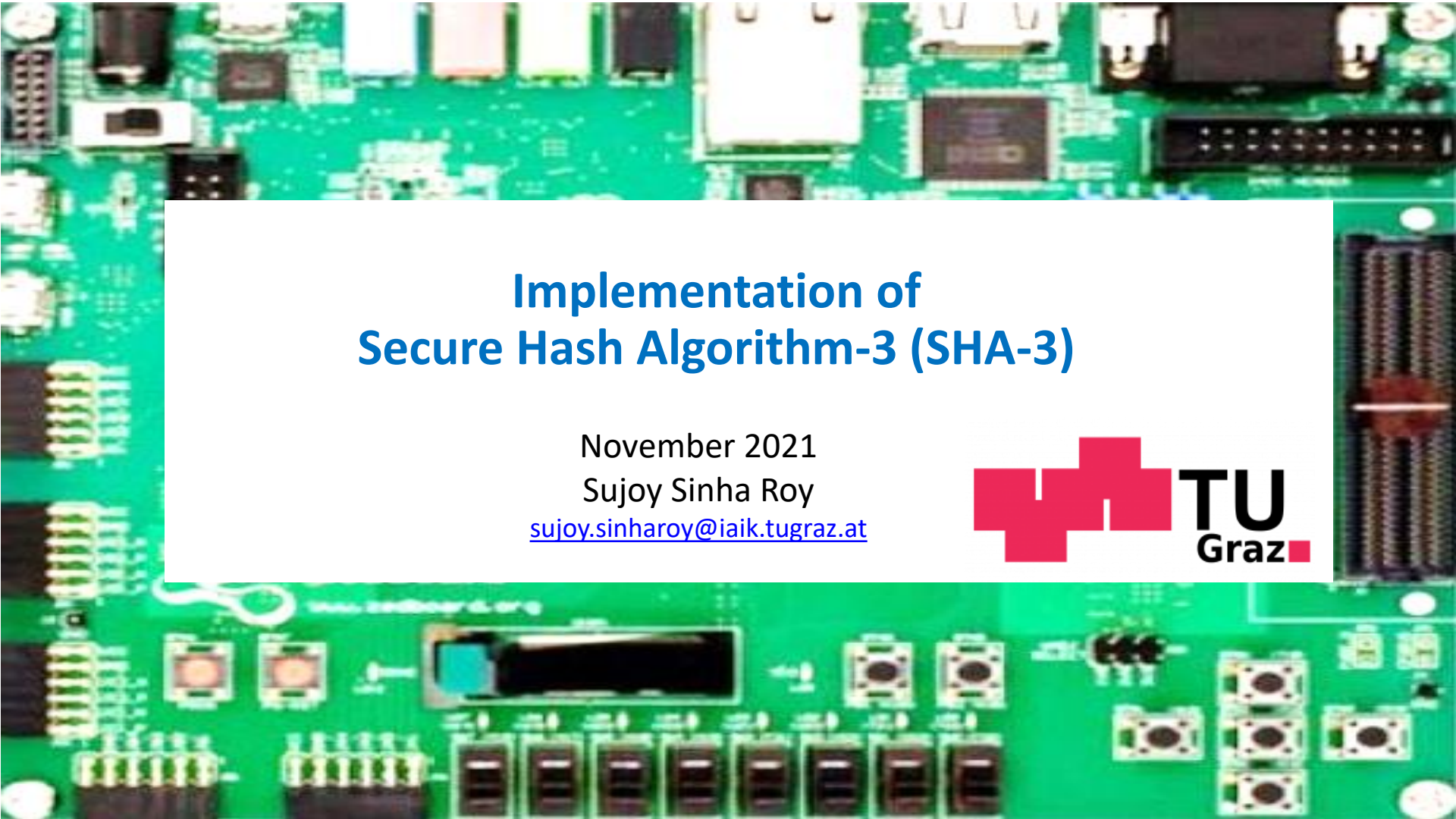


Implementation of Secure Hash Algorithm-3 (SHA-3)

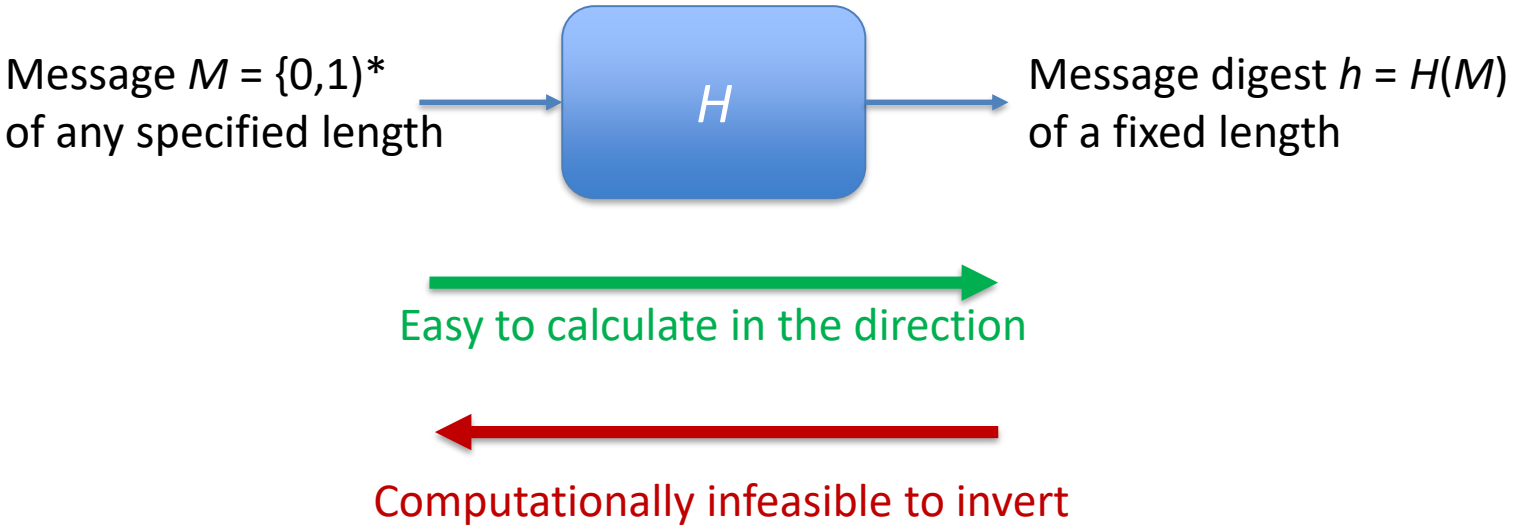
November 2021

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

TU Graz

Cryptographic hash functions

Hash function *H* maps a binary sequence of arbitrary length {0,1}* to a binary sequence of fixed length.



Applications of cryptographic hash functions

The main goal of hash function is to create a small fingerprint of a large input data.

Hash functions are used in cryptographic protocols, e.g.,

- Generation and verification of digital signatures
- Key derivation
- Pseudorandom bit generation
- Blockchains

Example: textbook RSA signature

Signature generation

Input: Private key d and message M

1. Compute the hash $h = H(M)$
2. Do RSA-encryption of h to calculate the signature $s = h^d \bmod N$

Signature verification

Input: Public key e, message M, and signature s

1. Compute the hash $h = H(M)$
2. Decrypt the received signature and obtain $h' = s^e \bmod N$
3. Check if $h' \stackrel{?}{=} h$.
   If equal, then the signature is valid.
   Otherwise not.

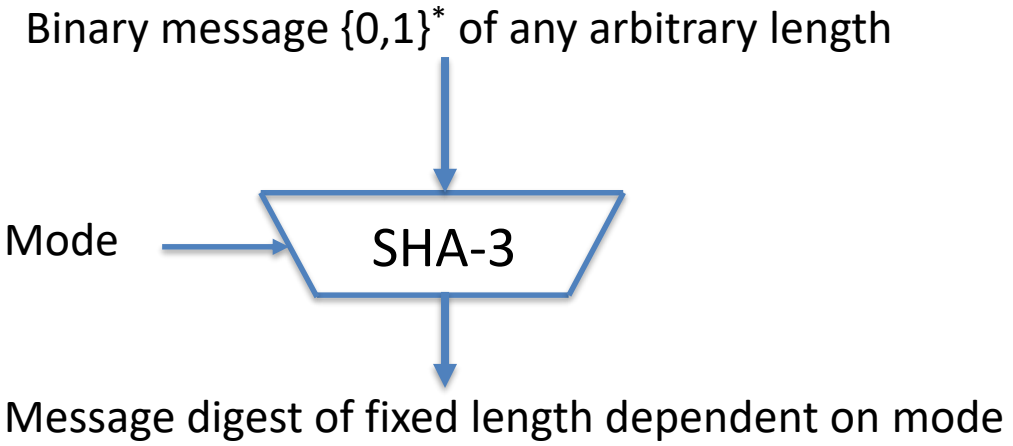
Popular hash functions

- MD5 : Newer version of MD4 proposed in 1991. Broken in 1995
- SHA-1 : Published in 1995 by NIST. Theoretical attack in 2005. Practically broken in 2017.
- SHA-2 : Proposed in 2001 and standardized by NIST. Not broken till date.
- SHA-3 : NIST called for proposals for SHA-3 in 2007. Following a four year competition, “Keccak” was selected as the SHA-3 algorithm.

This lecture:

- Not about analysing cryptographic properties of hash functions
- More about the implementation aspects of hash functions with a focus on SHA-3, i.e., Keccak.

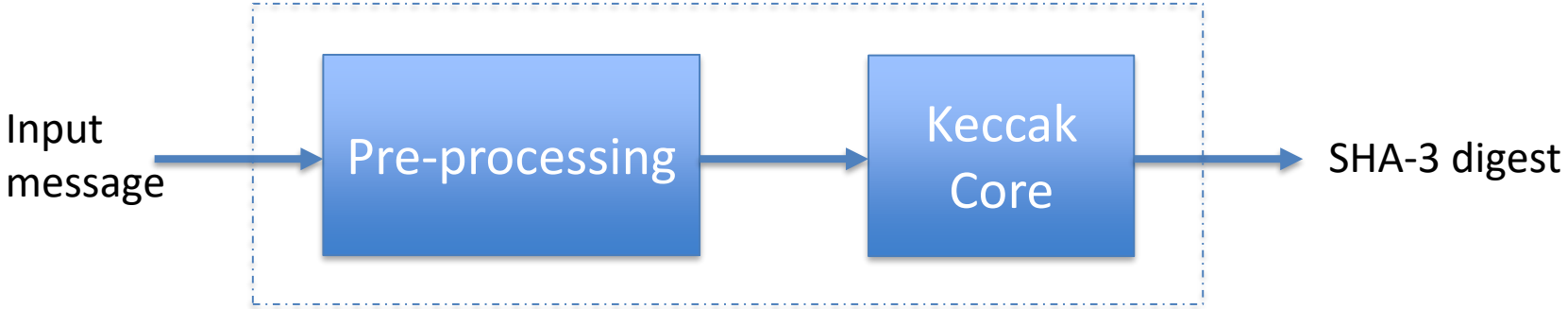
SHA-3 input/output specification



Mode	Output length (bits)	Attack resistance
SHA3-224	224	112 (≡ to 3DES)
SHA3-256	256	128 (≡ to AES128)
SHA3-384	384	192 (≡ to AES192)
SHA3-512	512	256 (≡ to AES256)

SHA-3 and Keccak

SHA-3 is Keccak-based.



High level diagram of SHA-3

Steps used:

1. A pre-processing is used to pad message and then split it into blocks.
2. These blocks are ‘absorbed’ by Keccak one-by-one
3. Finally, the message digest is produced by ‘squeezing’ Keccak.

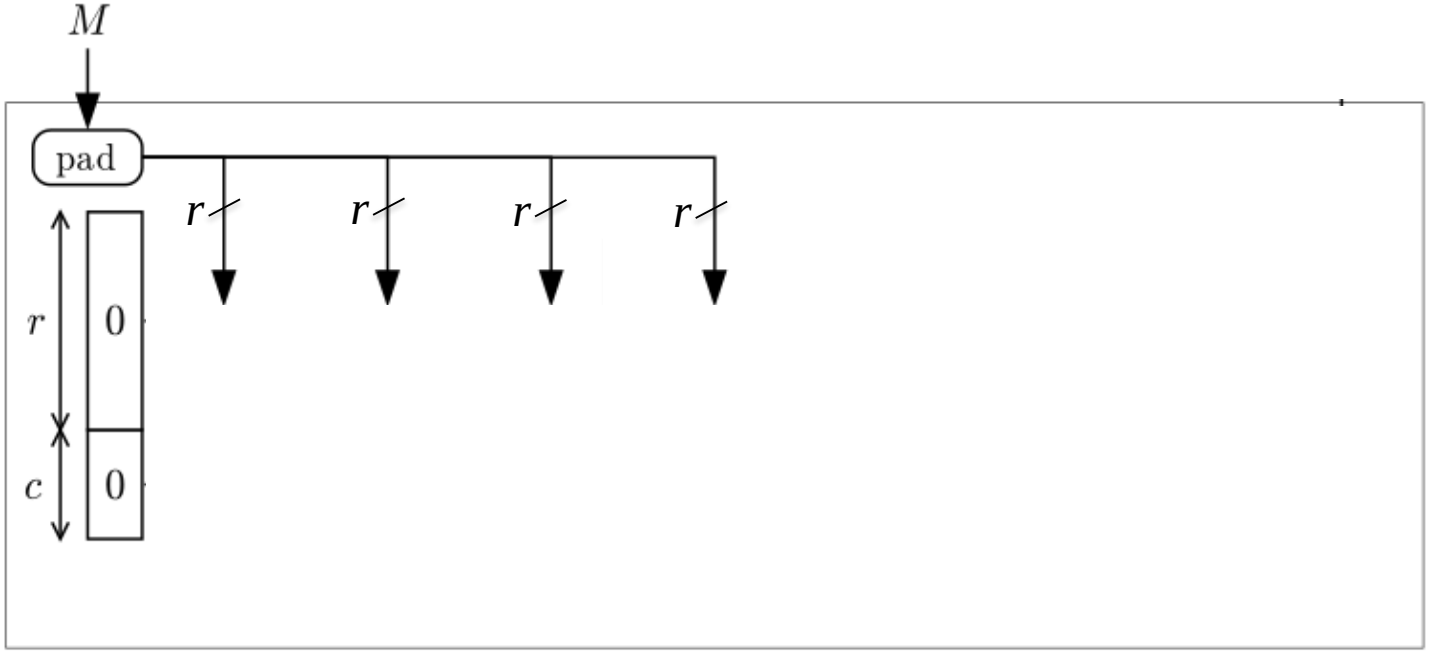
Like a sponge

Keccak sponge construction overview (1)

Preprocess:

1. The input string *M* is padded following a ‘rule’ and cut into blocks of r bits.

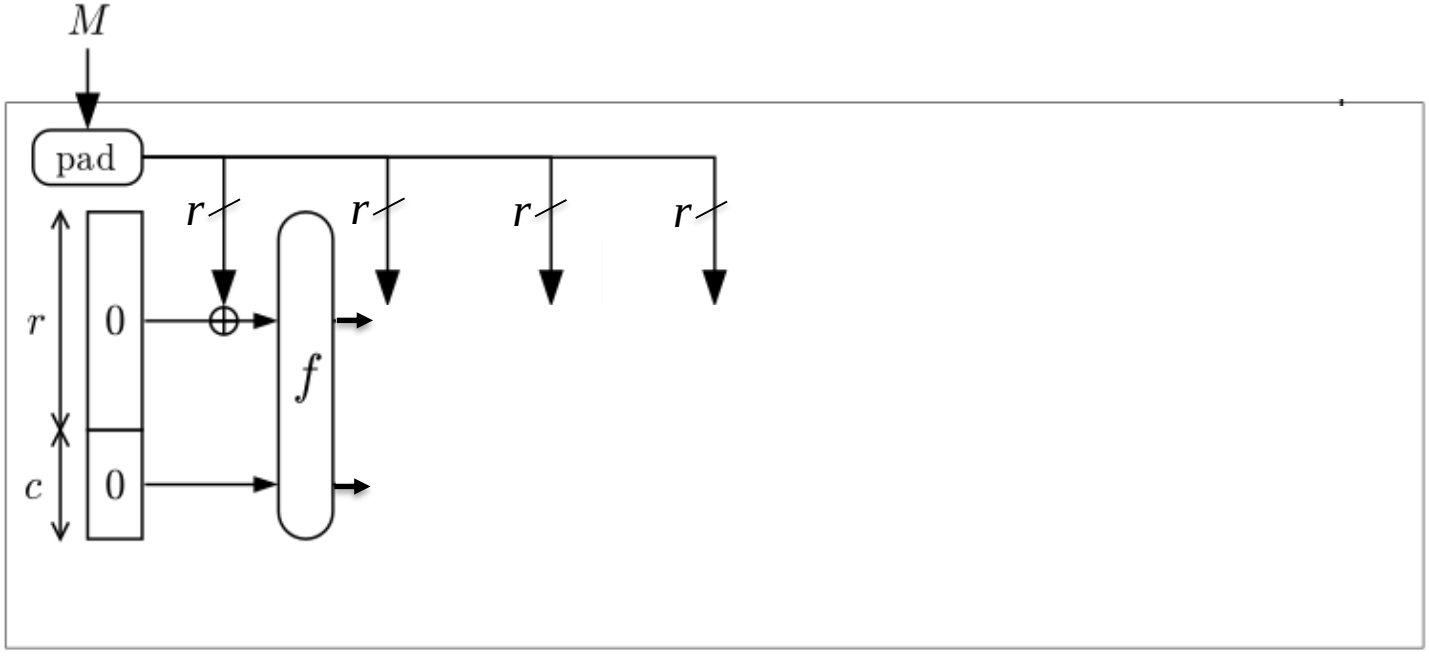
2. Next, the $b = r + c$ bit ‘state’ is initialized with all 0s.



Keccak sponge construction overview (2)

Absorb:

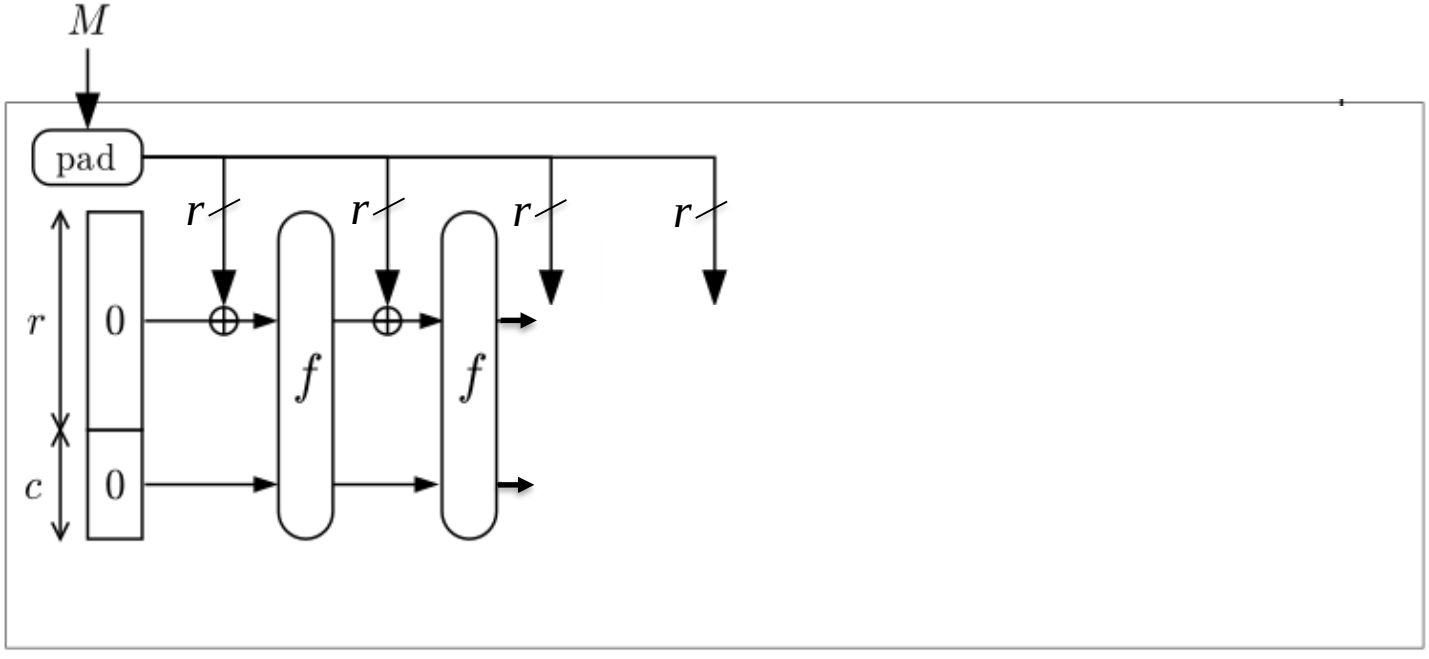
1. The 1st r-bit input block is XOR-ed into the first r bits of the state.
2. Next, state permutation function *f*() is applied to the *b*-bit state.



Keccak sponge construction overview (2)

Absorb:

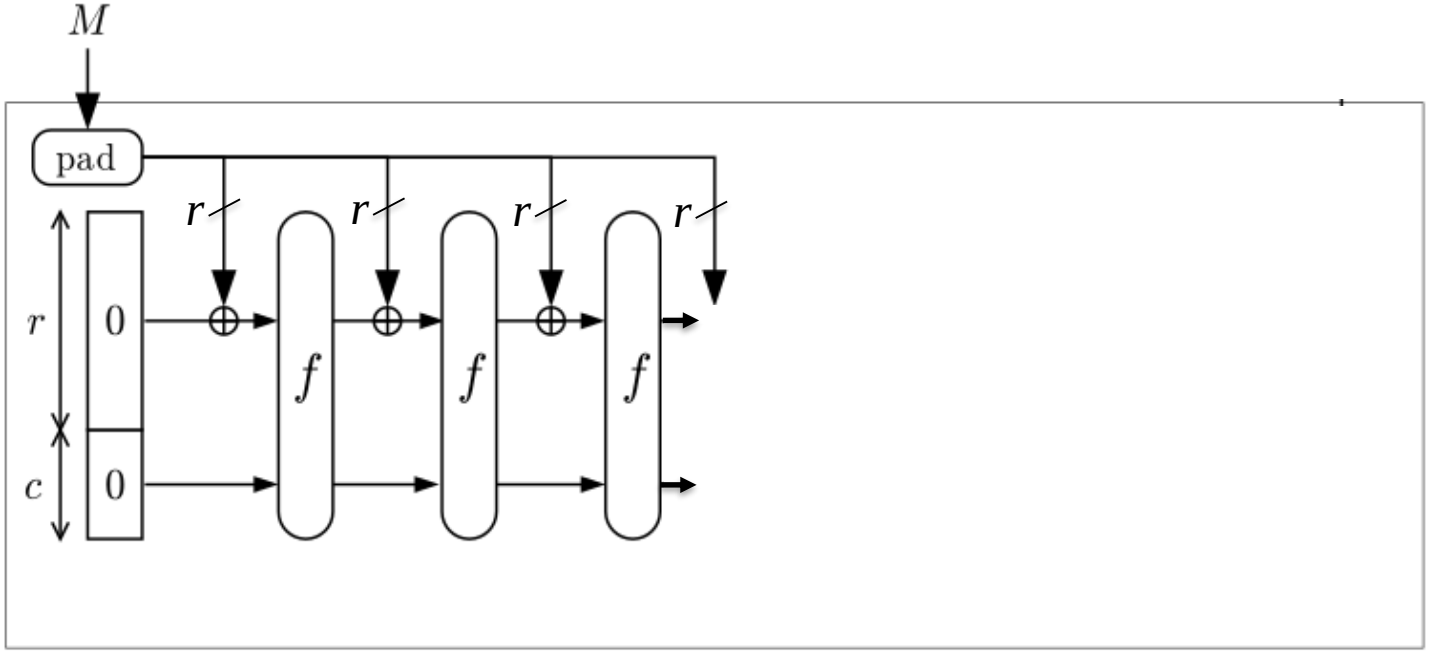
3. Follow the previous two steps for all the remaining r-bit blocks one-by-one.



Keccak sponge construction overview (2)

Absorb:

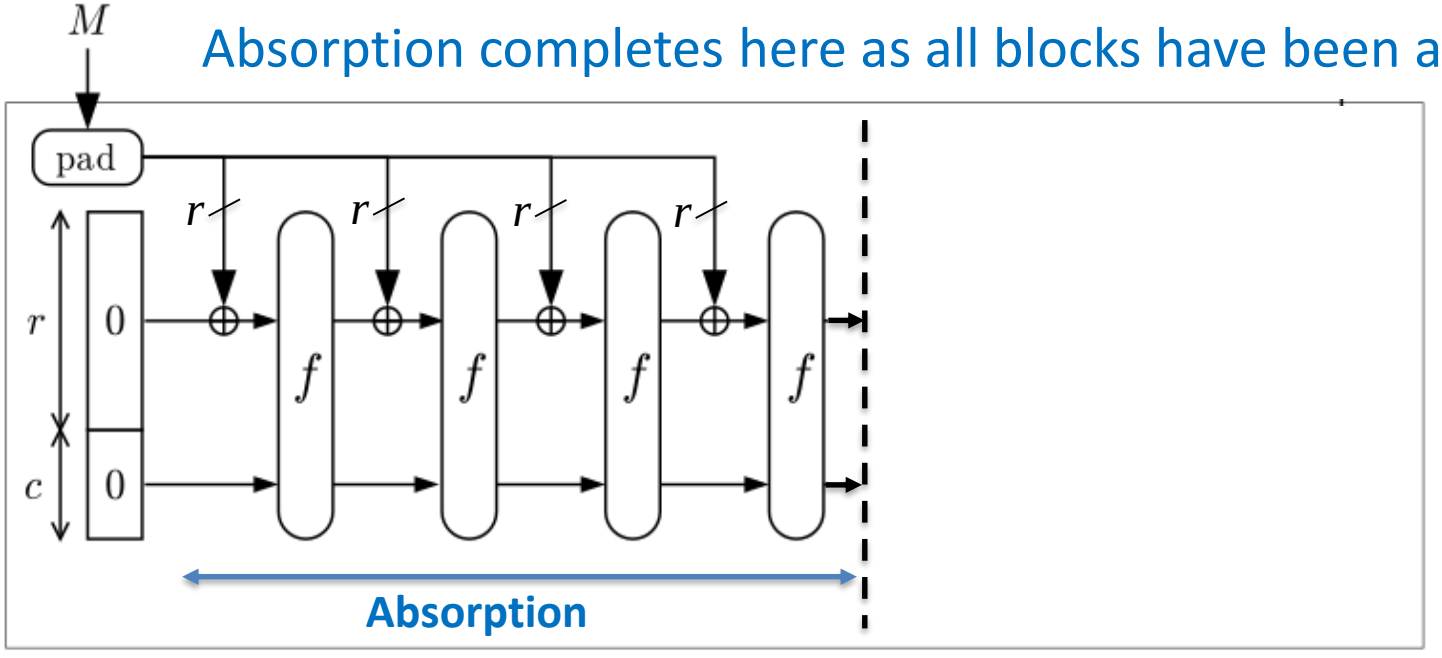
3. Follow the previous two steps for all the remaining r-bit blocks one-by-one.



Keccak sponge construction overview (2)

Absorb:

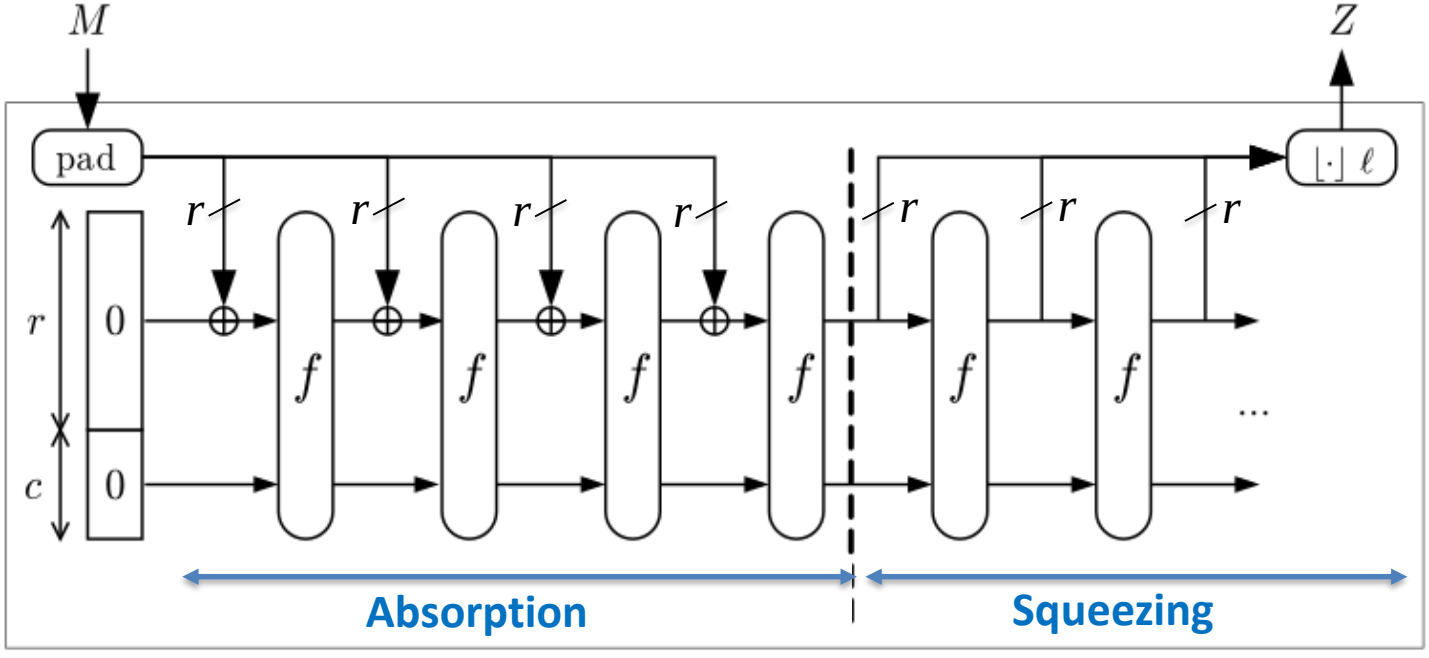
3. Follow the previous two steps for all the remaining r-bit blocks one-by-one.



Keccak sponge construction overview (2)

Squeeze:

1. First *r* bits of the state are output, interleaved with applications of the function *f*.
2. The number of output *r*-bit blocks is specified by the user or another algorithm.



Keccak parameter

The state size $b = 25 \cdot 2^l$ where $l = 0, 1, \ldots, 6$.

i.e., $b \in \{ 25, 50, 100, 200, 400, 800, 1600 \}$

The number of rounds $n_r = 12 + 2 \cdot l$

Keccak parameter

The state size $b = 25 \cdot 2^l$ where $l = 0, 1, \ldots, 6$.

i.e., $b \in \{ 25, 50, 100, 200, 400, 800, 1600 \}$

The number of rounds $n_r = 12 + 2 \cdot l$

SHA-3 standard uses Keccak with $l = 6$.
➔ $b = 1600$ and $n_r = 24$

SHA-3 parameter

Output length	State size *b*	Rate *r*	Capacity *c* = *b* - *r*
224	1600	1152	448
256	1600	1088	512
384	1600	833	767
512	1600	576	1024

Note: Rate *r* is the input/output block size.

For SHA3-*i* we take only the first *i* bits from the *r* bits.

Summary so far

- Keccak-based SHA-3 follows a sponge construction
  - Input data is pre-processed
  - Then absorbed into the state
  - The state is updated using a cryptographically secure mapping
  - Finally, the output bits are extracted from the state

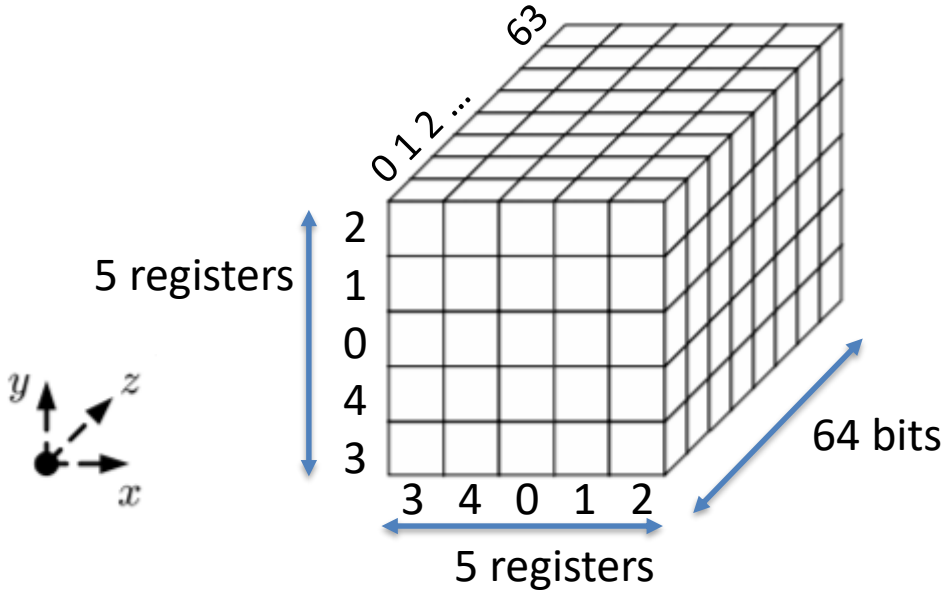
- Parameters of Keccak are discussed

Next:
How is the state of Keccak is represented?
How is the state updated using *f*()?

(From now on we will use the parameter for SHA-3, i.e., b = 1600, l = 6, and n_r = 24.)

Representation of Keccak state

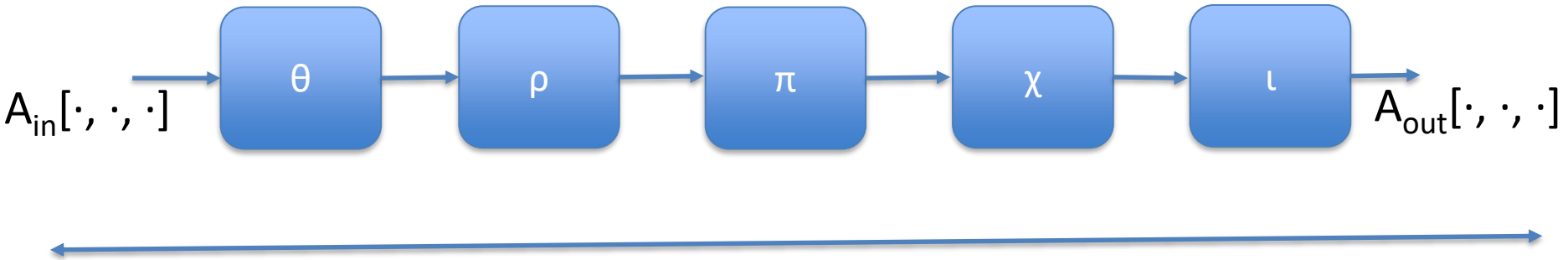
- The $b = 25 \cdot 2^l = 1600$ bit state of Keccak is represented as a 3D array



- Equivalent to 25 registers, each register is of size 64 bits.
- 3D-vs-1D interchange is as follows: A[x,y,z]=S[64·(5y+x)+z].

Step mapping

Each round of Keccak comprises of 5 steps: θ, ρ, π, χ, and ι



SHA-3 applies this round $n_r = 24$ times to its state

Pseudocode description of the steps is available at:
https://keccak.team/keccak_specs_summary.html

I will be explaining the steps using diagrams.

Specification of θ

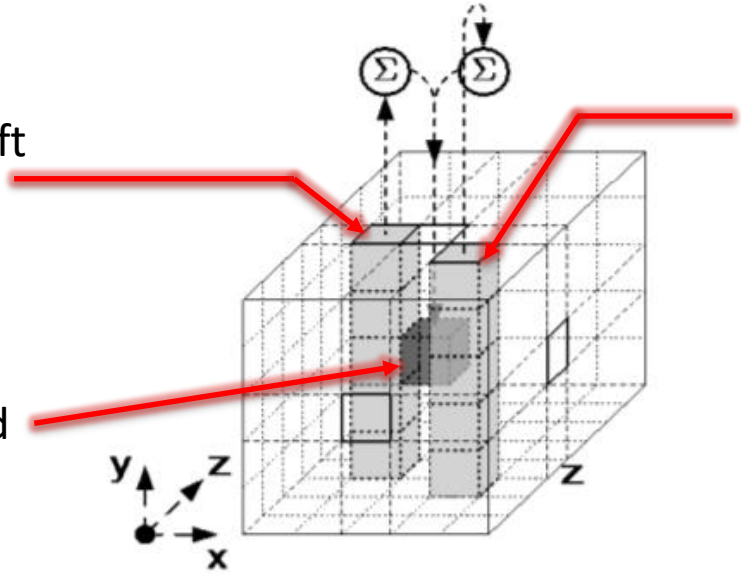
Each bit of the state is replaced by 11 bits in the following way:

$$A[x,y,z] = A[x,y,z] \oplus (\text{The column to its left}) \oplus (\text{The column to its right and one position in the front})$$

The column to the left

The column to the right and one position in the front

The bit to be updated

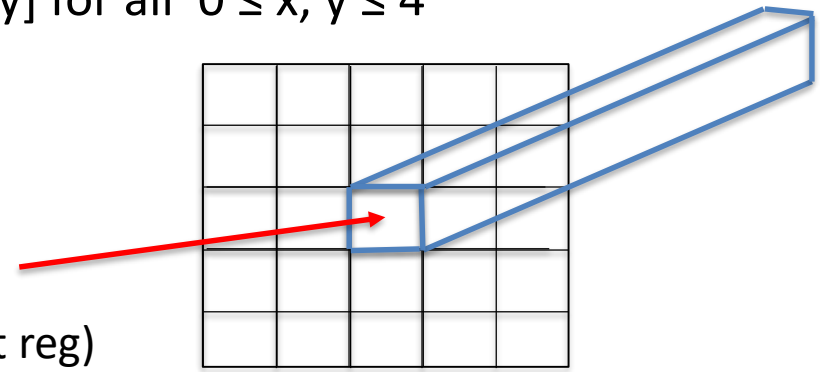


Note: All ‘Left’, ‘Right’, ‘Front’ are modulo 5.

Specification of ρ

Symbol ‘rho’ represents ‘rotation’. Cyclic rotation of the bits of each state-register.

Input: Register $A[x,y]$ for all $0 \le x, y \le 4$



$A[x,y]$

(this is a 64-bit reg)

Output: $B[x,y] = \text{rot}(A[x,y], r[x,y])$ where $r[x,y]$ is a rotation-constant from a table.
Here rot() is bitwise cyclic shift, moving bit at position i into position $i + r[x,y] \bmod 64$

Specification of ρ

Values of the rotation constants *r*[x, y]

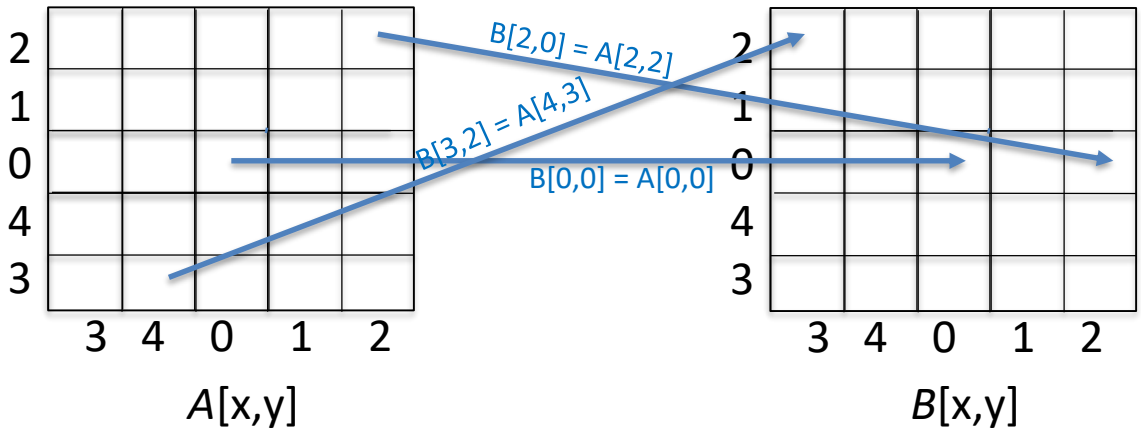
	x = 3	x = 4	x = 0	x = 1	x = 2
y = 2	25	39	3	10	43
y = 1	55	20	36	44	6
y = 0	28	27	0	1	62
y = 4	56	14	18	2	61
y = 3	21	8	41	45	15

Source: https://keccak.team/keccak_specs_summary.html#rotationOffsets

Specification of π

Symbol ‘pi’ represents ‘permutation’. Permutation of the state-registers.

Input: Register A[x, y] for all $0 \le x, y \le 4$



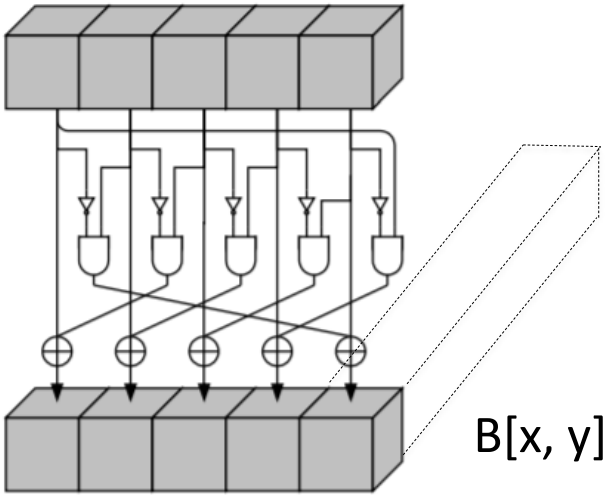
Permutation of 64-bit registers. (E.g., only a few of them are shown using arrows.)

Output: Register B[y, 2*x + 3*y mod 5] = A[x, y]

Specification of χ

This is the only layer where non-linear computation (bit-AND) is performed.

Input: Register A[x, y] for all $0 \le x, y \le 4$



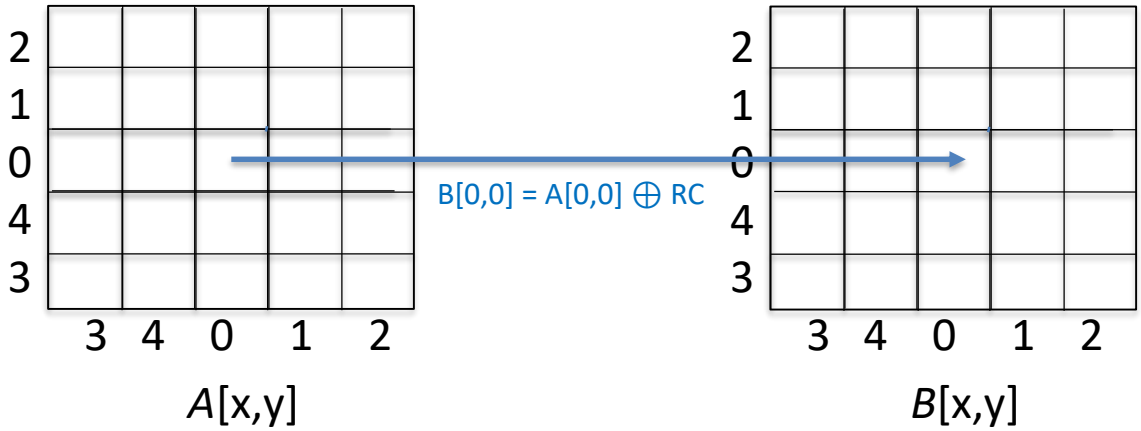
$$B[x, y] = A[x, y] \oplus (\overline{A}[x+1 \bmod 5, y] \,\&\, A[x+2 \bmod 5, y])$$

Output: Register B[x, y] for all $0 \le x, y \le 4$

Specification of ι

This step is applied *only to* the register A[0, 0].

Input: Register A[0, 0]



where RC is a round constant (dependent on round number) from a table.

Output: Register B[0, 0]

Specification of ι

Values of the round constants. Note that there are 24 rounds in SHA-3.

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

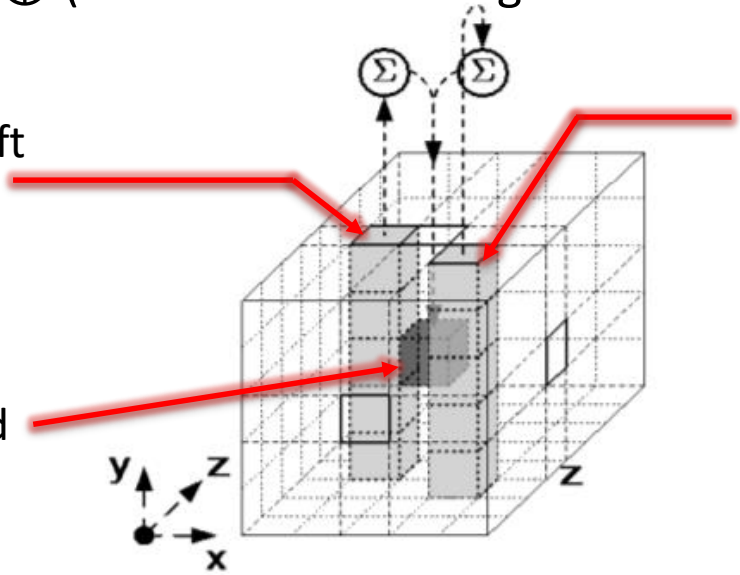
Source https://keccak.team/keccak_specs_summary.html#rotationOffsets

Implementation of Keccak

Implementation of θ

Each bit of the state is replaced by 11 bits in the following way:

$$A[x,y,z] = A[x,y,z] \oplus (\text{The column to its left}) \oplus (\text{The column to its right and one position in the front})$$



The column to the left

The bit to be updated

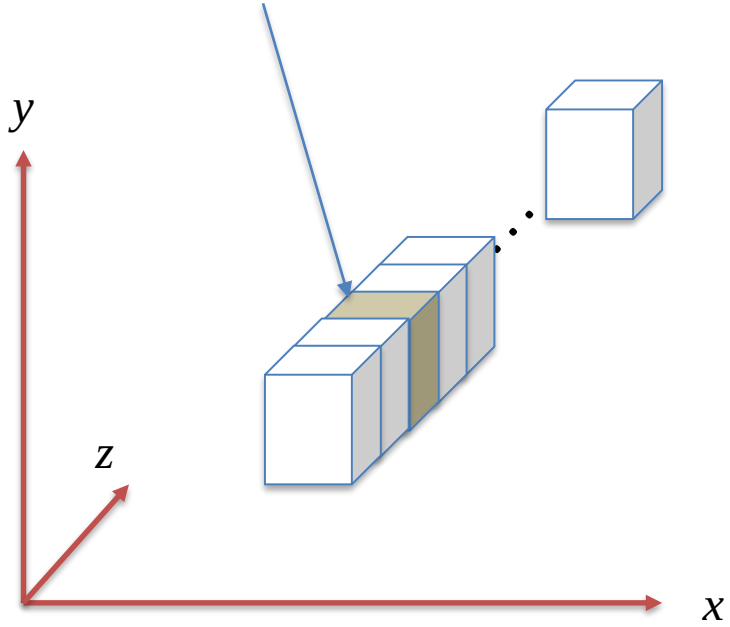
The column to the right and one position in the front

Note: All ‘Left’, ‘Right’, ‘Front’ are modulo 5.

Question: How to implement this step efficiently on a 64-bit computer?

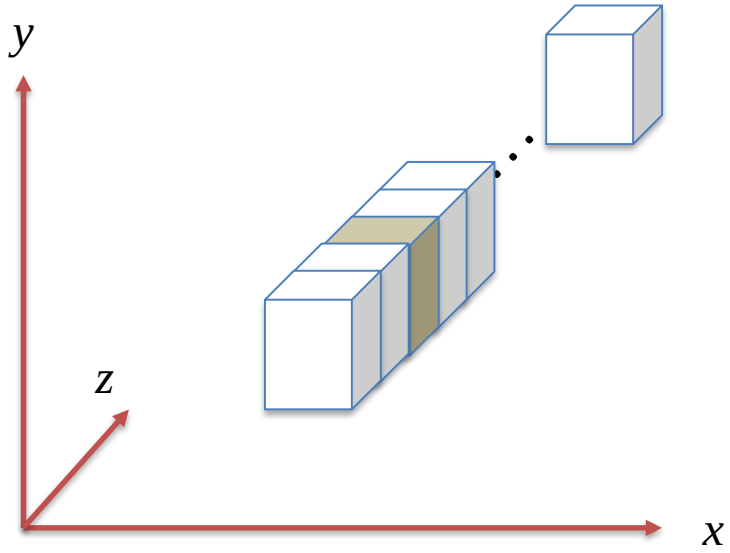
Implementation of θ

Let's update the bit $A[x_1, y_1, z_1]$



Implementation of θ

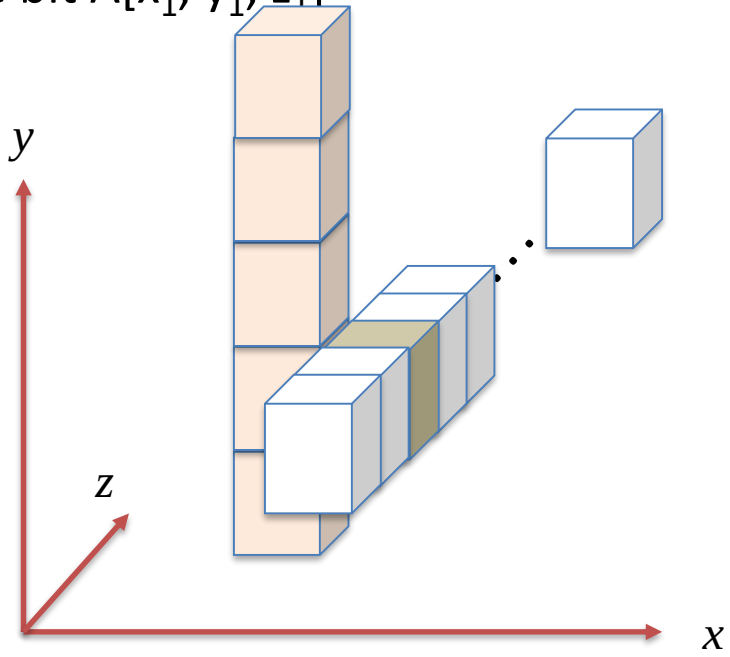
Let's update the bit $A[x_1, y_1, z_1]$



Output bit $B[x_1, y_1, z_1] = A[x_1, y_1, z_1] \oplus$

Implementation of θ

Let's update the bit $A[x_1, y_1, z_1]$

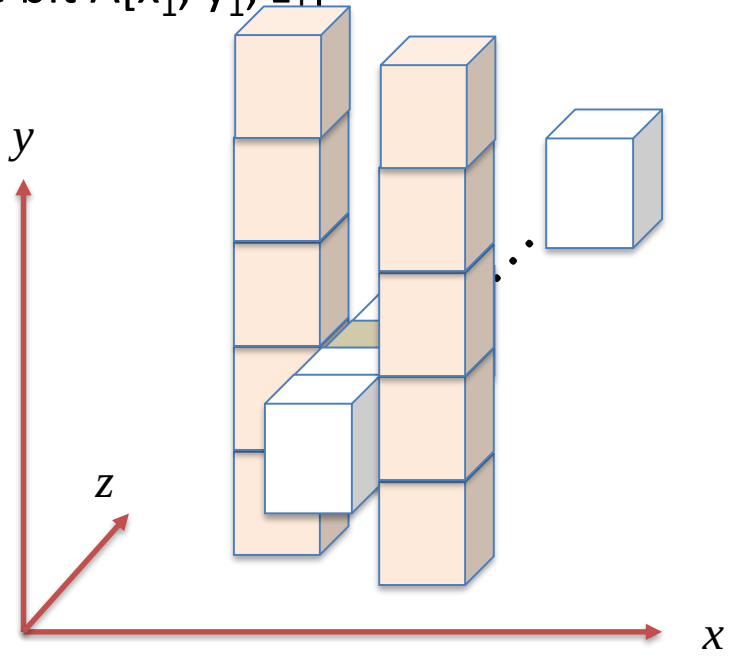


Output bit $B[x_1, y_1, z_1] = A[x_1, y_1, z_1] \oplus$ (All bits of column $A[x_1, 0{:}4, z_1]$)

$\oplus$

Implementation of θ

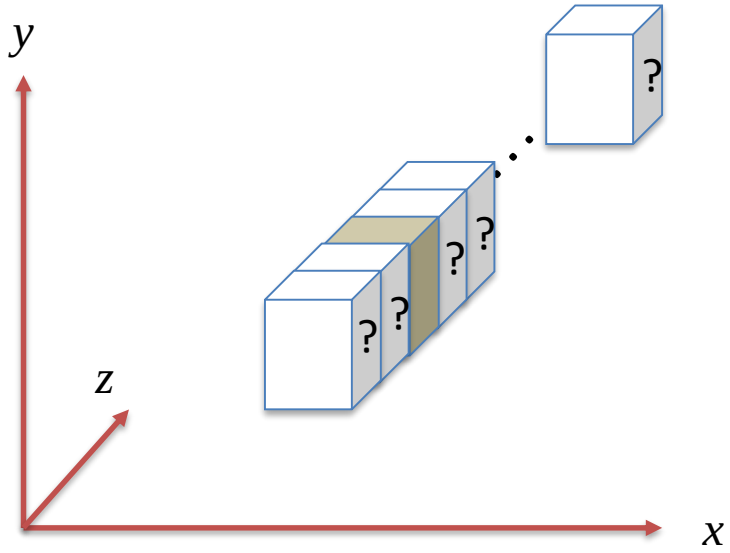
Let's update the bit $A[x_1, y_1, z_1]$



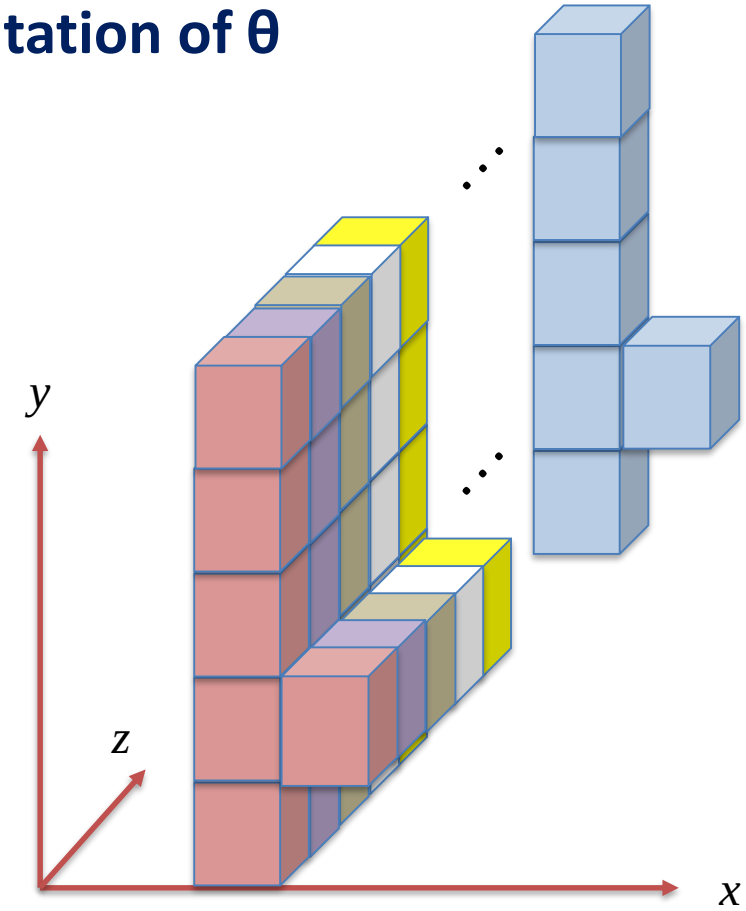
Output bit $B[x_1, y_1, z_1] = A[x_1, y_1, z_1] \oplus$ (All bits of column $A[x_1\text{-}1, 0{:}4, z_1]$)
$\oplus$ (All bits of column $A[x_1\text{+}1, 0{:}4, z_1 - 1]$)

Implementation of θ

What happens to the remaining bits in the same lane?

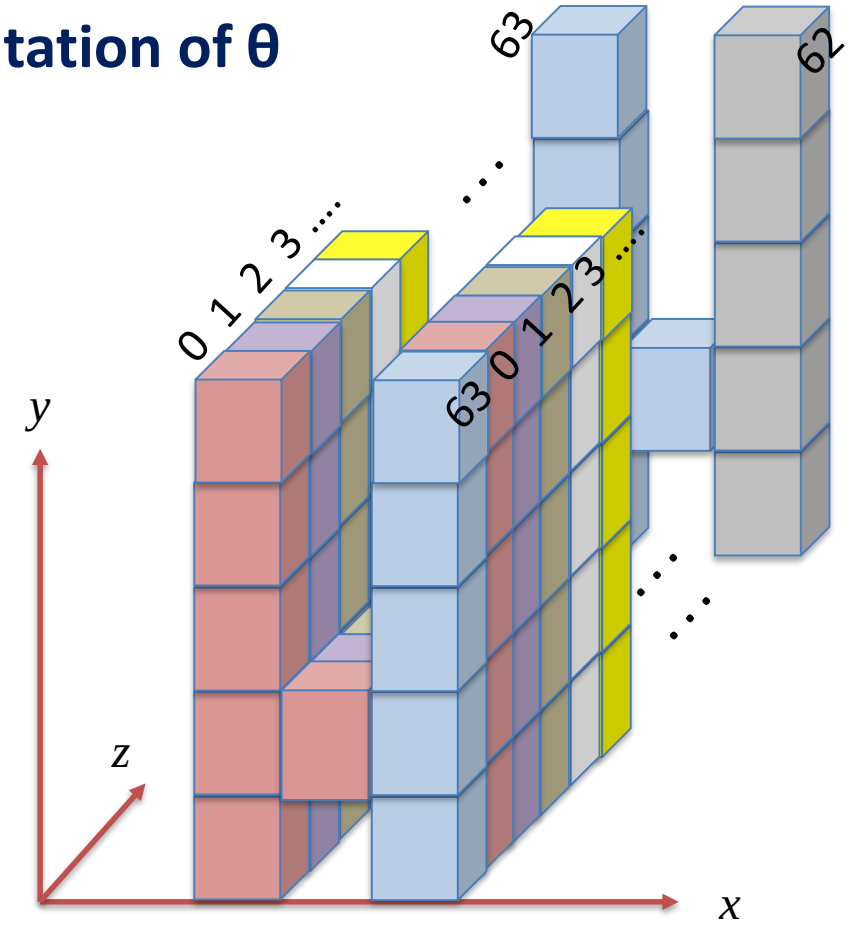


Implementation of θ



The columns that are XOR-ed slide along the z-axis.

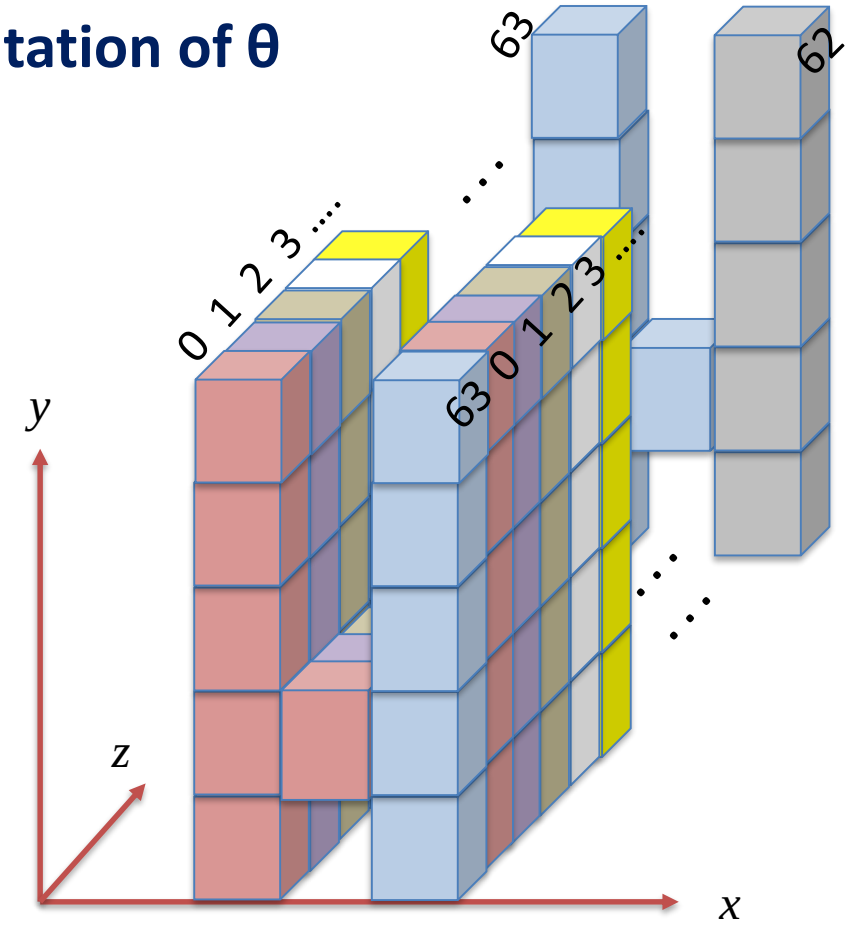
Implementation of θ



Note: The right ‘sheet’ has 1 position cyclic rotation.
This is because we choose ‘front’ columns (modulo 64)

The columns that are XOR-ed slide along the z-axis.

Implementation of θ



Efficient implementation on 64-bit platform:

We do word-level operations on the state registers instead of bit-level operations

$$B[x, y] = A[x, y] \oplus A[x-1, 0] \oplus A[x-1, 1] \oplus \dots \oplus A[x-1, 4] \oplus \mathrm{Rot}(A[x+1, 0],1) \oplus \mathrm{Rot}(A[x+1, 1],1) \dots \oplus \mathrm{Rot}(A[x+1, 4],1)$$

Word operations

Implementation of ρ, π, χ and ι

Similarly,

ρ is cyclic rotation of 64-bit words.

π is permutation of the 64-bit words.

χ is also operation involving 64-bit words.

ι is simple XOR of a 64-bit word by a constant.

```
Keccak-f[b](A) {
  for i in 0…n-1
    A = Round[b](A, RC[i])
  return A
}

Round[b](A,RC) {
  # θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   for x in 0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                            for x in 0…4
  A[x,y] = A[x,y] xor D[x],                           for (x,y) in (0…4,0…4)

  # ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                 for (x,y) in (0…4,0…4)

  # χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),  for (x,y) in (0…4,0…4)

  # ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

https://keccak.team/keccak_specs_summary.html

Implementation summary

- All computations are performed on the 64-bit state registers
  - Bitwise XOR, NOT, AND
  - Cyclic rotations of bits within a word
  - Permutation of words

- Hence, these operations are friendly to 64-bit platforms

- As there are 24 rounds, we apply these transformations 24 times

On **hardware platforms**, we can do design space exploration for various area-performance trade offs.

Hardware implementation for high speed

E.g., fully unroll the steps and design a bit-parallel round() that takes 1 cycle.

Or you can do partial unrolling to aim a balanced implementation.

```
Round[b](A,RC) {
  # θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   for x in 0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                            for x in 0…4
  A[x,y] = A[x,y] xor D[x],                          for (x,y) in (0…4,0…4)

  # ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                for (x,y) in (0…4,0…4)

  # χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),  for (x,y) in (0…4,0…4)

  # ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

References

https://keccak.team/keccak_specs_summary.html