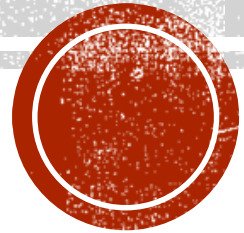
УЧЕБНАЯ ПРАКТИКА



Занятие 2

20 декабря 2018 г.

ХЕШ-ФУНКЦИИ

Опр. 1: **Хеш-функция** – это функция, которая детерминированным образом переводит строку бит произвольной длины в строку бит фиксированной длины.

- Не имеют секретного ключа
- Работают быстро
- Обладают «лавинным эффектом»
- Имеют большое практическое применение как вспомогательные функции

ПРИМЕР РАБОТЫ ХЕШ-ФУНКЦИИ

SHA1 hash for "Hello, World!"

Algorithm

SHA1

String to encode

Hello, World!

SHA1 encoded string

```
0a0a9f2a6772942557ab5355d76af442f8f65e01
```

SHA1 hash for "Hello, world!"

Algorithm

SHA1

String to encode

Hello, world!

SHA1 encoded string

```
943a702d06f34599aee1f8da8ef9f7296031d699
```

Ресурс: md5calc.com

ПРИМЕРЫ ХЕШ-ФУНКЦИЙ

Algorithm		Output [bit]	Input [bit]	No. of rounds	Collisions found
MD5		128	512	64	yes
SHA-1		160	512	80	not yet
SHA-2	**SHA-224**	224	512	64	no
	SHA-256	256	512	64	no
	SHA-384	384	1024	80	no
	SHA-512	512	1024	80	no

SHA-1 (SECURE HASH ALGORITHM)

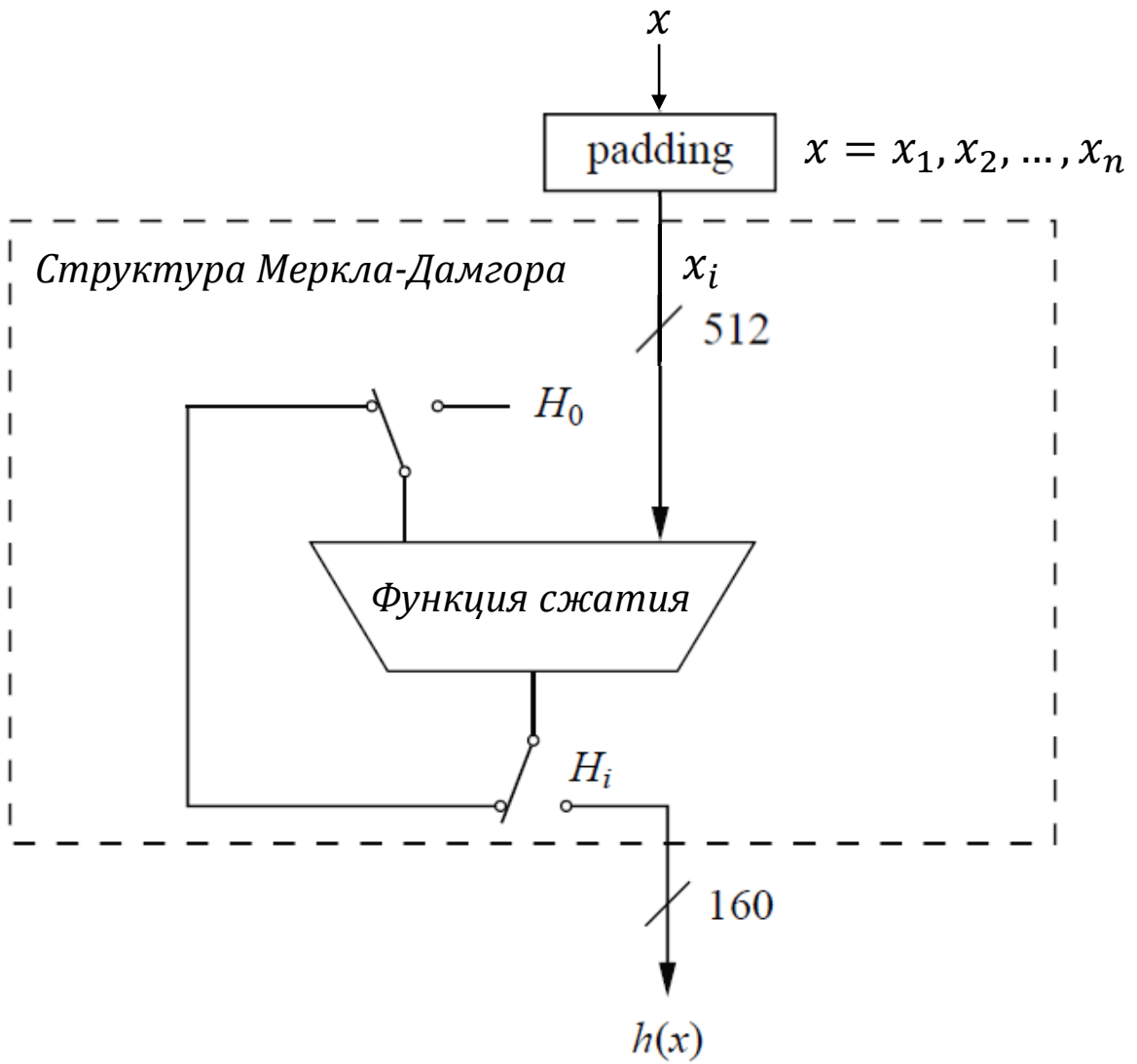
Основан на структуре Меркла-Дамгора (**Merkle–Damgård**)

Сообщение разбивается на блоки $x_1, x_2, \ldots, x_n$ и для каждого блока вычисляется:

$$H_i = f(H_{i-1}, x_i)$$

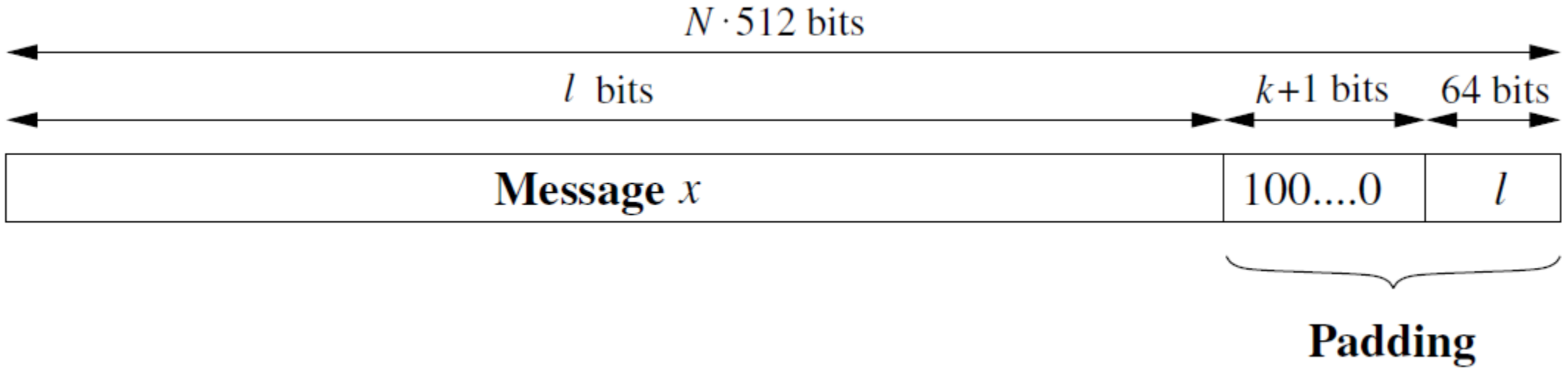
H_n и будет результатом хеширования.

H_0 – стандартная константа инициализации алгоритма.



ШАГ 1. PADDING

Используется для приведения размера сообщения к кратному 512 и для обеспечения криптостойкости.



Бит ‘1’ и длина исходного сообщения в битах (l) добавляются **всегда** – даже если длина исходного сообщения уже кратна 512. Нулей же может и не быть.

Максимальный размер сообщения, хеш от которого может найти **SHA-1**: 2^{64} бит ($\approx 2 \cdot 10^{18}$ байт $\approx$ 2 эксабайт).

Шаг 1. **padding** (пример)

Пусть **x** = “**abc**”

$$\underbrace{01100001}_{\texttt{a}}\quad\underbrace{01100010}_{\texttt{b}}\quad\underbrace{01100011}_{\texttt{c}}$$

$$\underbrace{01100001}_{\texttt{a}}\quad\underbrace{01100010}_{\texttt{b}}\quad\underbrace{01100011}_{\texttt{c}}\quad 1\quad\underbrace{00\ldots0}_{423\text{ zeros}}\quad\underbrace{00\ldots011000}_{l=24}$$

ШАГ 2. ДЕЛЕНИЕ НА БЛОКИ

Делим сообщение на блоки по 512 бит, каждый из которых делим на 16 32-битных слов.

$$x_i = \left(x_i^{(0)}\ x_i^{(1)}\ \dots\ x_i^{(15)}\right)$$

ШАГ 3. ИНИЦИАЛИЗАЦИЯ

H_0 – стандартная константа инициализации алгоритма:

$$A = H_0^{(0)} = \texttt{67452301}$$

$$B = H_0^{(1)} = \texttt{EFCDAB89}$$

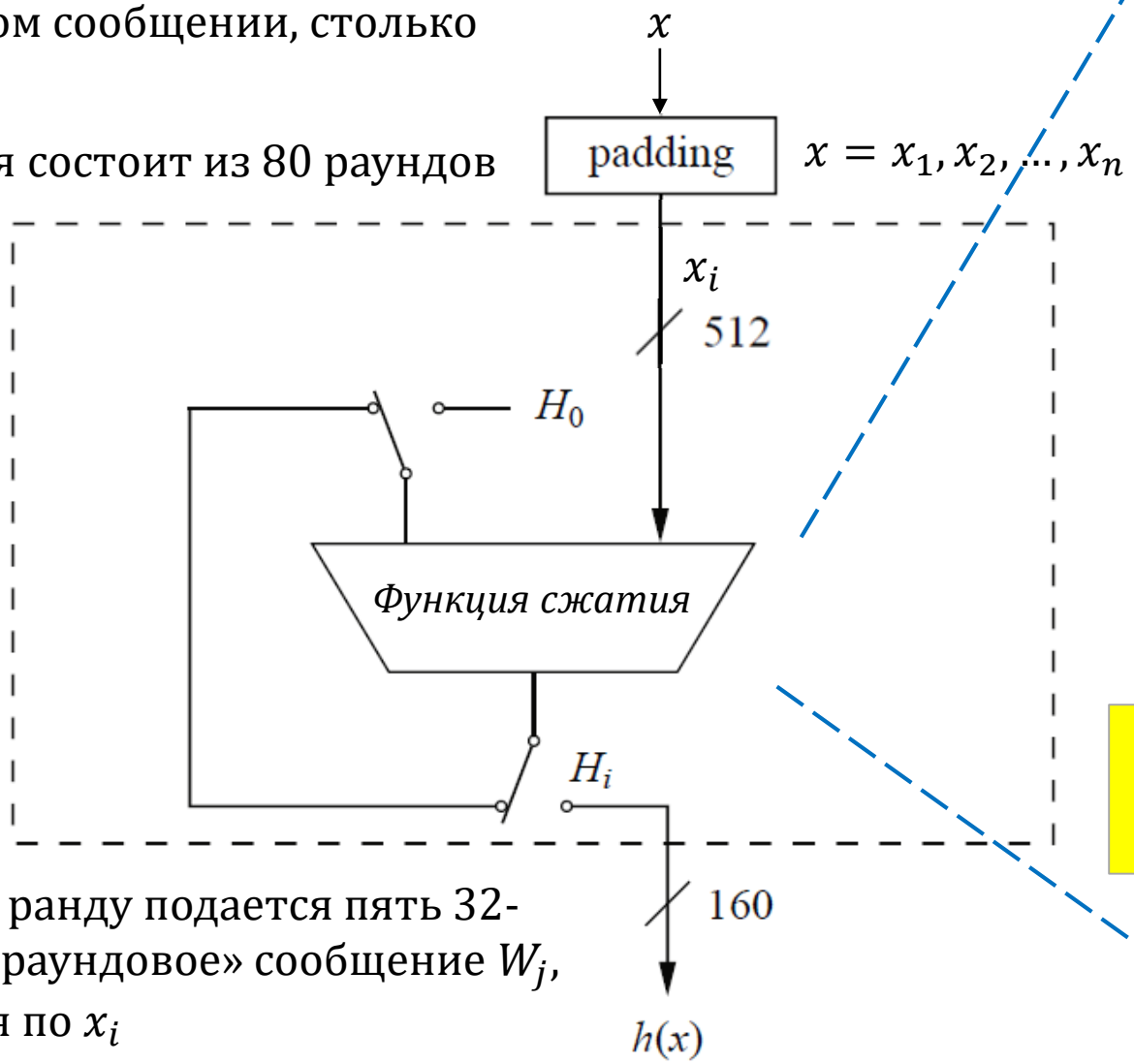
$$C = H_0^{(2)} = \texttt{98BADCFE}$$

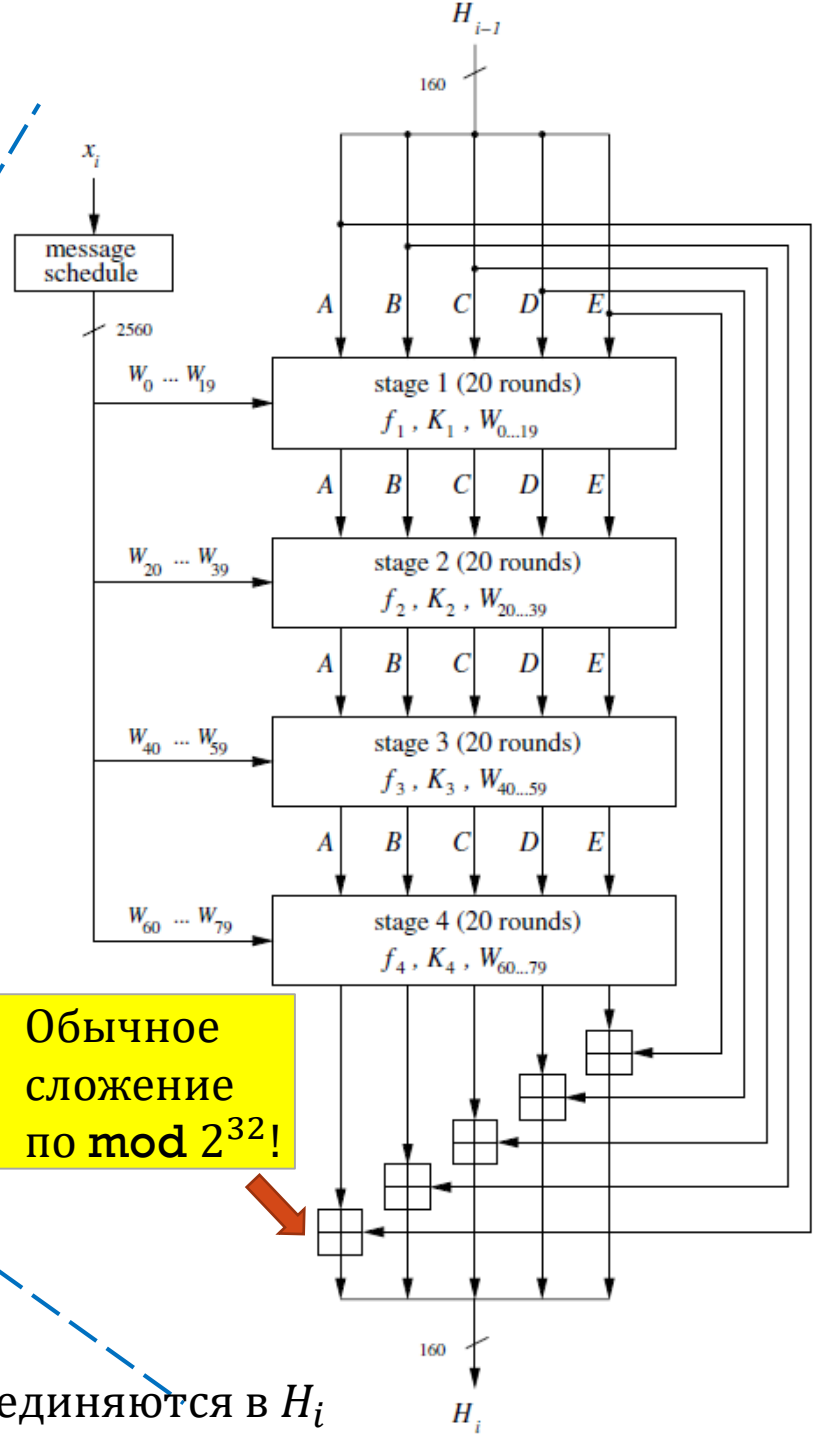
$$D = H_0^{(3)} = \texttt{10325476}$$

$$E = H_0^{(4)} = \texttt{C3D2E1F0}$$

ШАГ 4. ФУНКЦИЯ СЖАТИЯ

- Выполняется итерационно: сколько блоков в исходном сообщении, столько итераций
- Каждая итерация состоит из 80 раундов
- Все раунды разбиты на 4 группы по 20, различающиеся функцией раунда f_t и константой раунда K_t
- Внутри группы все раунды идентичны
- На вход каждому ранду подается пять 32-битных слова и «раундовое» сообщение W_j, которое строится по x_i
- После 80-го раунда все слова **A,B,C,D,E** суммируются с исходными и объединяются в H_i





Обычное сложение по **mod** 2^{32}!

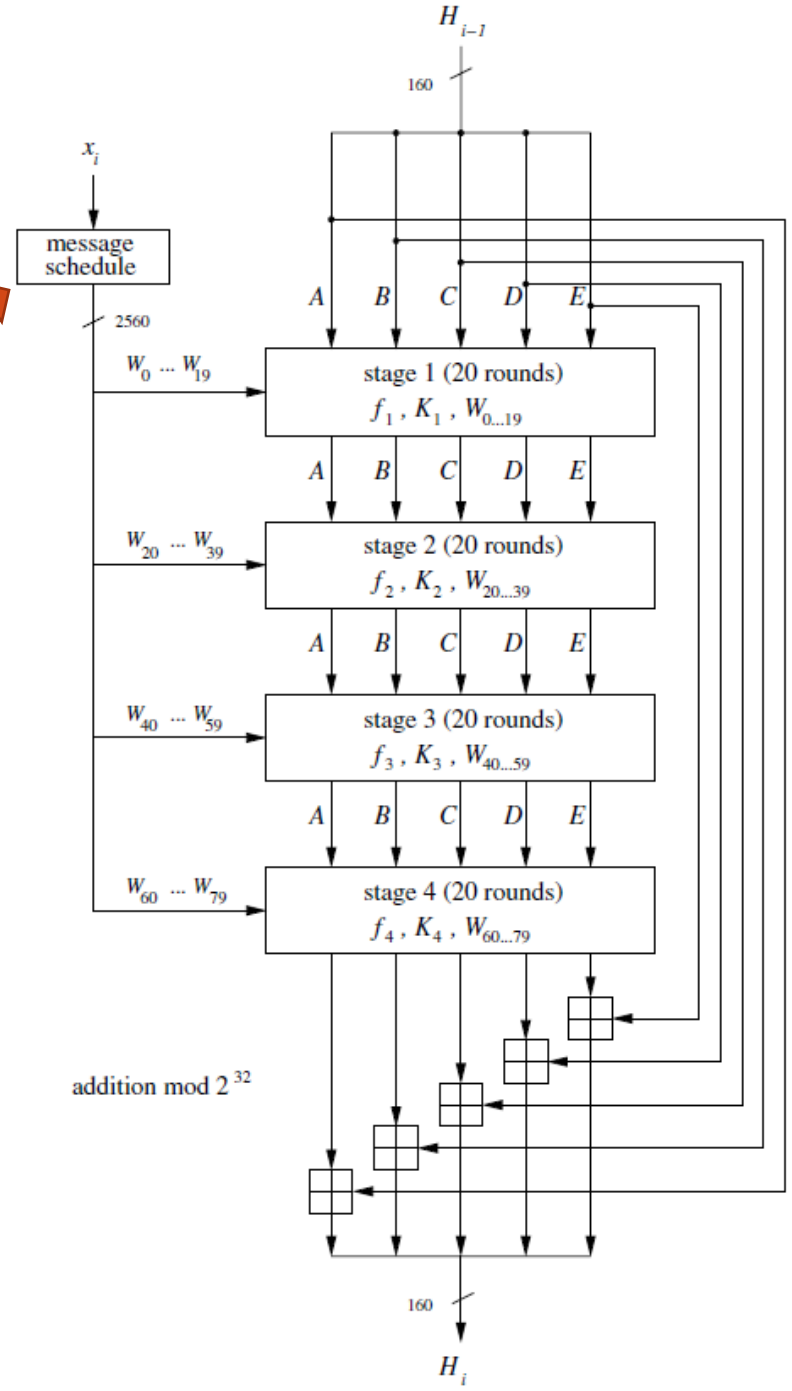
ШАГ 5. ВЫЧИСЛЕНИЕ W_j ПО СООБЩЕНИЮ

$$W_j = \begin{cases} x_i^{(j)} & 0 \le j \le 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) \lll 1 & 16 \le j \le 79, \end{cases}$$

- Все W_j строятся по 512-битному блоку x_i исходного сообщения, выдаваемого функцией **padding**
- Первые 16 значений W_j равны 16 словам 512-битного блока x_i исходного сообщения
- Следующие 64 вычисляются по рекуррентной формуле по четырем предыдущим значениям. Не забываем про циклический сдвиг на 1 бит!

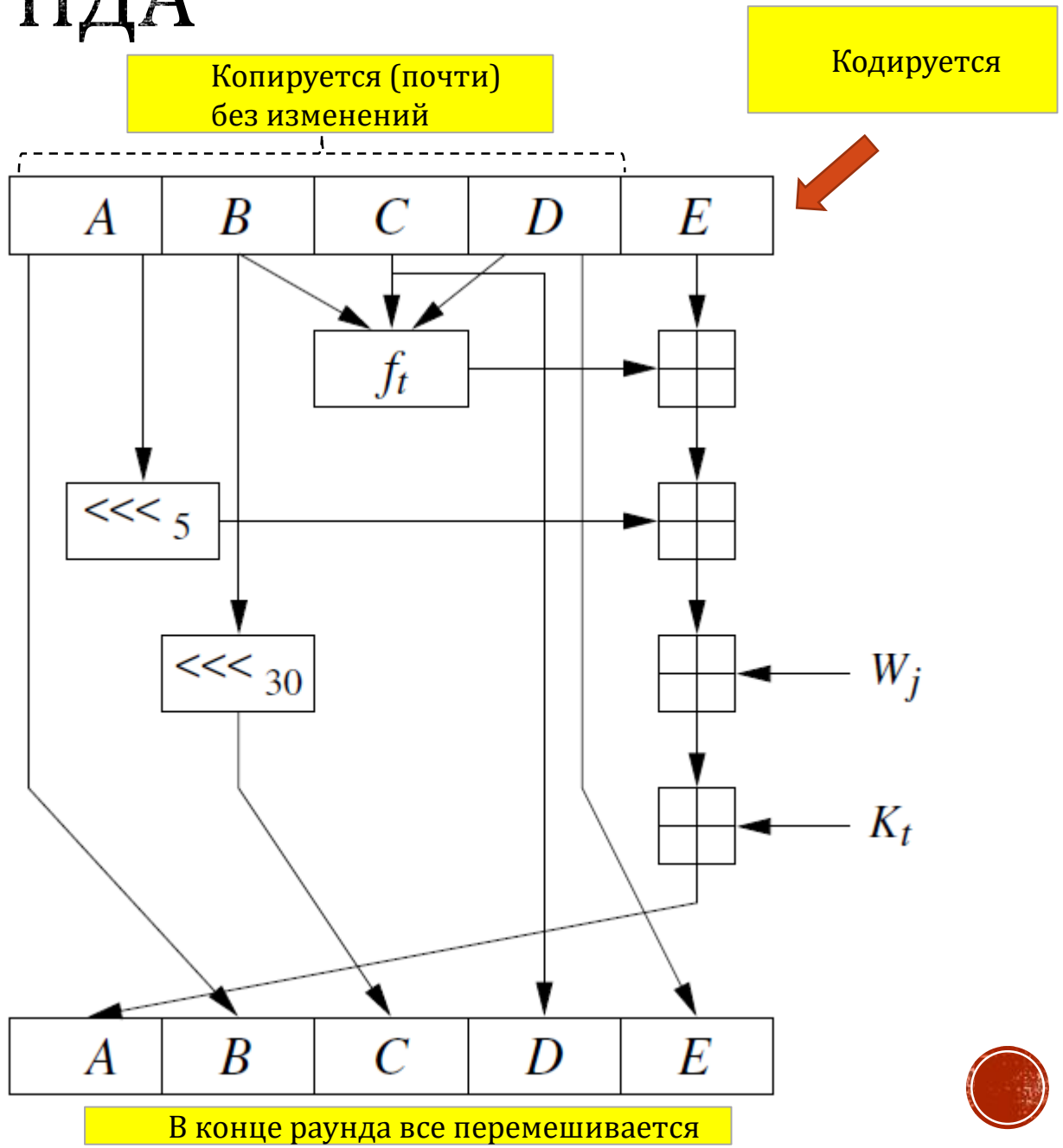
Диапазоны индексов величин, используемых на слайдах:

- $i = 1, \ldots, n$ – *номера блоков исходного сообщения x*
- $j = 0, \ldots, 79$ – *номера раундов на одной итерации функции сжатия*
- $t = 1, 2, 3, 4$ – *номер группы раундов (4 группы по 20 раундов)*



СТРУКТУРА ОДНОГО РАУНДА

- Структура раунда называется обобщенной (или несбалансированной) сетью Фейстеля:
  - часть исходного сообщения (E) кодируется с помощью других частей
  - остальные четыре части не кодируются
  - все пять частей сдвигаются друг относительно друга
- Все раунды различаются только функцией f_t и константой K_t: для первых двадцати раундов $t = 1$, для следующих двадцати $t = 2$ и т.д.



ЗНАЧЕНИЯ РАУНДОВЫХ ПАРАМЕТРОВ

Битовое НЕ!

Группа	Раунд	Раундовая константа	Раундовая функция
1	0...19	$K_1 = \texttt{5A827999}$	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	20...39	$K_2 = \texttt{6ED9EBA1}$	$f_2(B,C,D) = B \oplus C \oplus D$
3	40...59	$K_3 = \texttt{8F1BBCDC}$	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	60...79	$K_4 = \texttt{CA62C1D6}$	$f_4(B,C,D) = B \oplus C \oplus D$

ЗАДАНИЕ №2

- Реализовать хеш-функцию **SHA-1**
- Функция принимает на вход только текстовые сообщения, состоящие из символов латиницы, цифр и пунктуационных знаков

Технические детали:

- Взять в Питоне код символа: **ord('H')**
- Вывести шестнадцатеричное представление на печать: **hex(45)**
- Сдвинуть на **n** бит влево: **1 << n**
- Логические операции: | - или, **&** - и, **^ - XOR**, ~ - не
- Итерации $i = 1, \dots, n$ реализовать с помощью **генератора**
- Выдачу W_j реализовать с помощью **замыкания**

ЗАДАНИЕ №2

Пример функции, порождающей замыкания для генерации W:

```
def genW(x):
    # Замыкание - выдает раундовые сообщения W[j].
    # На вход получает x - 512-битное число.

    # Создаем список для всех 80 значений W. Его мы и "замкнем"
    W = [0] * 80

    # Заранее вычисляем все 80 значений
    for i in range(80):
        #...

    # Создаем замыкание и возвращаем его
    def makeGenerator(j):
        return W[j]

    return makeGenerator
```

ЗАДАНИЕ №2

Пример главной функции, использующей генератор и замыкание:

```
def sha1(message):
    # стандартные значения для инициализации алгоритма (H[0])
    H = [0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0]

    # используем генератор для выдачи x[i]
    for x in padding(message):
        # создаем замыкание, которое будет нам возвращать "раундовые" сообщения W[j]
        W = genW(x)

        # проходим все 80 раундов и перевычисляем A, B, C, D, E
        for j in range(80):
            # ...

        # суммируем результирующие значения A, B, C, D, E с исходными

    # выводим результат в шестнадцатеричном представлении
    return makeHashStr(H)


def makeHashStr(H):
    # делаем из списка 32-битных слов одно большое число и выводим его
    # в шестнадцатеричном представлении (с добавлением опущенных лидирующих нулей)
    res = hex(reduce(lambda a, b: (a << 32) | b, H, 0))[2:]
    return res if len(res) == 40 else '0' * (40 - len(res)) + res
```