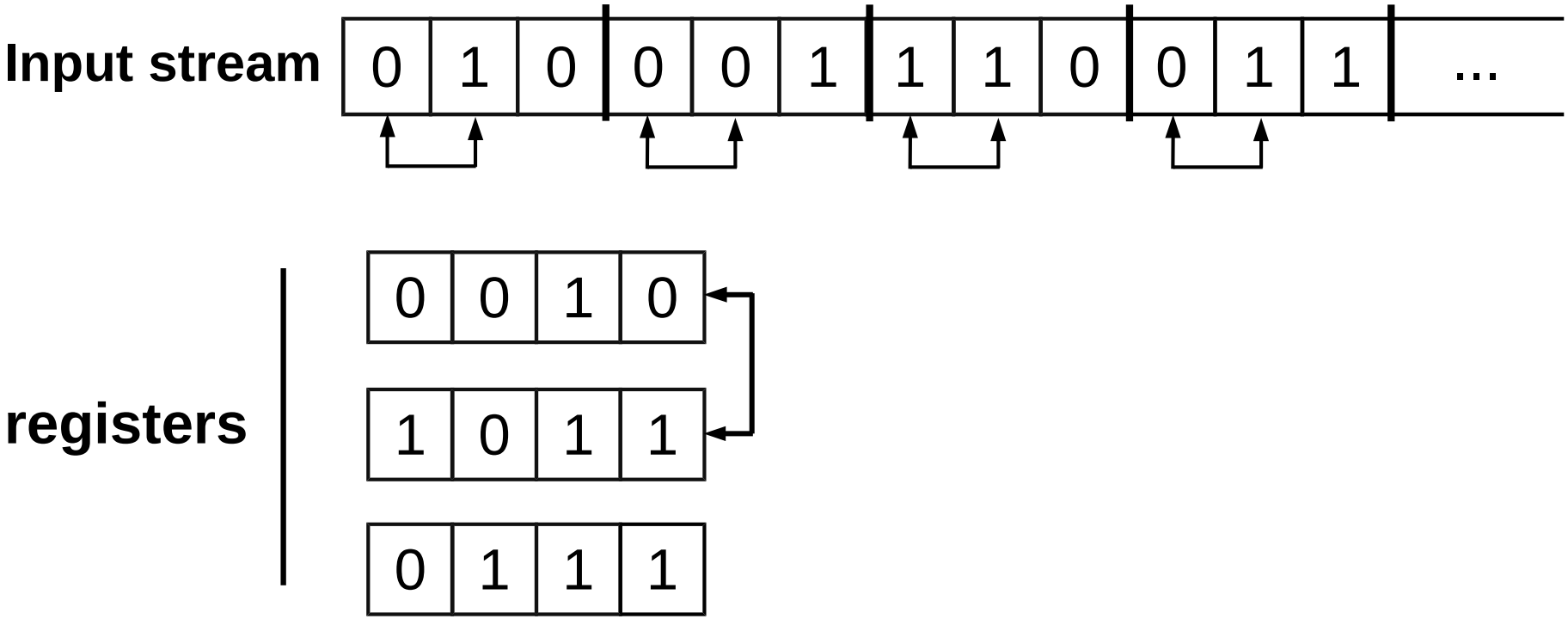
Bit-level parallelism



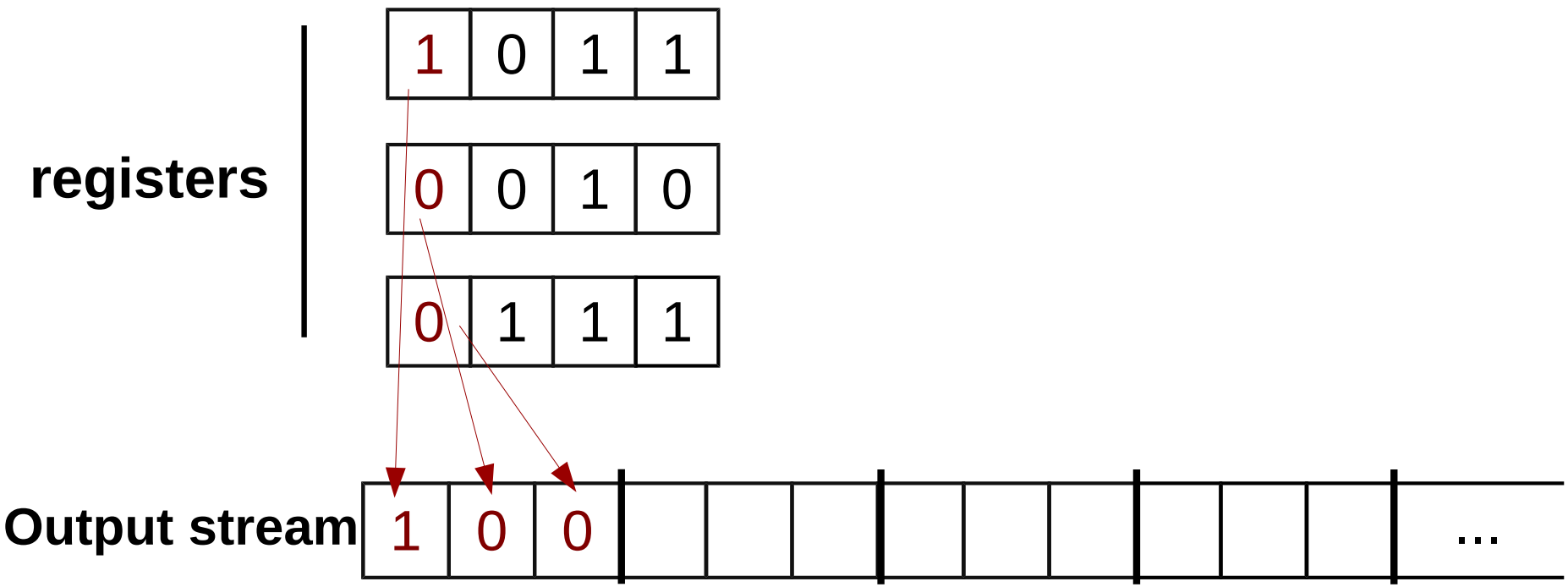
Bit-level parallelism



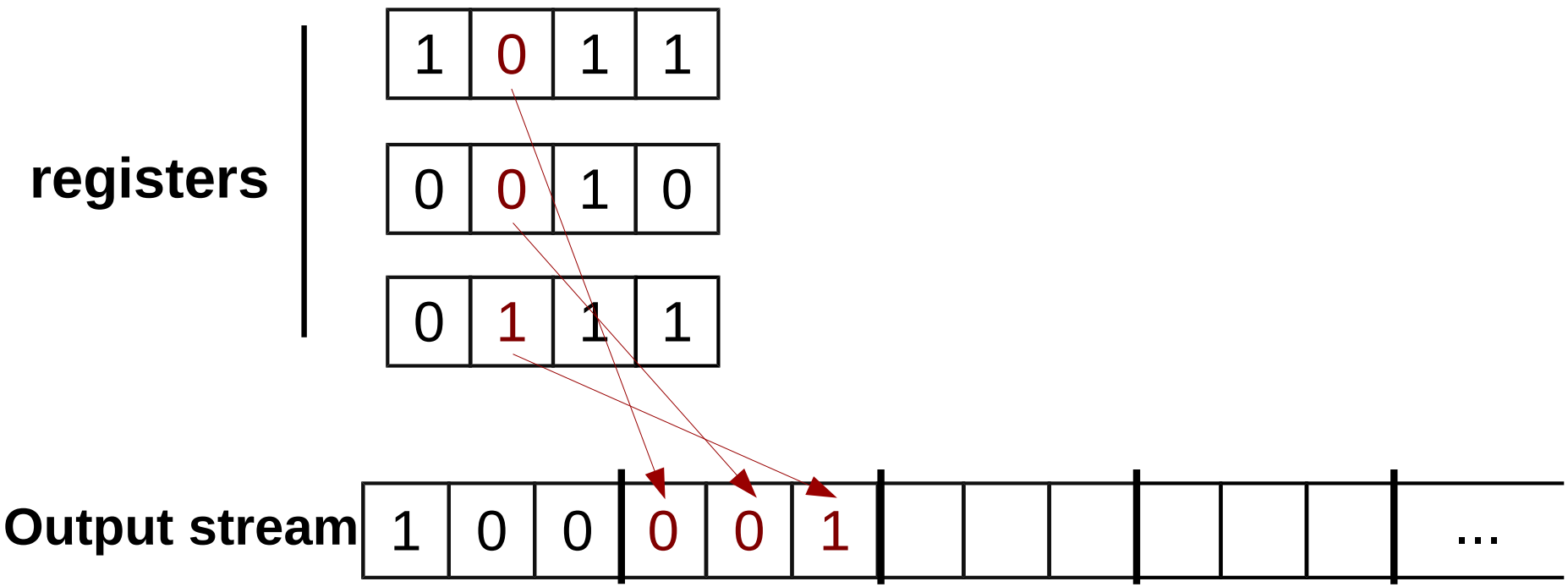
Bit-level parallelism



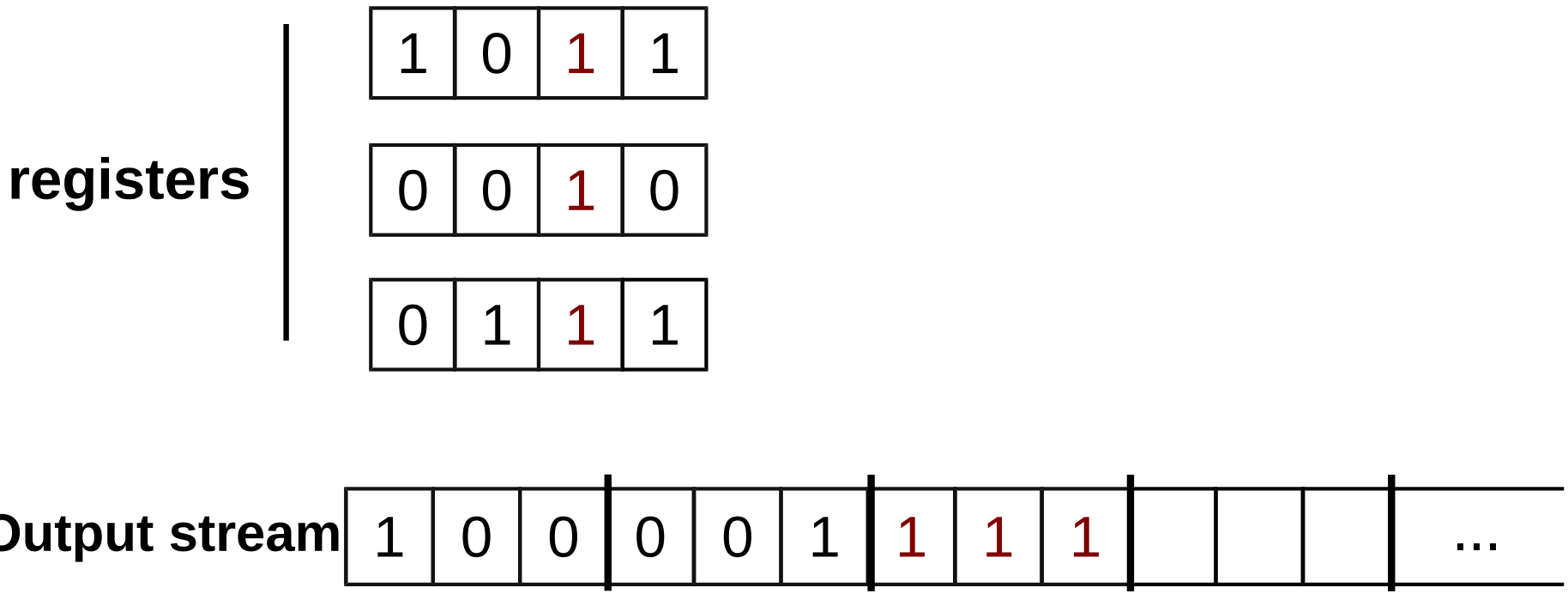
Bit-level parallelism



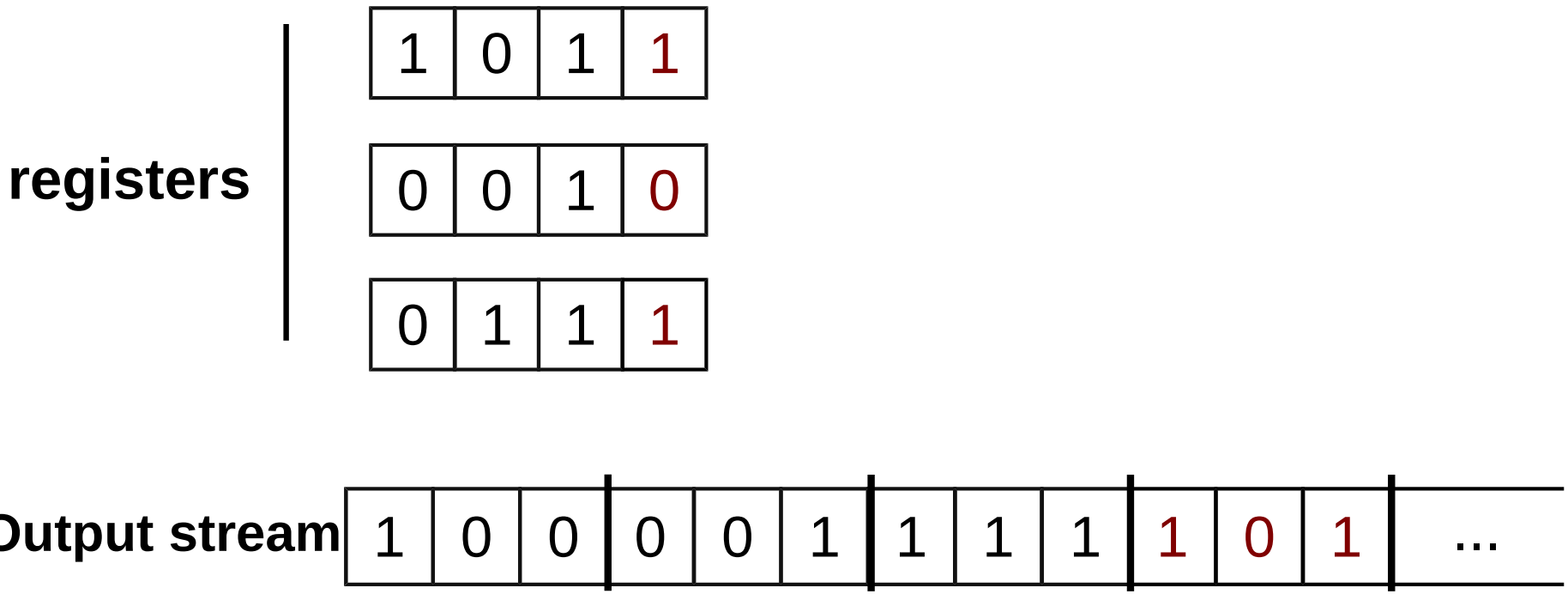
Bit-level parallelism



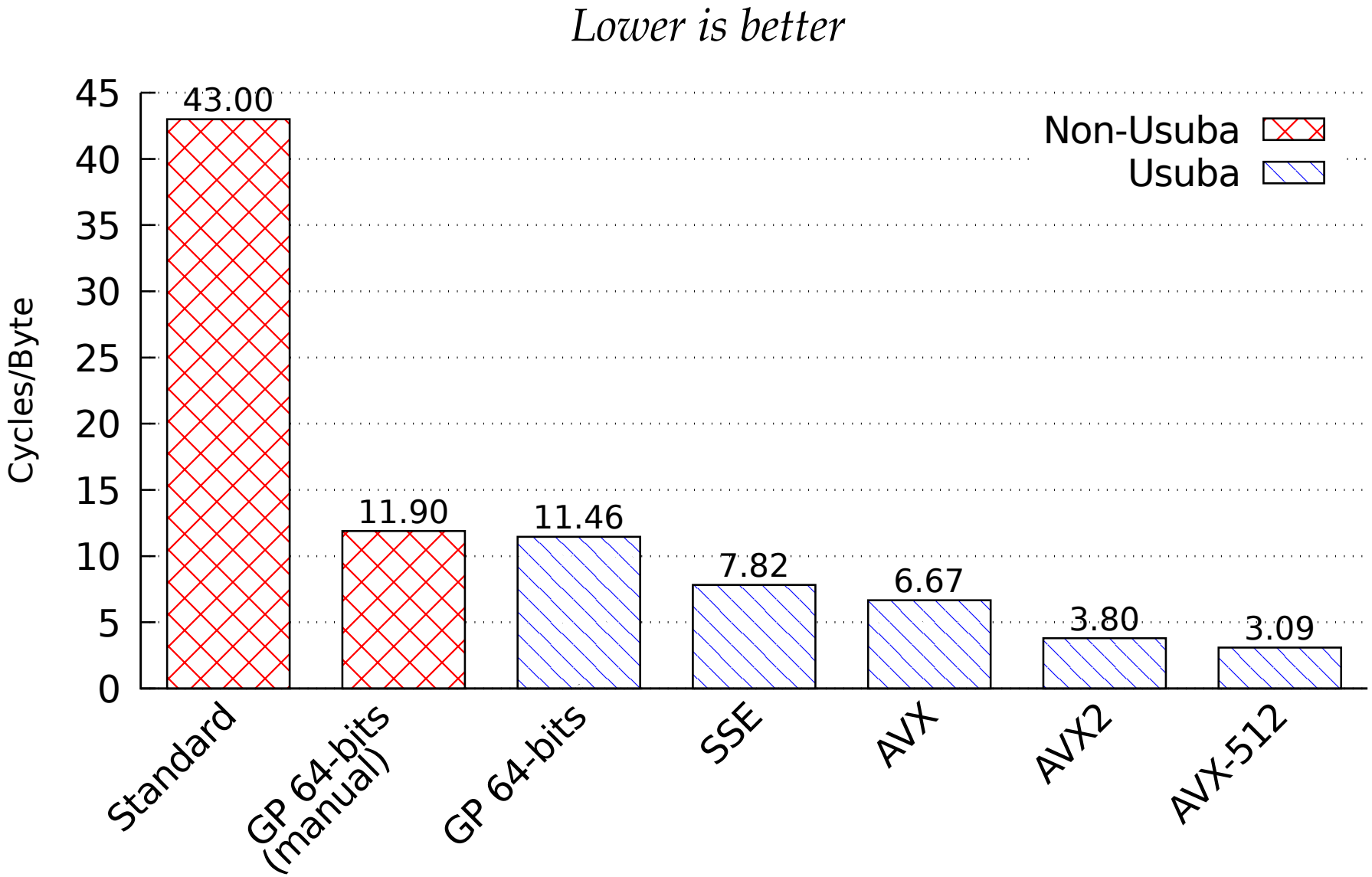
Bit-level parallelism



Bit-level parallelism



Scaling DES



D. Mercadier et al., Usuba, Optimizing Trustworthy Bitslicing Compiler, WPMVP'18

Normalization

Variable expansion

```
node dummy (a:u4,b:u4) returns (d:u4)
let
  d = c <<< 2;
  c = a ^ b
tel
```

Normalization

Variable expansion

```
node dummy (a:u4,b:u4) returns (d:u4)
let
  d = c <<< 2;
  c = a ^ b
tel
```

Normalization

Variable expansion

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c1,c2,c3,c4) <<< 2;
  (c1,c2,c3,c4) = (a1,a2,a3,a4) ^ (b1,b2,b3,b4)
tel
```

Normalization

Operator unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c1,c2,c3,c4) <<< 2;
  (c1,c2,c3,c4) = (a1,a2,a3,a4) ^ (b1,b2,b3,b4)
tel
```

Normalization

Operator unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c1,c2,c3,c4) <<< 2;
  (c1,c2,c3,c4) = (a1^b1,a2^b2,a3^b3,a4^b4)
tel
```

Normalization

Rotation unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c1,c2,c3,c4) <<< 2;
  (c1,c2,c3,c4) = (a1^b1,a2^b2,a3^b3,a4^b4)
tel
```

Normalization

Rotation unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c3,c4,c1,c2);
  (c1,c2,c3,c4) = (a1^b1,a2^b2,a3^b3,a4^b4)
tel
```

Normalization

Tuple unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  (d1,d2,d3,d4) = (c3,c4,c1,c2);
  (c1,c2,c3,c4) = (a1^b1,a2^b2,a3^b3,a4^b4)
tel
```

Normalization

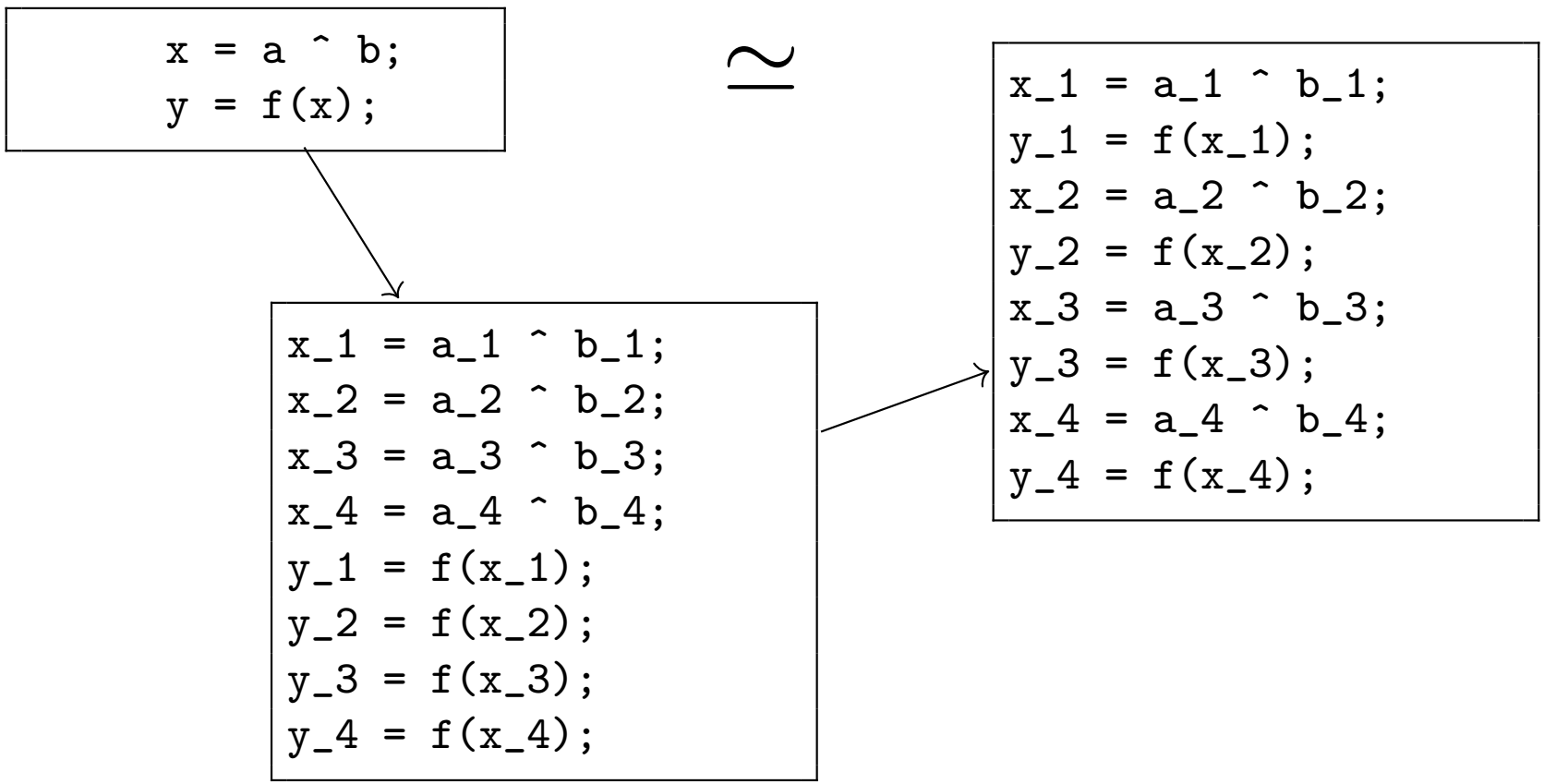
Tuple unfolding

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  d1 = c3;
  d2 = c4;
  d3 = c1;
  d4 = c2;
  c1 = a1^b1;
  c2 = a2^b2;
  c3 = a3^b3;
  c4 = a4^b4
tel
```

Optimizations

Scheduling

Idea: remove "chunks" of instructions to minimize live ranges



Optimizations

Interleaving

Idea: Interleave several encryptions to reduce pipeline stalls

```
y_0 = x_0 ^ x_1;
y_1 = y_0 ^ x_2;
y_2 = y_1 ^ x_3;
y_3 = y_2 ^ x_4;
```



```
y_0  = x_0  ^ x_1;
  y_0' = x_0' ^ x_1';
y_1  = y_0  ^ x_2;
  y_1' = y_0' ^ x_2';
y_2  = y_1  ^ x_3;
  y_2' = y_1' ^ x_3';
y_3  = y_2  ^ x_4;
  y_3' = y_2' ^ x_4';
```

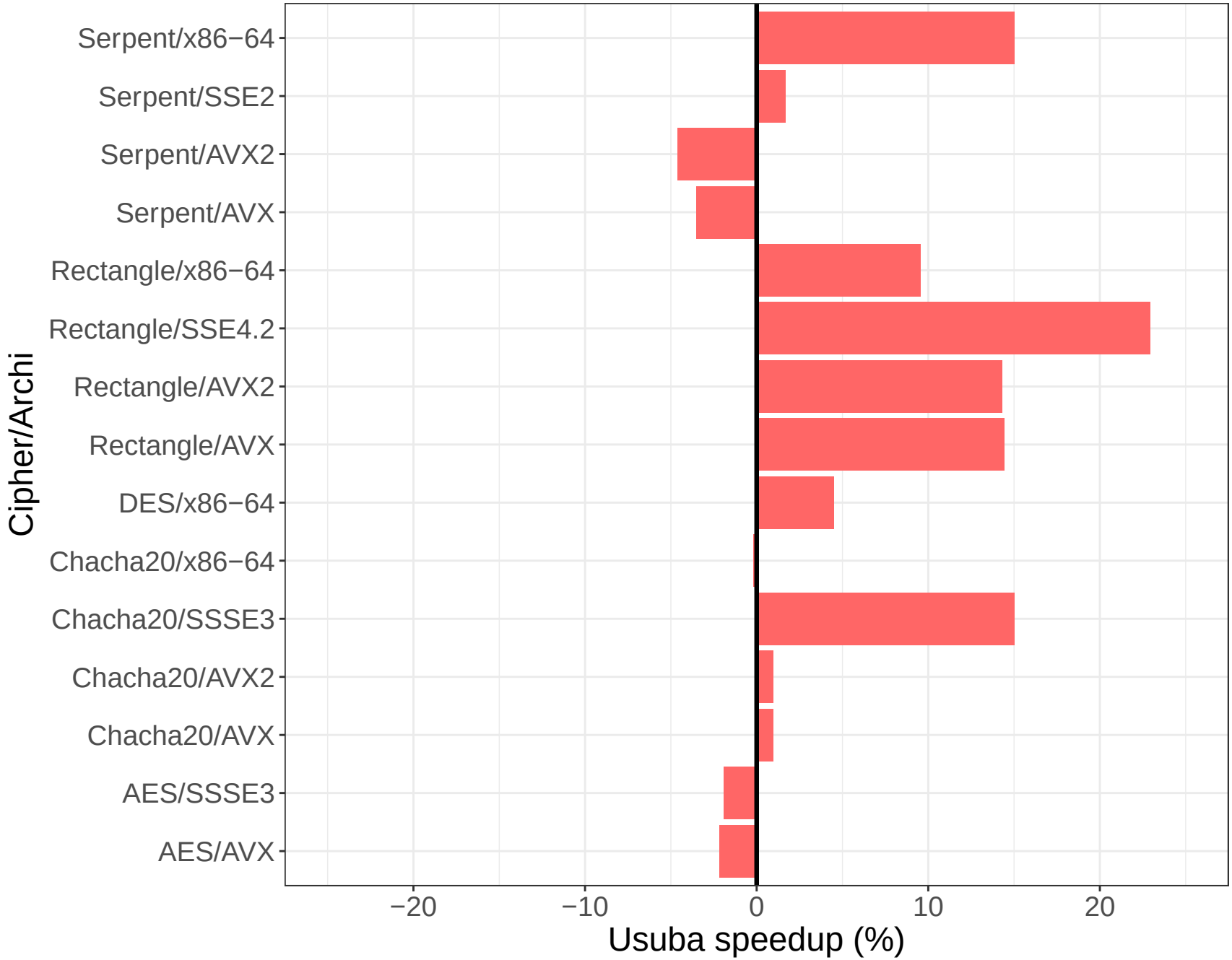
Transpiling to C

```
node dummy (a1,a2,a3,a4,b1,b2,b3,b4:bool)
    returns (d1,d2,d3,d4:bool)
let
  c1 = a1^b1;
  d1 = c3;
  c2 = a2^b2;
  d2 = c4;
  c3 = a3^b3;
  d3 = c1;
  c4 = a4^b4;
  d4 = c2
tel
```

Transpiling to C

```
void dummy (__m256i a1,__m256i a2,__m256i a3,__m256i a4,
            __m256i b1,__m256i b2,__m256i b3,__m256i b4,
            __m256i*d1,__m256i*d2,__m256i*d3,__m256i*d4) {
  c1 = a1^b1;
  *d3 = c1;
  c2 = a2^b2;
  *d4 = c2;
  c3 = a3^b3;
  *d1 = c3;
  c4 = a4^b4;
  *d2 = c4;
}
```

Performance: Usuba *vs.* Reference



SKIVA

(from Old Norse *skífa*, a slice.)

Threat model:

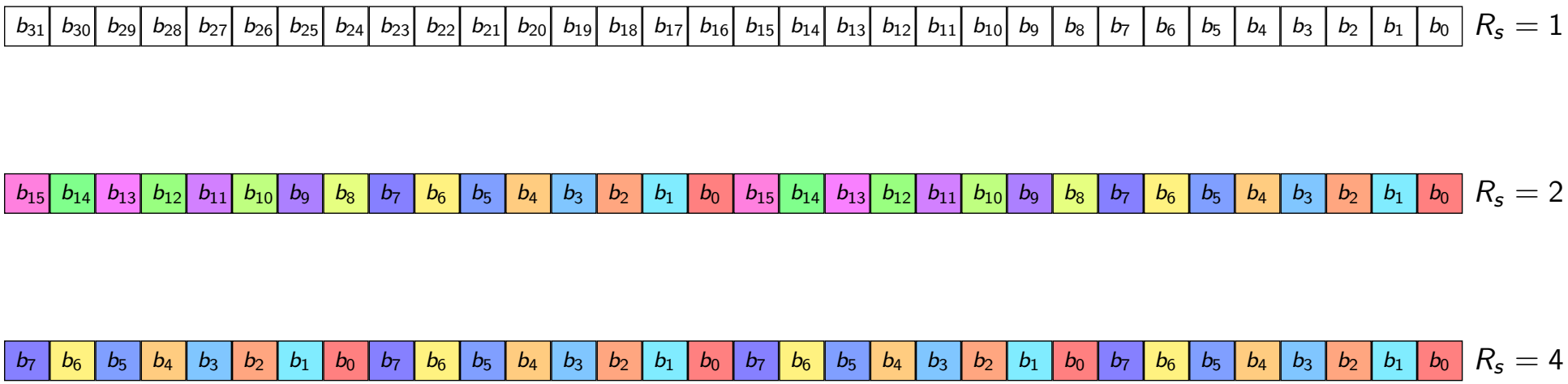
- time-based SCA $\rightarrow$ bitslicing
- electromagnetic SCA $\rightarrow$ higher-order masking
- data faults $\rightarrow$ spatial redundancy
- control faults $\rightarrow$ temporal redundancy

$\Rightarrow$ *Aggregated bitslicing model*

in collaboration with Pantea Kiaei, Patrick Schaumont (Virginia Tech) and Karine Heydemann (LIP6)

Intra-instruction redundancy

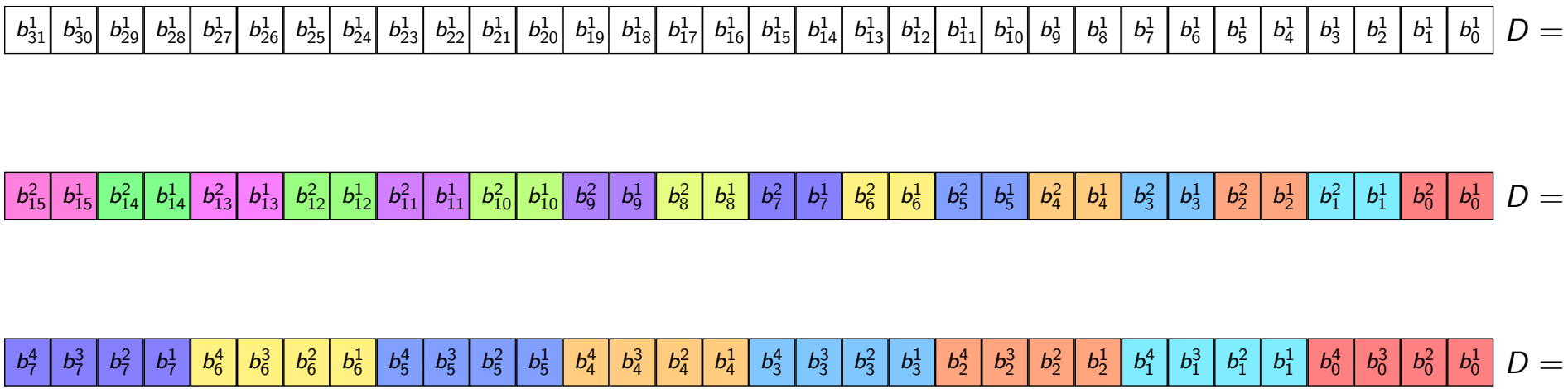
$R_s \in \{1, 2, 4\}$



$\Rightarrow$ *Duplicate the same data to R_s slices*

Higher-order masking

$D \in \{1, 2, 4\}$



$$b_i = \bigoplus_{1 \leq j \leq D} b_i^j$$

$\Rightarrow$ *Spread b_i into D shares*

"Parallel Implementations of Masking Schemes", 2016, Barthe et al.

Security, compositionally

$\Rightarrow$ *Static allocation of slices!*

Conclusion

Usuba:

- High-level description of combinational circuits
- Generates **optimized** C code
- Bitslicing as a programming model

Future work:

- Optimizations: counter caching, masking...
- Certified compiler, circuit equivalence
- Compiling for embedded systems

`https://github.com/DadaIsCrazy/usuba`

Backup slides

Side-Channel and/or Fault countermeasures

Collaboration with Pantea Kiaei, Patrick Schaumont (Virginia Tech) and Karine Heydemann (LIP6)

SKIVA

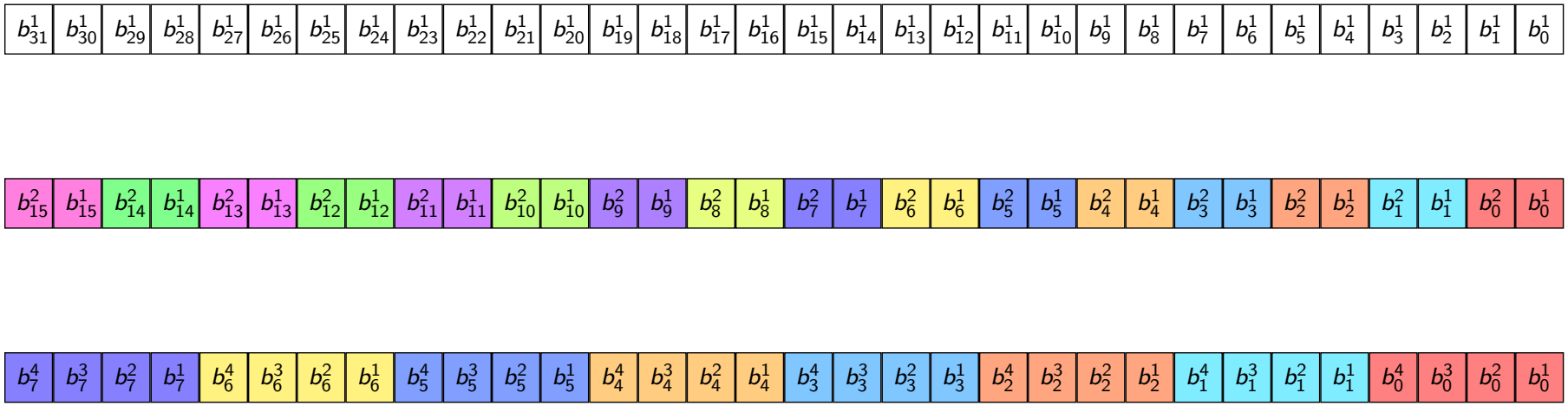
Skiva (c.): from Old Norse *skífa*, a slice.

Threat model:

- time-based SCA $\rightarrow$ bitslicing
- electromagnetic SCA $\rightarrow$ higher-order masking
- data faults $\rightarrow$ spatial redundancy
- control faults $\rightarrow$ temporal redundancy

$\Rightarrow$ *aggregated bitslicing model*

Higher-order masking ($D \in \{1, 2, 4\}$)

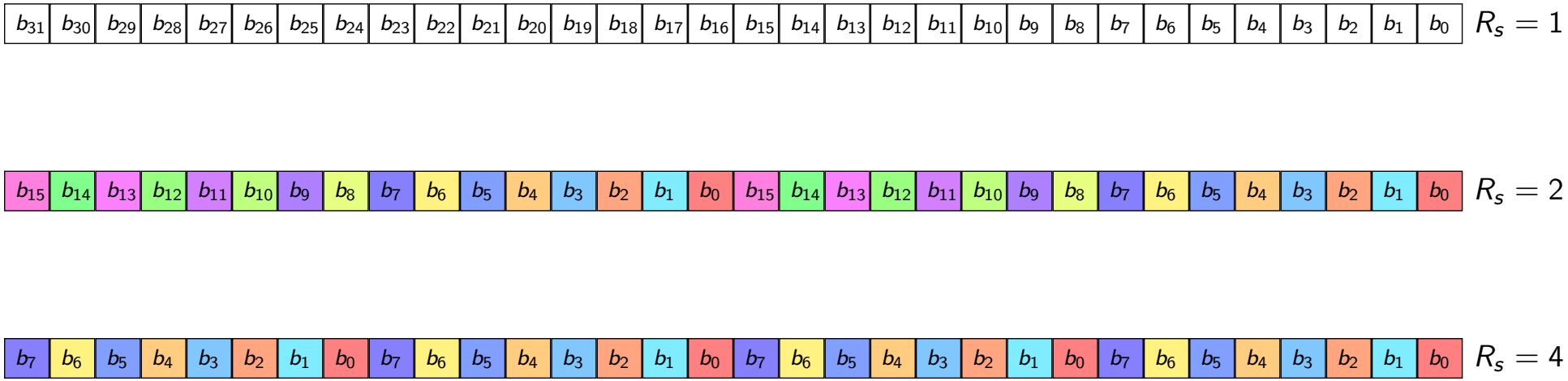



```
# input: %i2 (a), %i3 (b),
#        %i4 (random)
# output: %o0
AND         %i3, %i2, %o5
SUBROT      %i2,   2, %l0
AND         %i3, %l0, %o3
XOR         %o5, %i4, %o2
XOR         %o2, %o3, %o1
SUBROT      %i4,   2, %l1
XOR         %o1, %l1, %o0
```

2^{nd} order protected multiplication

G. Barthe et al.,
Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model,
2016

Intra-instruction redundancy ($R_s \in \{1, 2, 4\}$)



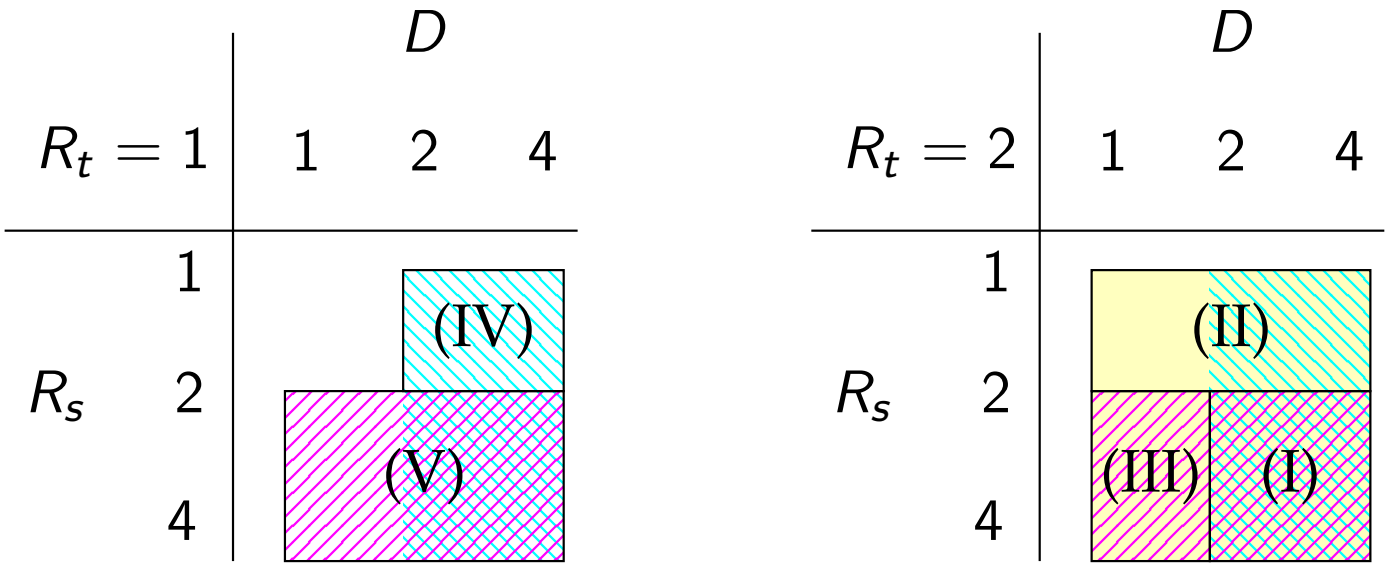
Instruction format	Semantics
`TR2 rs1, rs2, rd`	Normal → Bitslice
`INVTR2 rs1, rs2, rd`	Bitslice → Normal
`RED rs, imm, rd`	Redundancy Generation
`FTCHK rs, imm, rd`	Redundancy Checking
`ANDC16 rs1, rs2, rd`	Redundant AND (R=2)
`XORC16 rs1, rs2, rd`	Redundant XOR (R=2)
`ANDC8 rs1, rs2, rd`	Redundant AND (R=4)
`XORC8 rs1, rs2, rd`	Redundant XOR (R=4)

Combined masking and intra-instruction redundancy

Temporal redundancy ($R_t \in \{1, 2\}$)

```
void AES_secure(uint plain[128], uint keys[11][128],
                uint cipher[128]) {
    uint state[128];
    // Aggregated bitslice ‘state‘: plain and first round
    init_round(state, plain, keys[0], keys[1]);
    // Data-duplicated loop counter, increment and guard
    int round_cpt = 1 | (1 << 4);
    const int incr = 1 | (1 << 4);
    const int last_round = 9 | (9 << 4);

    // Duplicated loop structure
    while (1) {
        while (1) {
            // Retrieve key from duplicated round index
            uint[128] round_key = load_key(keys, round_cpt);
            // Compute current and previous round in parallel
            AES_round_bitsliced(state, round_key, plain);
            // Check temporal redundancy
            check(state, plain);
            memcpy_secure(plain, state, 128*sizeof(uint));
            // Increment data-duplicated counter
            round_cpt += incr;
            // Duplicated loop exit
            if (round_cpt == last_round) break;
        }
        if (round_cpt == last_round) break;
    }
    // last round twice, checked for temporal redundancy
    (..)
}
```

SKIVA/Usuba: attacker model



higher-order masking

intra-instr. redundancy

temporal redundancy

I Cycle-accurate **and** bit-precise

II Cycle-accurate **or** bit-precise

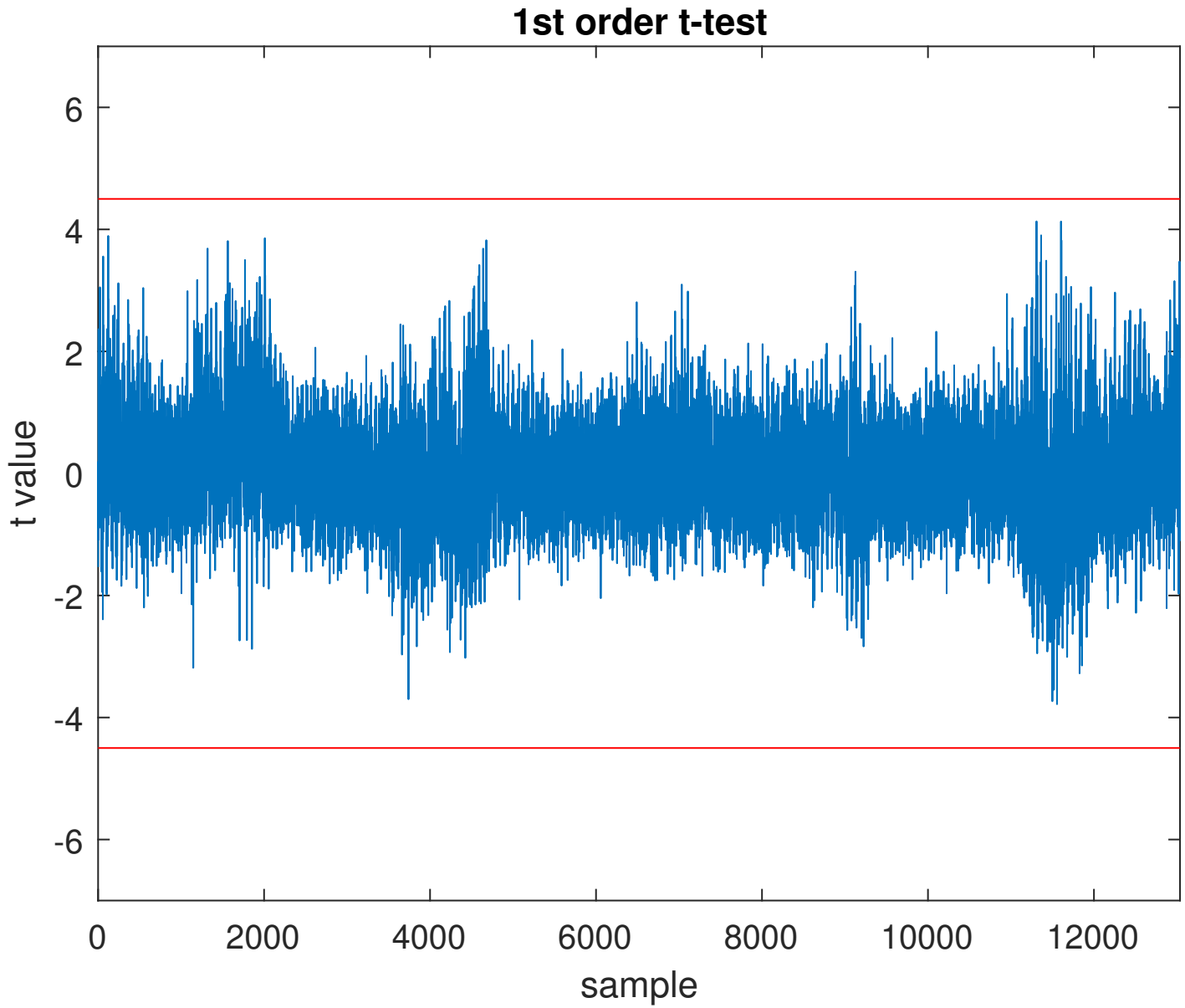
III Physical protection against CPA

IV Physical protection against faults

V Physical control-flow integrity

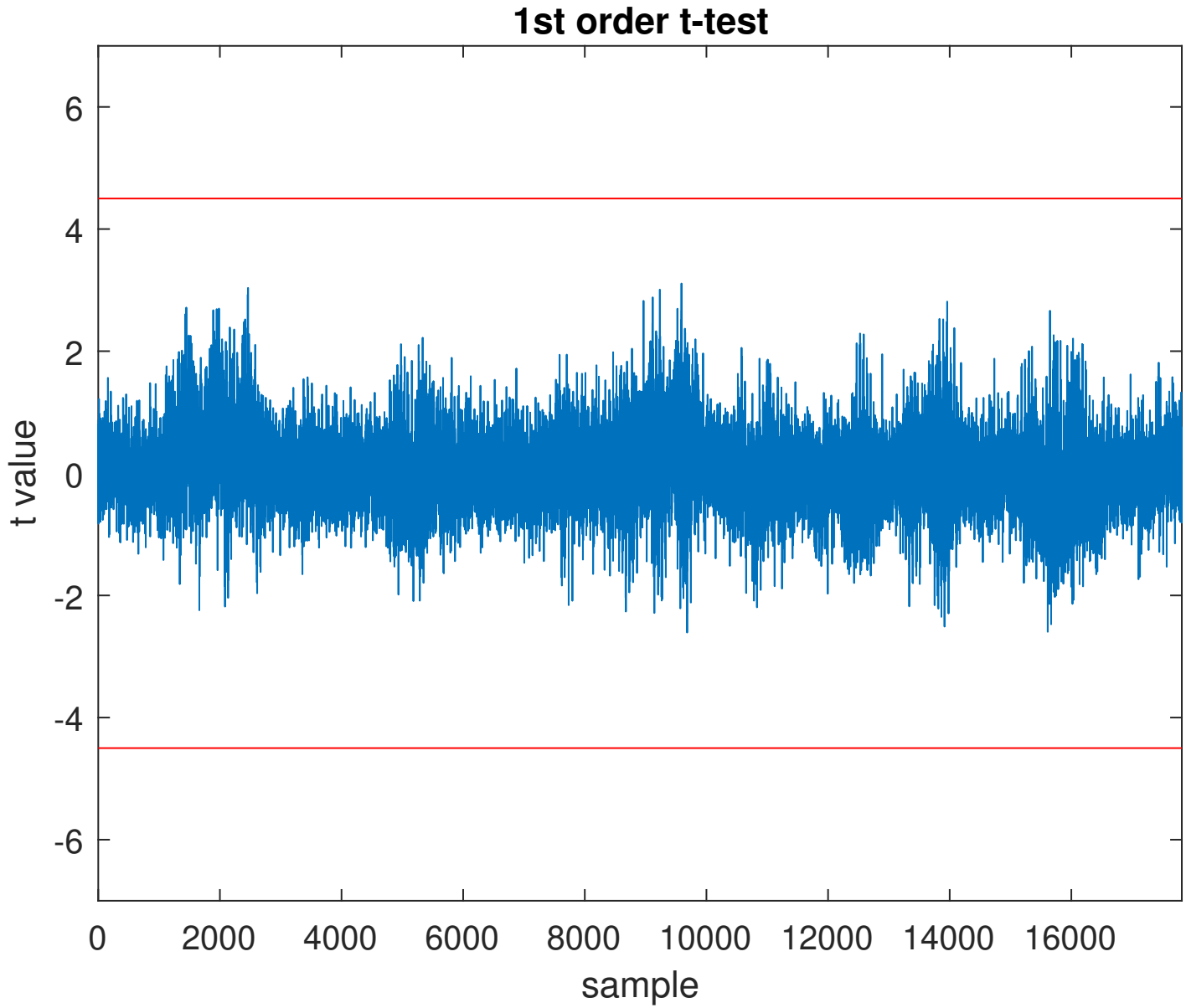
TVLA

1^{st} order masking - 1^{st} order t-test



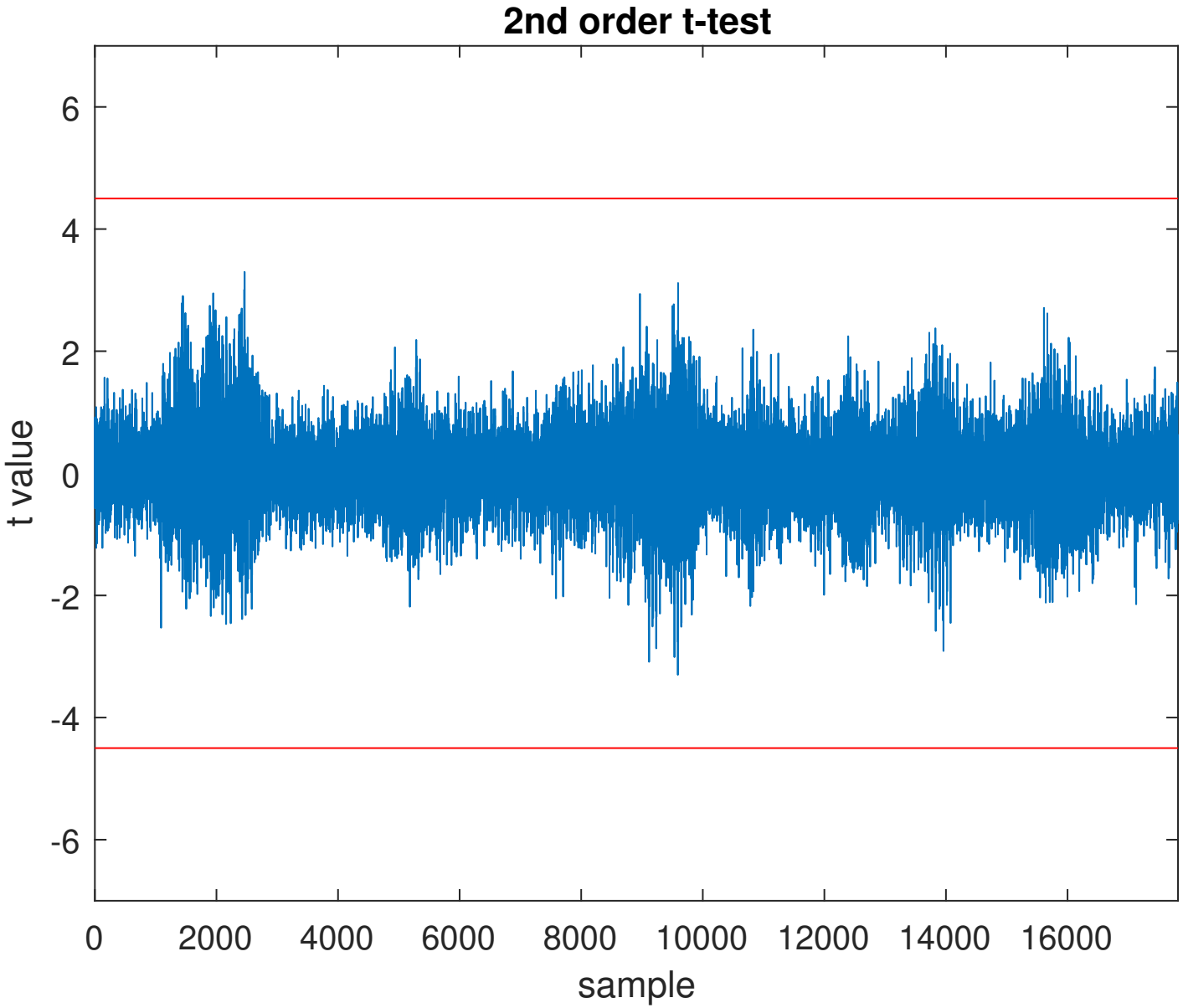
TVLA

3^{rd} order masking - 1^{st} order t-test



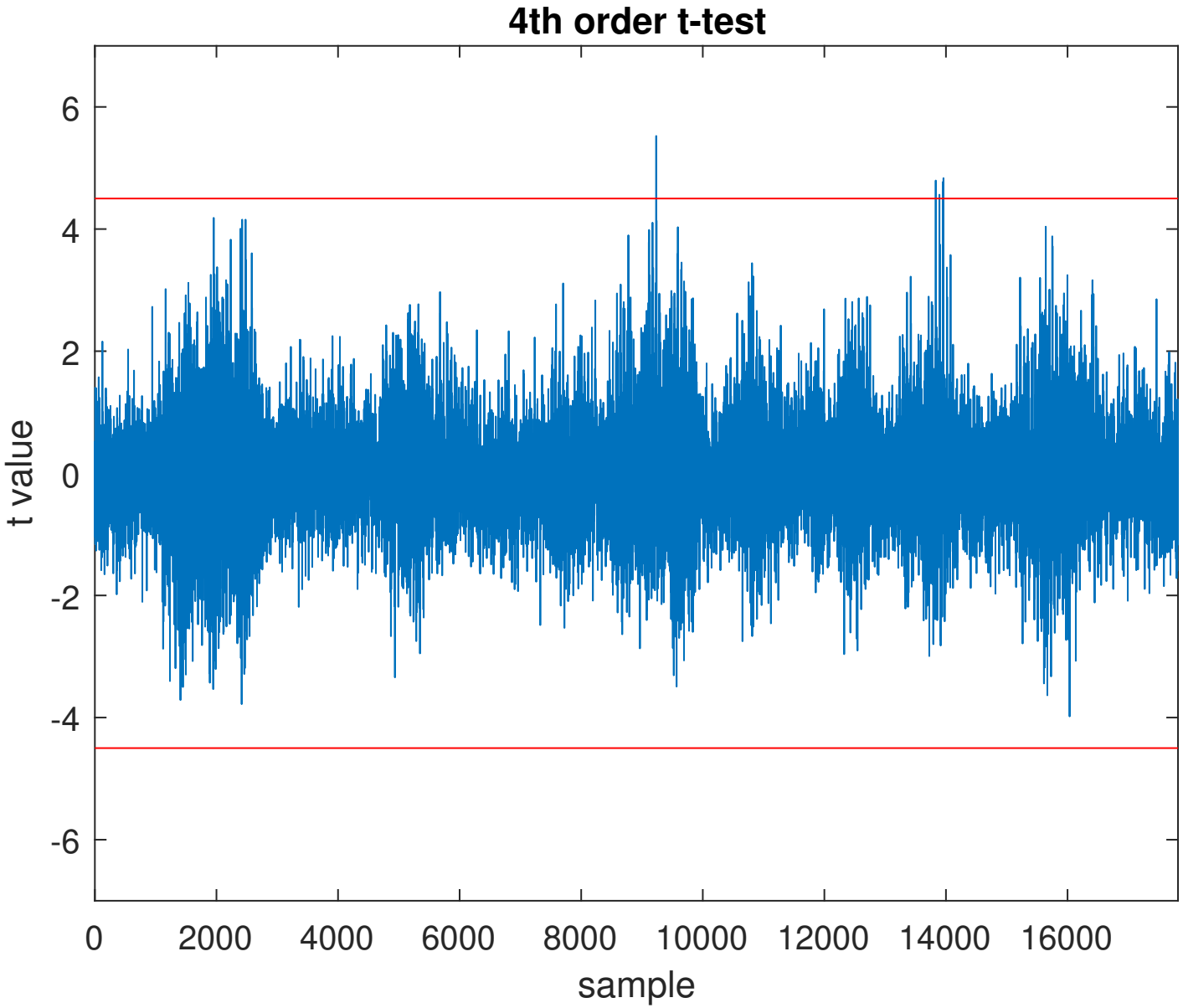
TVLA

3^{rd} order masking - 2^{nd} order t-test



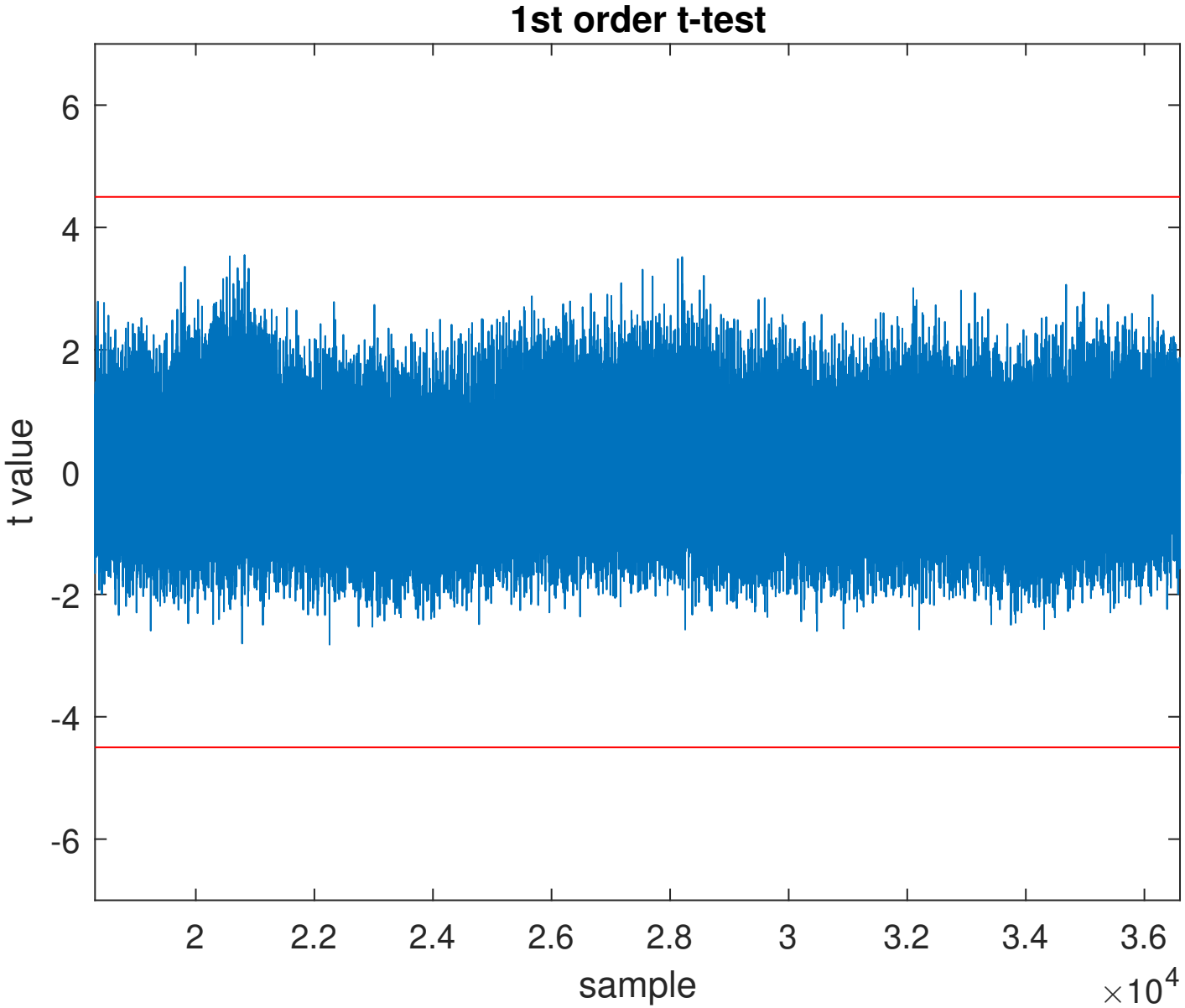
TVLA

3^{rd} order masking - 4^{th} order t-test



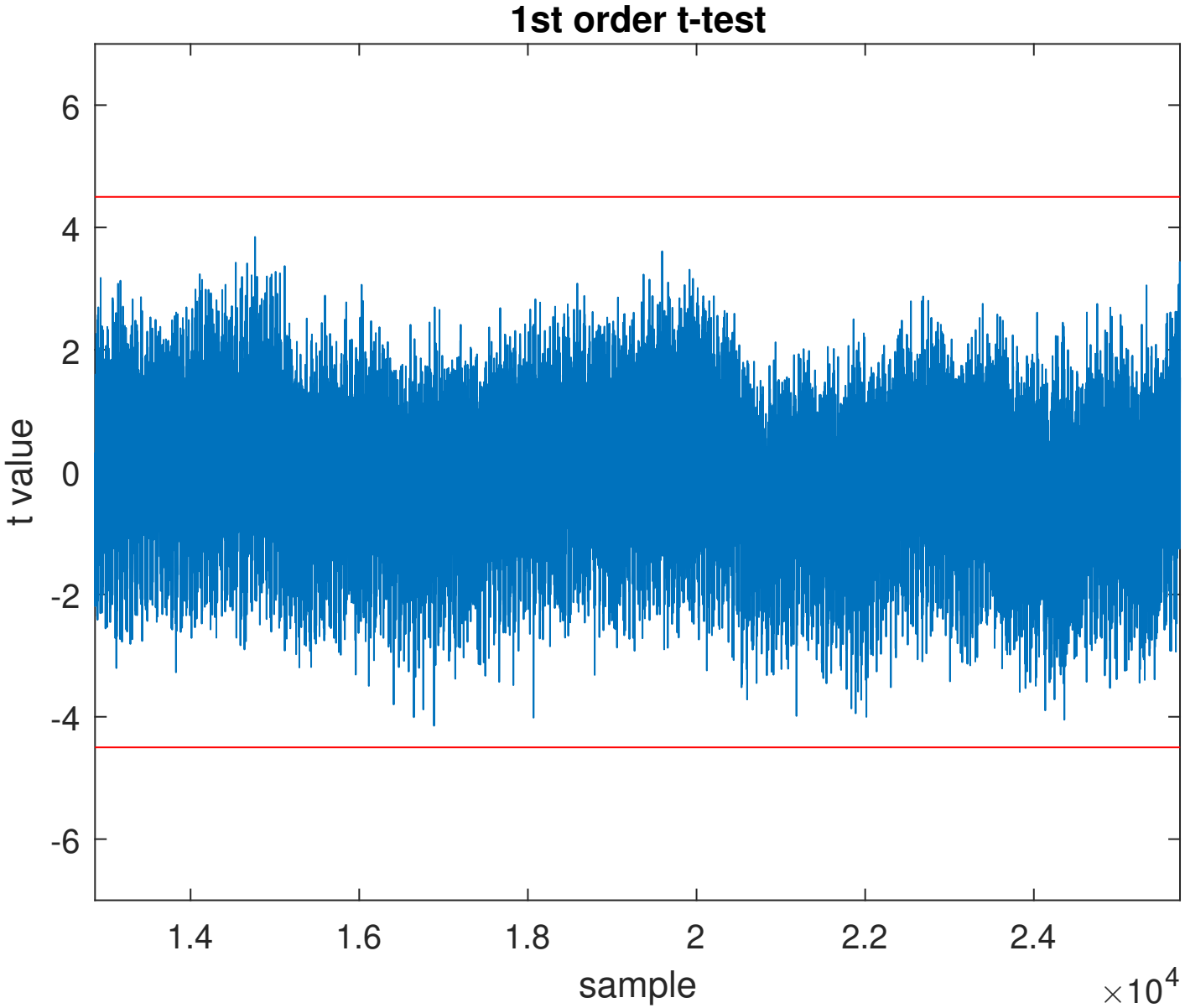
TVLA: complementary redundancy

1^{st} order masking - 1^{st} order t-test



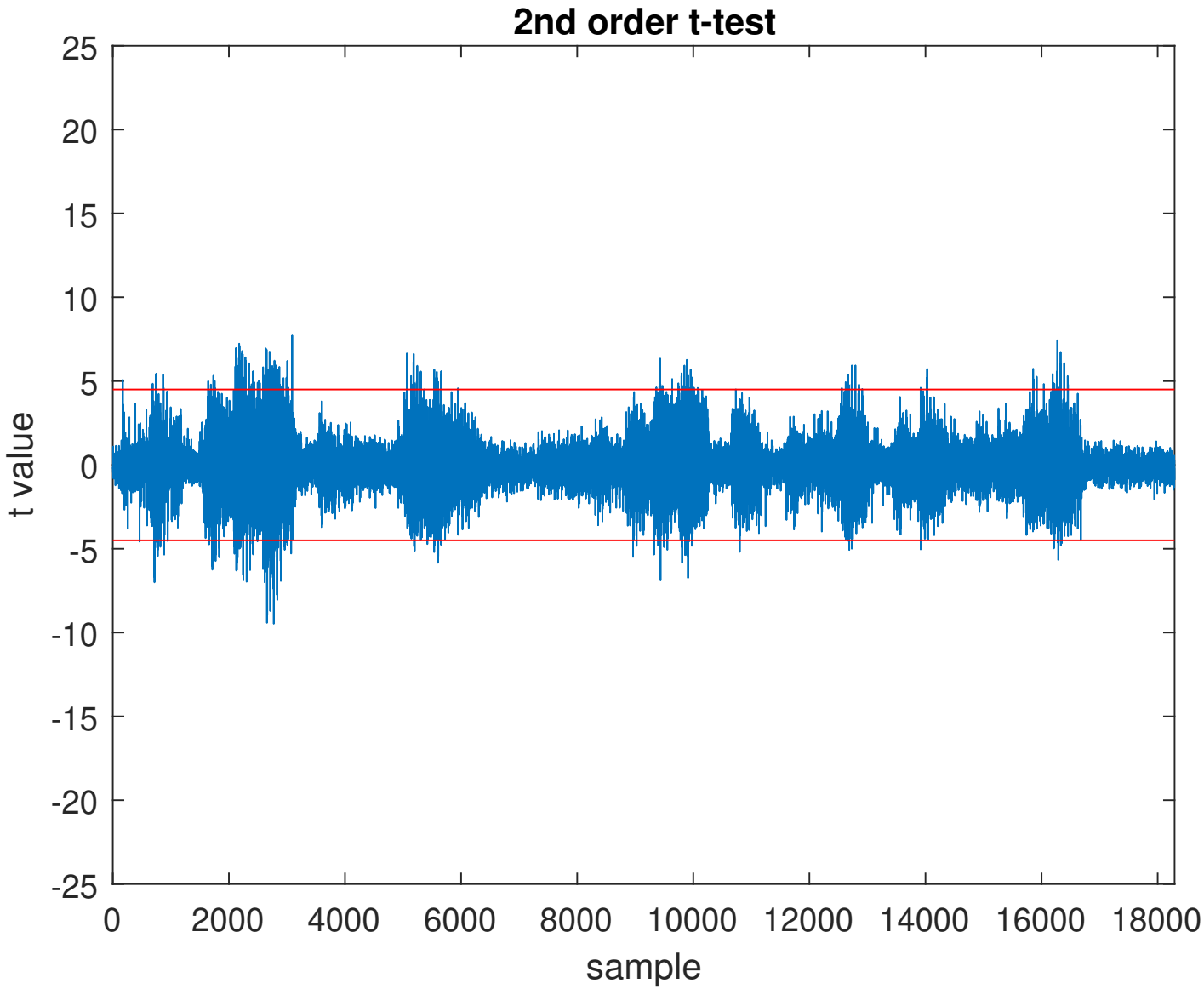
TVLA: direct redundancy

1^{st} order masking - 1^{st} order t-test



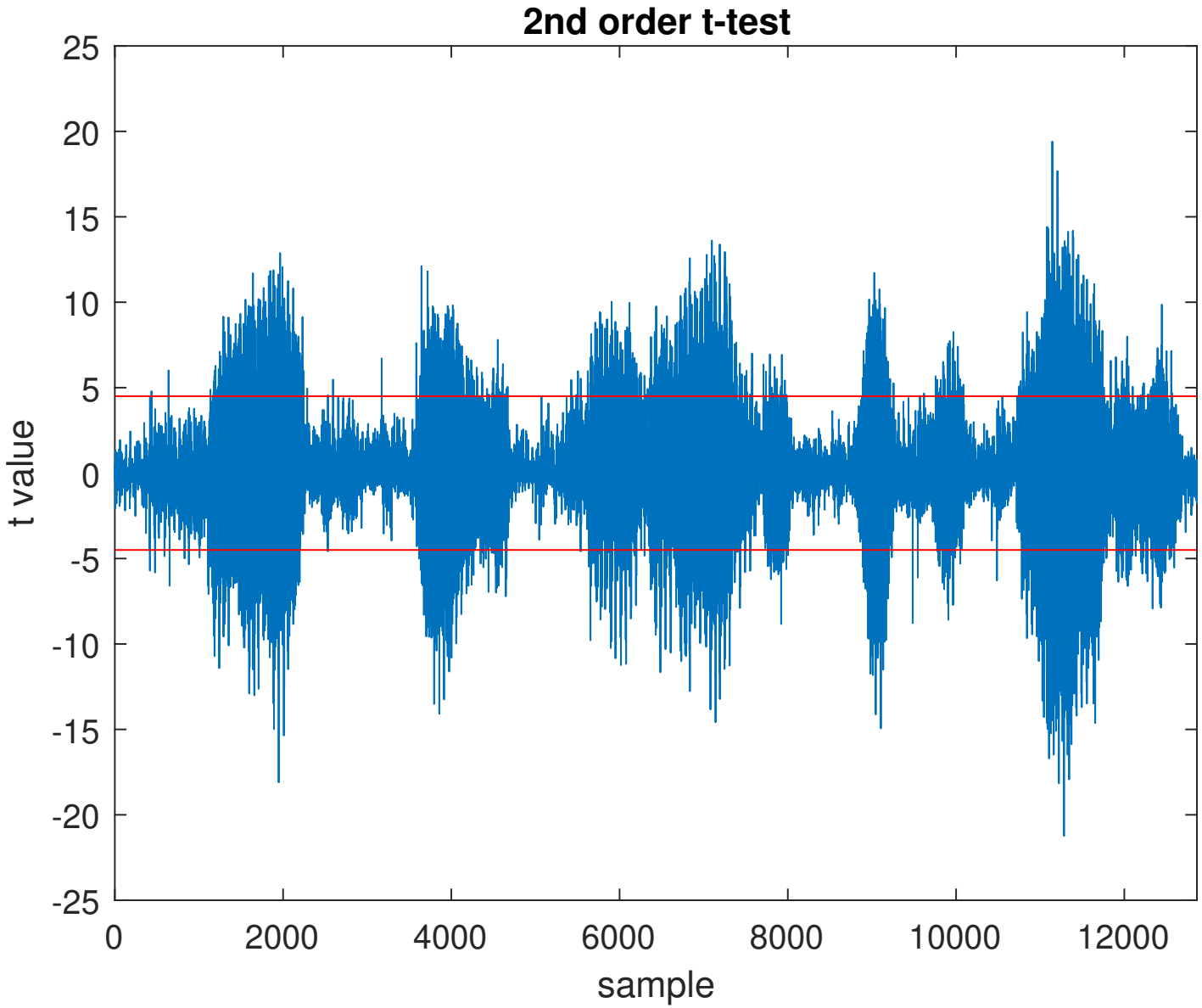
TVLA: complementary redundancy

1^{st} order masking - 2^{nd} order t-test



TVLA: direct redundancy

1^{st} order masking - 2^{nd} order t-test



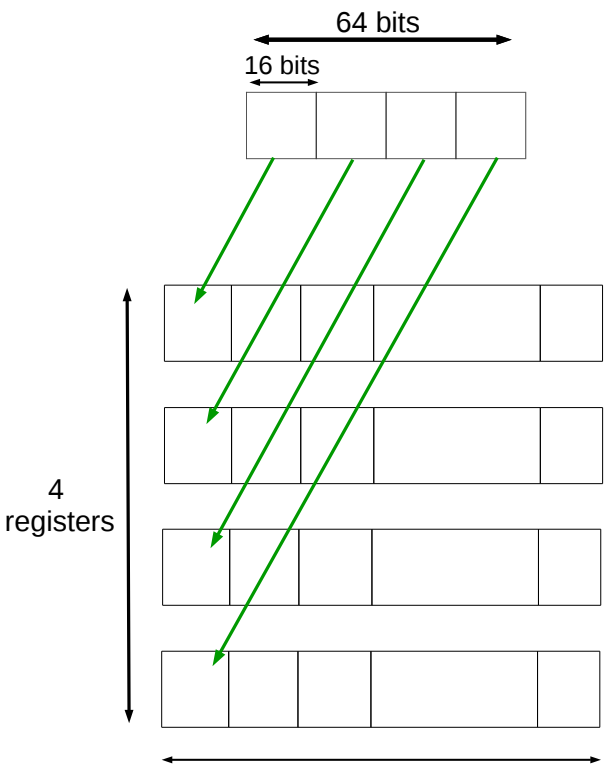
Performance

$R_t = 1$		D		
		1	2	4
R_s	1	44 C/B	173 C/B	459 C/B
	2	89 C/B	408 C/B	1179 C/B
	4	169 C/B	767 C/B	2157 C/B

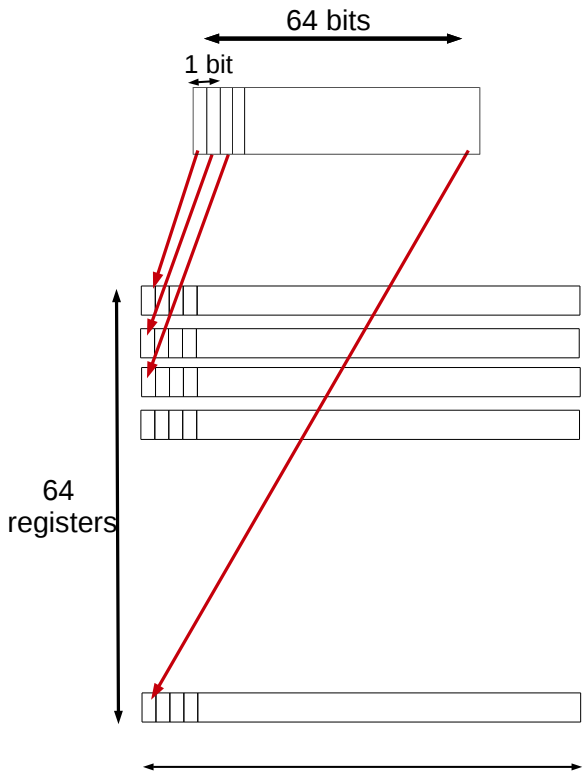
$R_t = 2$		D		
		1	2	4
R_s	1	127 C/B	447 C/B	1151 C/B
	2	261 C/B	1036 C/B	2874 C/B
	4	512 C/B	1972 C/B	5318 C/B

Usuba

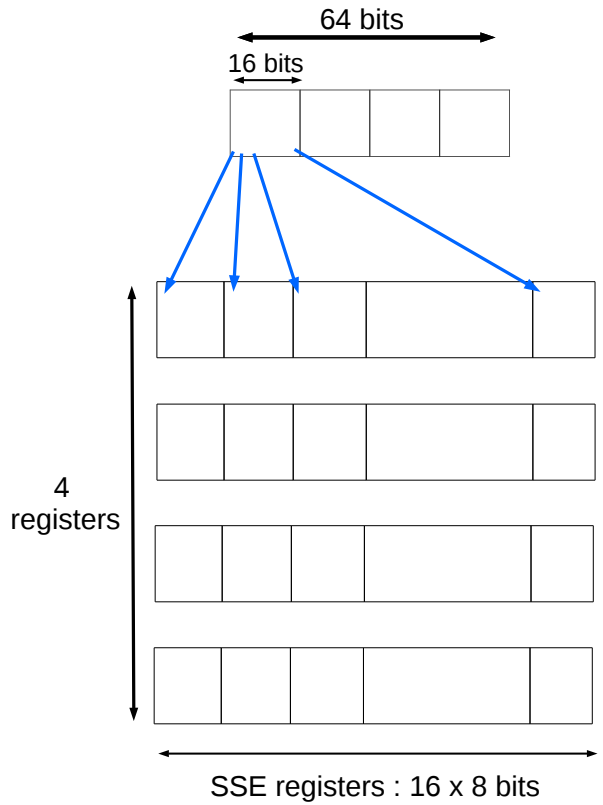
m-slicing



u_V16x4
(V-slicing)



u1x64
(bitslicing)



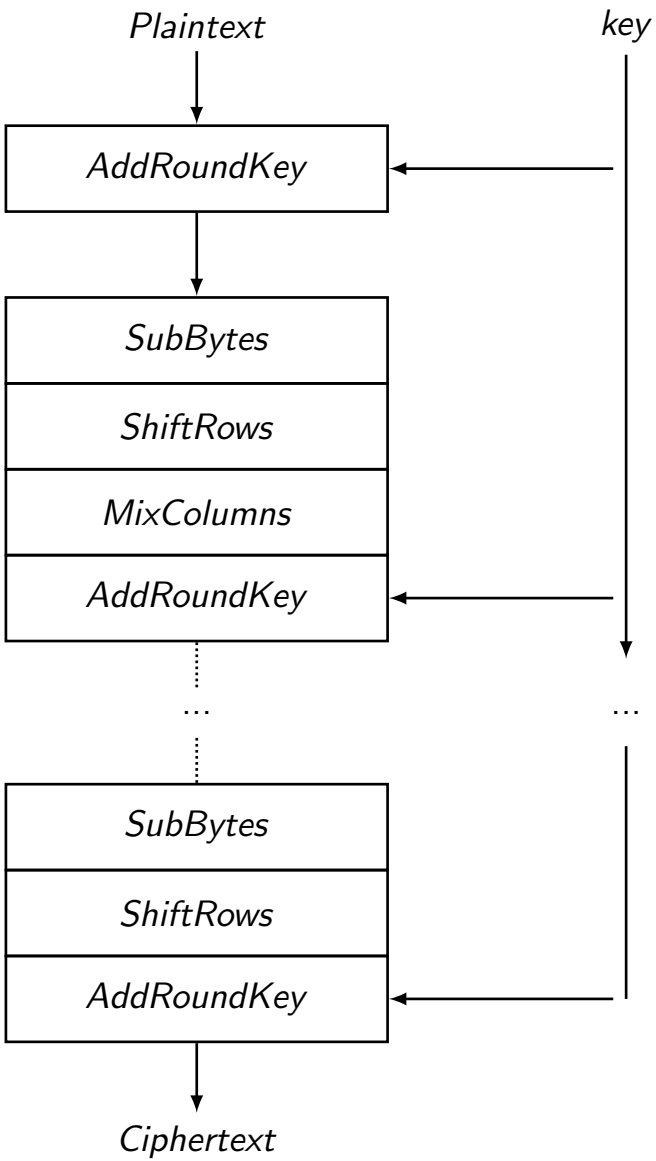
u_H16x4
(H-slicing)

Same Usuba code for each version, except for the types!

Rectangle in Usuba

```
table SubColumn (input:u4) returns (out:u4) {
    6 , 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3 , 13, 8, 15, 4, 2
}
node ShiftRows (input:u16x4) returns (out:u16x4)
let out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
node Rectangle (plain:u16x4,key:u16x4[26])
        returns (cipher:u16x4)
vars round : u16x4[26]
let
    round[0] = plain;
    forall i in [0,24] {
     round[i+1] = ShiftRows(SubColumn(round[i] ^ key[i]))
    }
    cipher = round[25] ^ key[25]
tel
```

Bitsliced AES




```
node AES (plain:u128,key:u128[11])
     returns (cipher:u128)
vars
    aes : u128[10]
let
    // Initial AddRoundKey
    aes[0] = plain ^ key[0];

    // 9 rounds (the last is special)
    forall i in [1,9] {
      aes[i] = MixColumn(
                  ShiftRows(
                    SubBytes( aes[i-1] )))
                ^ key[i]
    }

    // Last (10th) round (no MixColumn)
    cipher = ShiftRows(SubBytes(aes[9]))
             ^ key[10]
tel
```

Expressivity

Cipher	**Standard** (sloc)	**Bitslice** (sloc)	**Usuba** (sloc)
DES	220	400	125
AES	105	800	79
Serpent	88		38
Chacha20	43		64
Camellia	120	?	98
Rectangle	60		37

Table: Code size[1] of various ciphers.

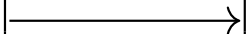
[1] only the block function, without key schedule

Optimizations

m-sliced code scheduling

Idea: increase Instruction Level Parallelism (ILP)

```
y_0 = x_0 ^ x_1;
y_1 = y_0 ^ x_2;
y_2 = y_1 ^ x_3;
y_3 = y_2 ^ x_4;
z_0 = pshufb(a_0,[0,2,1,3]);
z_1 = pshufb(a_1,[0,2,1,3]);
z_2 = pshufb(a_2,[0,2,1,3]);
z_3 = pshufb(a_3,[0,2,1,3]);
```



```
y_0 = x_0 ^ x_1;
z_0 = pshufb(a_0,[0,2,1,3]);
y_1 = y_0 ^ x_2;
z_1 = pshufb(a_1,[0,2,1,3]);
y_2 = y_1 ^ x_3;
z_2 = pshufb(a_2,[0,2,1,3]);
y_3 = y_2 ^ x_4;
z_3 = pshufb(a_3,[0,2,1,3]);
```

Evaluation

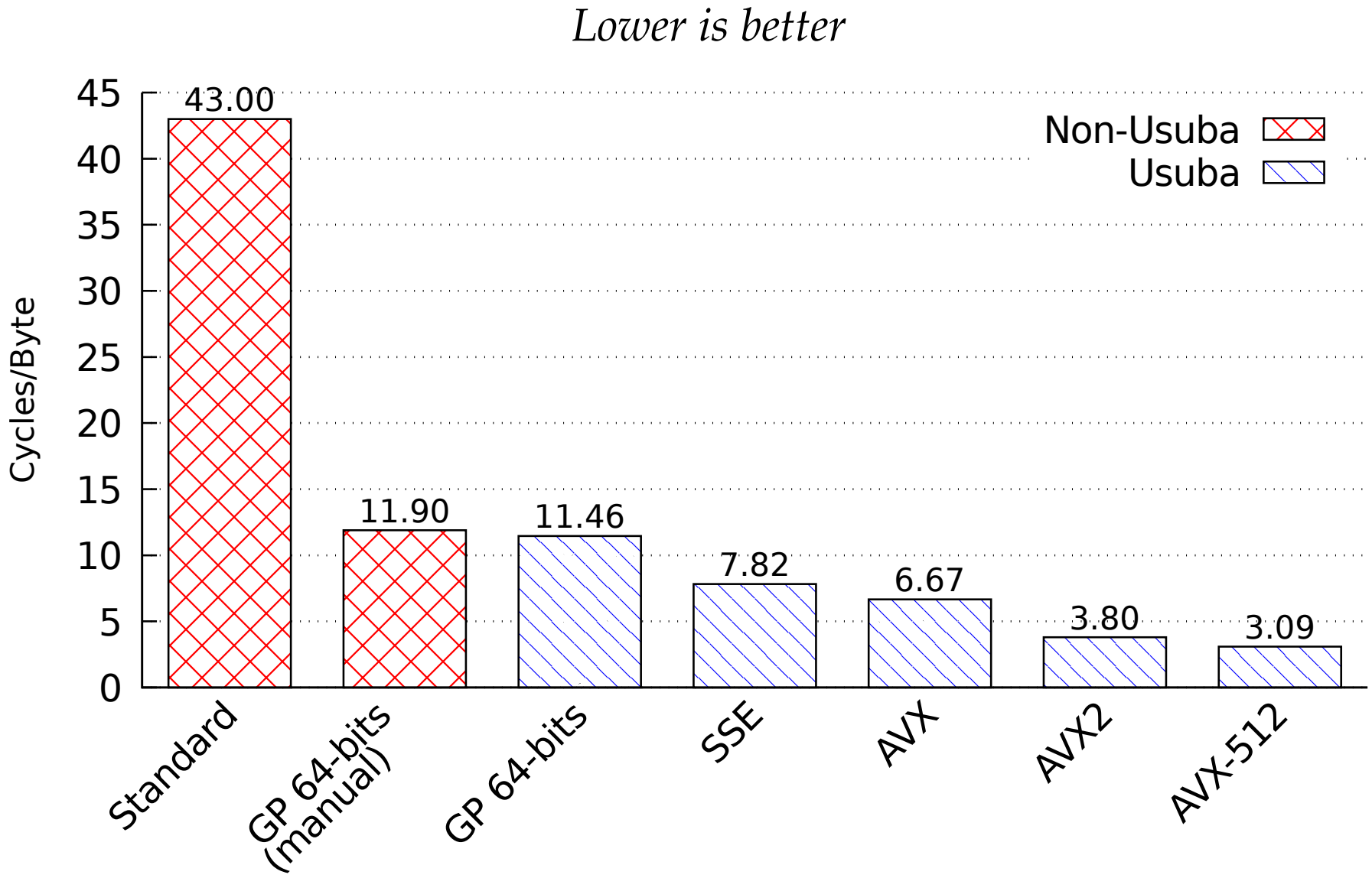
Backend architectures

Code name	CPU	Compiler	SIMD
SKL-X	Skylake i9-7900X	icc 17.0.2	SSE, AVX2, AVX-512
KBL	Kabylake i7-7500U	clang 4.0.1	SSE, AVX2
PowerPC	PowerPC 970MP	gcc 4.0.0	AltiVec
ARMv7	ARMv7	gcc 4.9.2	Neon

High-end Intel speedup

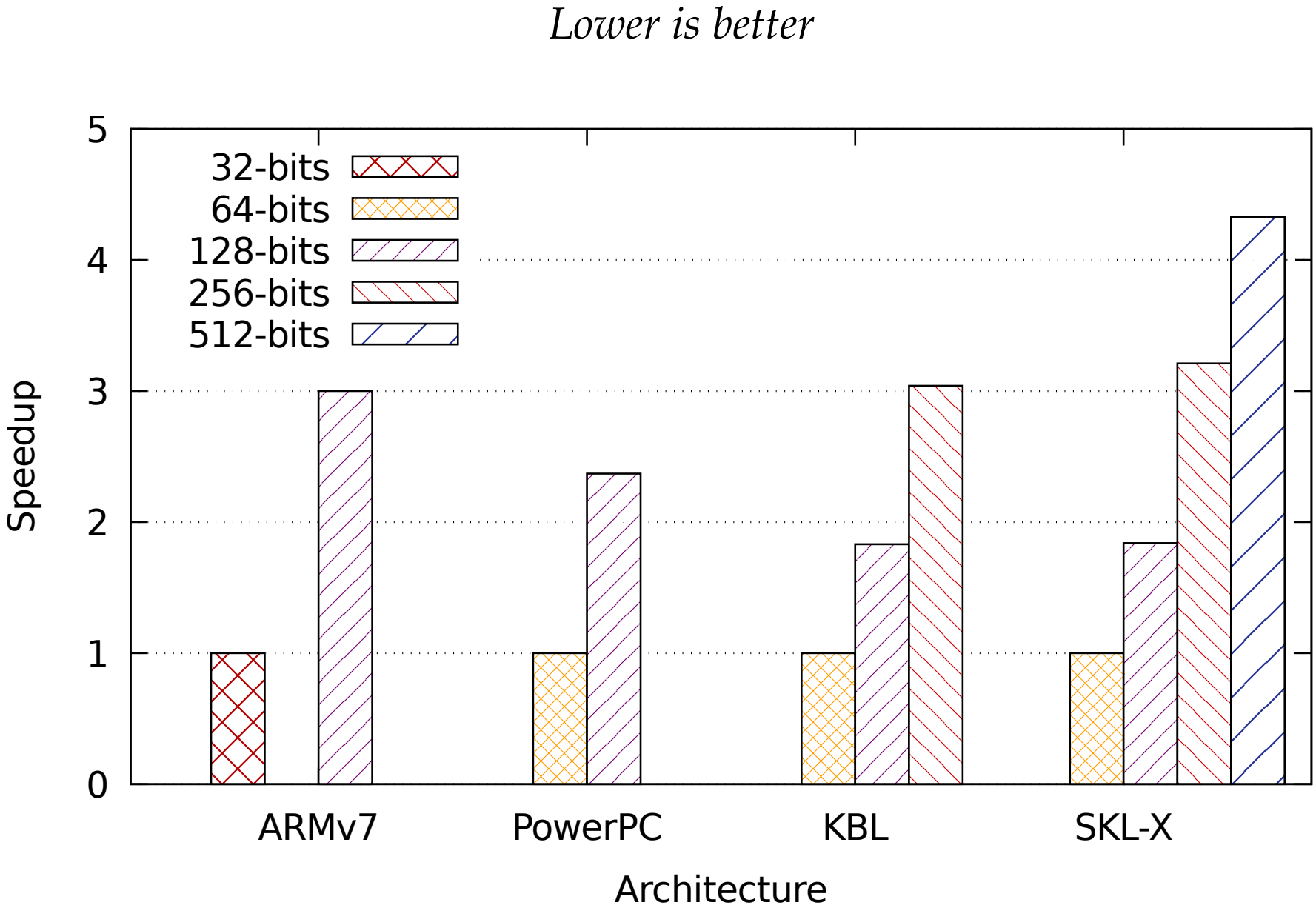
- L1 cache misses
- Throttling
- 2-operands (SSE) vs 3-operands instructions (AVX+)
- Number of registers (SSE: 8, AVX: 16, AVX512: 32)
- SIMD warm-up
- Number of ports / execution units of the CPU
- Instructions available (*e.g.* `vpternlog`)
- ...

Bitslice DES performance

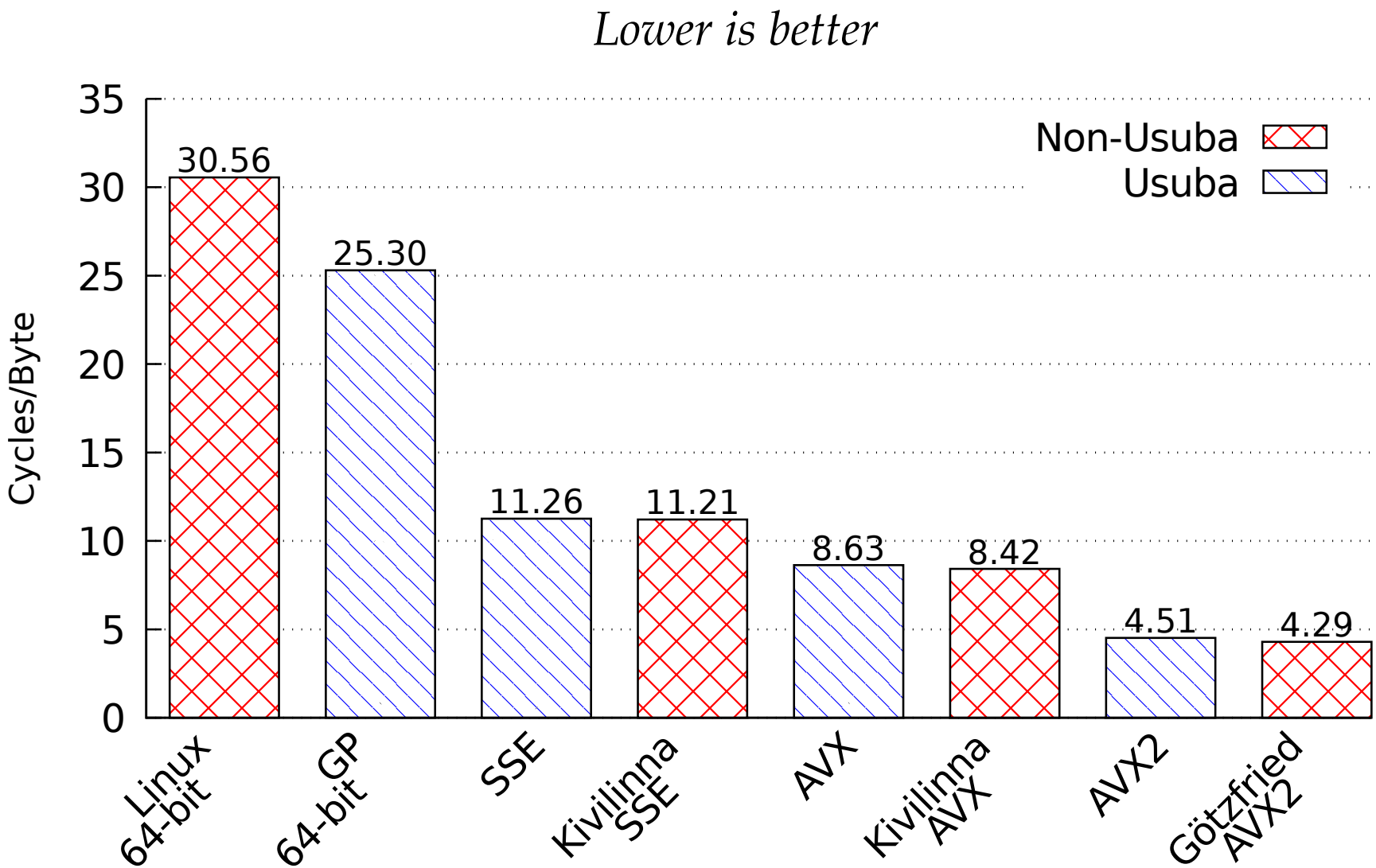


D. Mercadier et al., Usuba, Optimizing Trustworthy Bitslicing Compiler, WPMVP'18

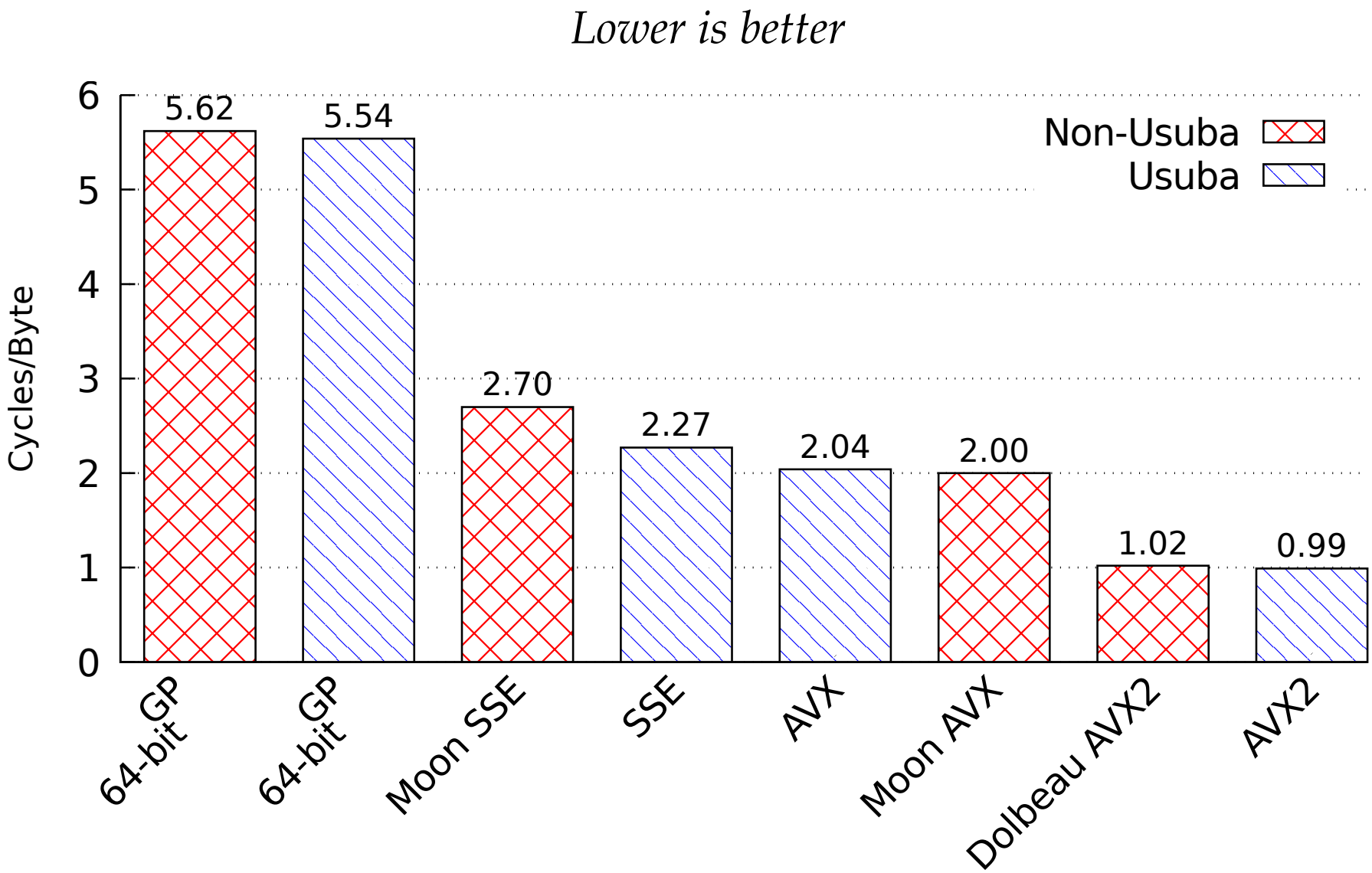
Bitslice DES scaling



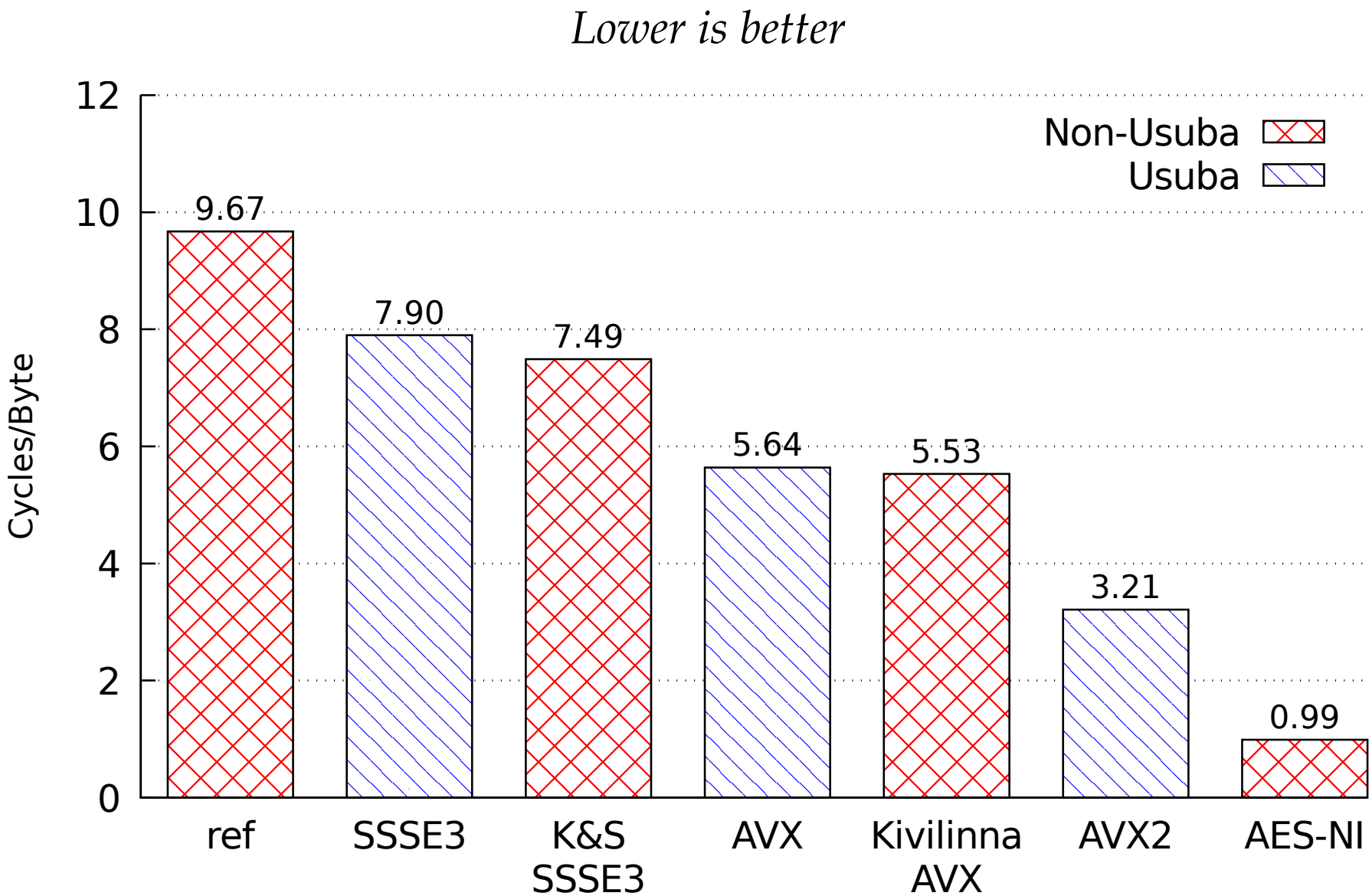
V-sliced Serpent performance



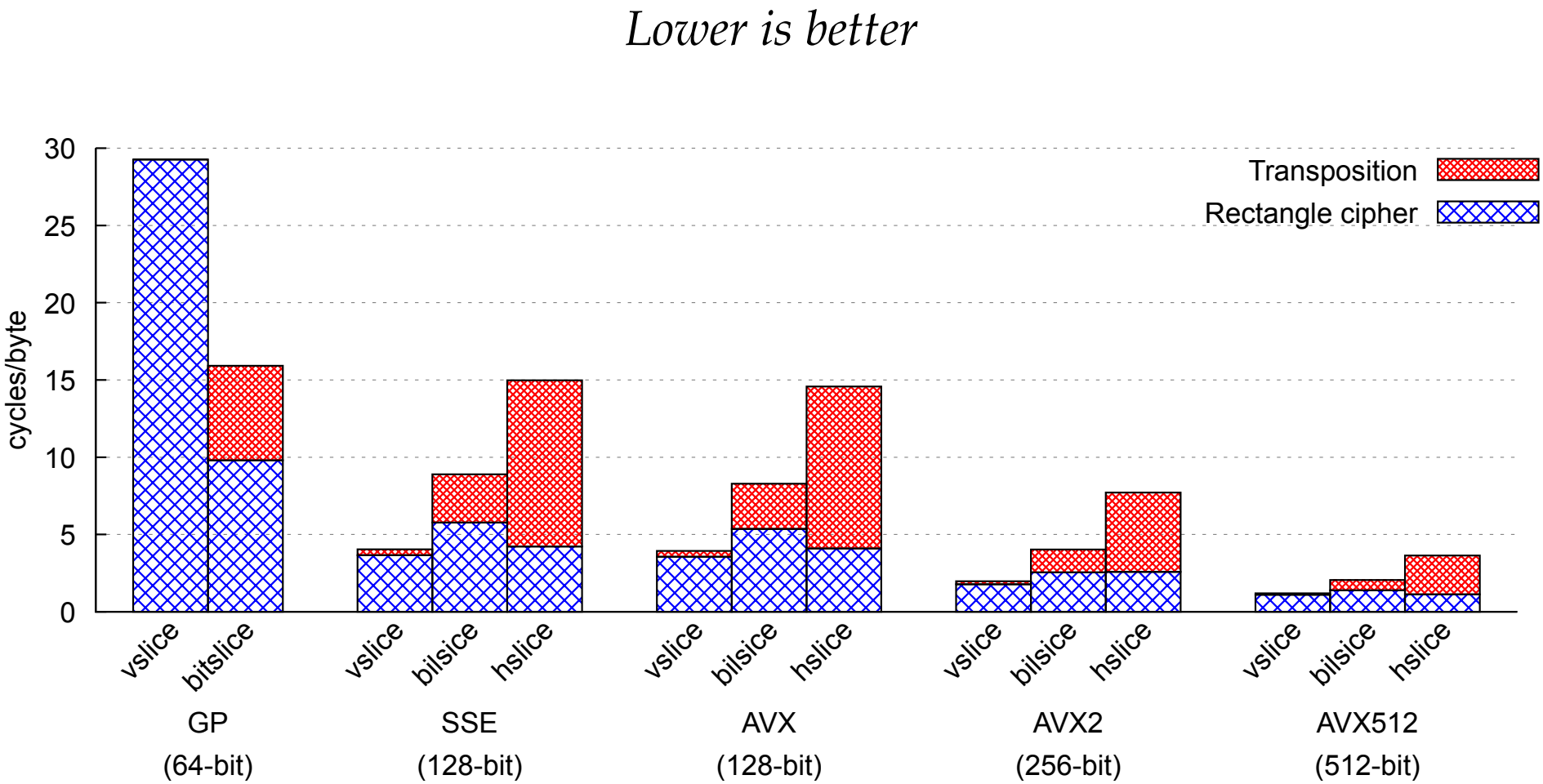
V-sliced Chacha20 performance



H-sliced AES performance



Rectangle: comparison of slicing modes



Transposition

Transposition

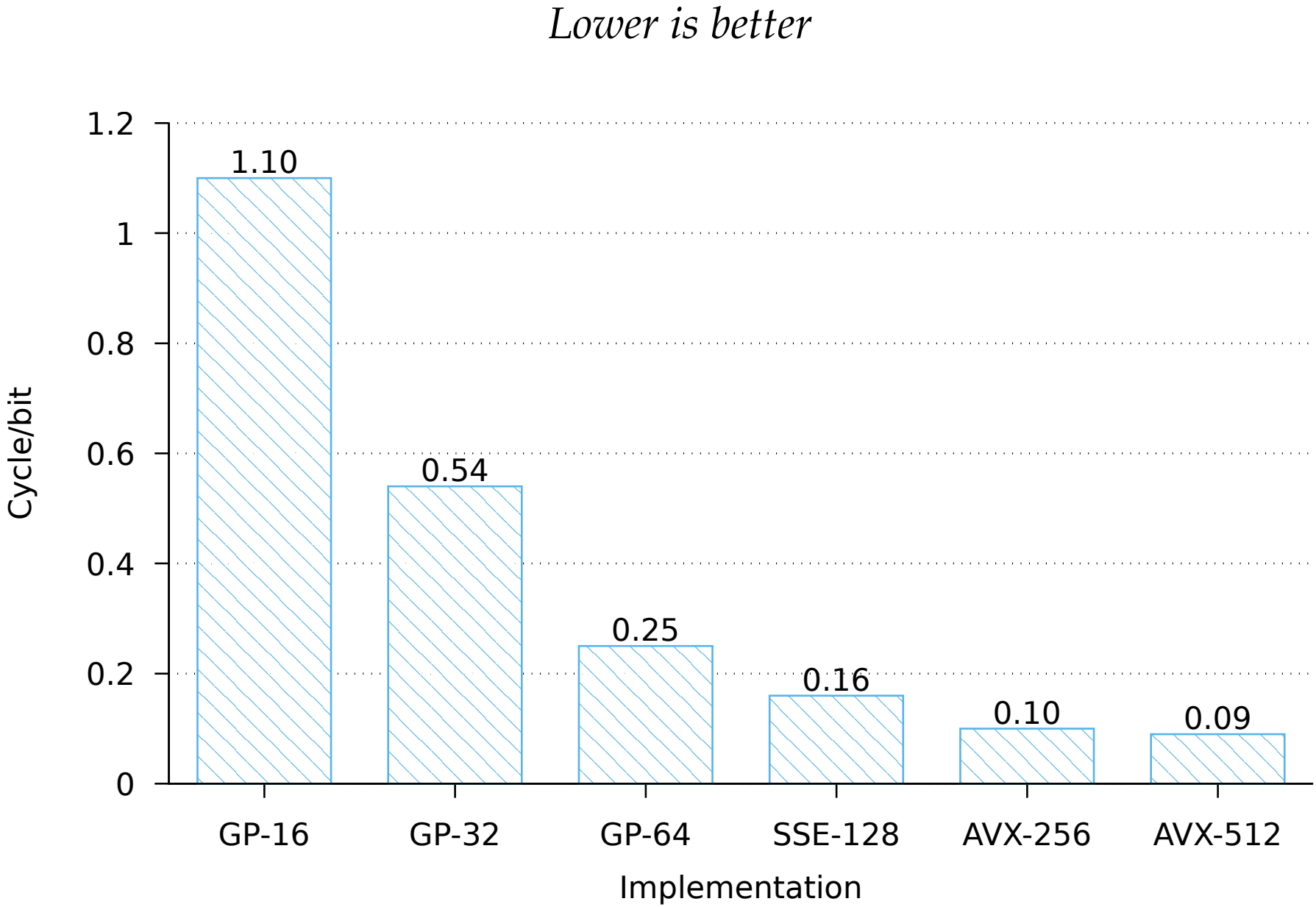
Intuition:

$$\text{Since } \begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix},$$

and A^T and B^T are stored in the same registers,
they can be computed at the same time.

$\mathcal{O}(n\ log(n))$ for n^2 bits

Transposition



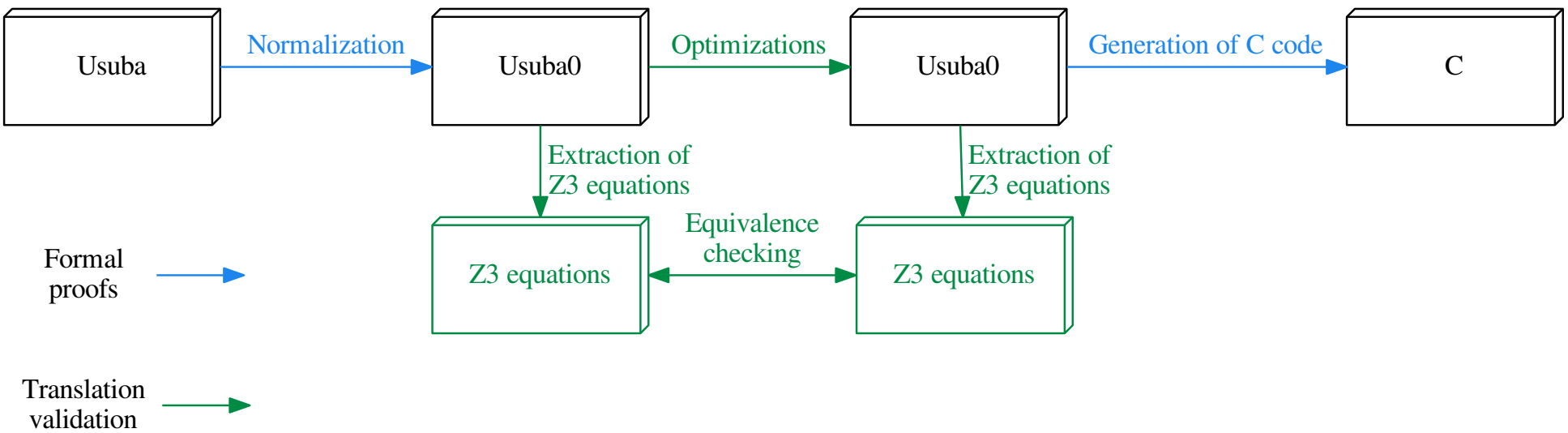
Transposition

```
void transpose(uint64_t data[64]) {
  for (int i = 0; i < 6; i ++) {
    int n = (1UL << i);
    for (int j = 0; j < 64; j += (2 * n))
      for (int k = 0; k < n; k ++) {
        uint64_t u = data[j + k] & mask_l[i];
        uint64_t v = data[j + k] & mask_r[i];
        uint64_t x = data[j + n + k] & mask_l[i];
        uint64_t y = data[j + n + k] & mask_r[i];
        data[j + k] = u | (x >> n);
        data[j + n + k] = (v << n) | y;
      }
  }
}
```

Straight forward adaptation to SIMD!

Verification

Verification



- Validate compilation
- Validate manual modifications

Verification

Validating manual modifications

```
table sbox_lookup (i:u4) returns (o:u4) {
  8, 6, 7, 9, 3, 12, 10, 15,
  13, 1, 14, 4, 0, 11, 5, 2
}
```

```
node sbox_circuit (a,b,c,d:bool) returns (w,x,y,z:bool)
vars t01,t02,t03,t04,t05,t06,t07,
     t08,t09,t10,t11,t12,t13,t14: bool
let
  t01 = a | c;          t02 = a ^ b;          t03 = d ^ t01;
  w = t02 ^ t03;        t05 = c ^ w;          t06 = b ^ t05;
  t07 = b | t05;        t08 = t01 & t06;      t09 = t03 ^ t07;
  t10 = t02 | t09;      x = t10 ^ t08;        t12 = a | d;
  t13 = t09 ^ x;        t14 = b ^ t13;        z = ~t09;
  y = t12 ^ t14;
tel
```

```
check_equal(sbox_lookup,sbox_circuit)
```