Main program:

000000000000000000000000000000000000000000000000000000000000 0x00 NaN

000000000000000000000000000000000000000000000000000000000000 0x20 NaN

000000000000000000000000000000000000000000000000000000000000 0x40 Free memory pointer

000000000000000000000000000000000000000000000000000000000000 0x60 NaN

000000000000000000000000000000000000000000000000000000000000 0x80 length bytes var1

000000000000000000000000000000000000000000000000000000000000 0xa0 length bytes var2

000000000000000000000000000000000000000000000000000000000000 0xc0

000000000000000000000000000000000000000000000000000000000000 0xe0

000000000000000000000000000000000000000000000000000000000000 0x100

0x00 0x12345678

0x04 000000000000000000000000000000000000000000000000000000000040 0x00 pointer var 1

0x24 000000000000000000000000000000000000000000000000000000000080 0x20 pointer var 2

0x44 000000000000000000000000000000000000000000000000000000000002 0x40 lenght var 1

0x64 000000000000000000000000000000000000000000000000000000000a0b 0x60 var1

0x84 000000000000000000000000000000000000000000000000000000000003 0x80 length var 2

0xa4 0000000000000000000000000000000000000000000000000000000c0d0e 0xa0 var 2



Solution to the call program:

0x6e6ecf96

0000000000000000000000000000000000000000000000000000000000000020

0000000000000000000000000000000000000000000000000000000000000011

00000000000000000000000000000000000000000000000000028e44ea450856

0000000000000000000000000000000000000000000000000000000000000580

00000000000000000000000000000000000000000000000000028e44ea450d4c

000000000000000000000000000000000000000000000000000000000000144e

00000000000000000000000000000000000000000000000000028e44ea455985

0000000000000000000000000000000000000000000000000000000000000b08

00000000000000000000000000000000000000000000000000028e44ea455894

0000000000000000000000000000000000000000000000000000000000000952

00000000000000000000000000000000000000000000000000028e44ea45d436

00000000000000000000000000000000000000000000000000000000000036a6

00000000000000000000000000000000000000000000000000028e44ea45c69f

0000000000000000000000000000000000000000000000000000000000003ee6

00000000000000000000000000000000000000000000000000028e44ea450a96

0000000000000000000000000000000000000000000000000000000000007a66

00000000000000000000000000000000000000000000000000028e44ea450de4

00000000000000000000000000000000000000000000000000028e44ea450877

0x6e6ecf960000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000001100000000000000000000000000000000000000000000000000028e44ea450856000000000000000000000000000000000000000000000000000000000000058000000000000000000000000000000000000000000000000000028e44ea450d4c000000000000000000000000000000000000000000000000000000000000144e00000000000000000000000000000000000000000000000000028e44ea4559850000000000000000000000000000000000000000000000000000000000000b0800000000000000000000000000000000000000000000000000028e44ea455894000000000000000000000000000000000000000000000000000000000000095200000000000000000000000000000000000000000000000000028e44ea45d43600000000000000000000000000000000000000000000000000000000000036a600000000000000000000000000000000000000000000000000028e44ea45c69f0000000000000000000000000000000000000000000000000000000000003ee600000000000000000000000000000000000000000000000000028e44ea450a960000000000000000000000000000000000000000000000000000000000007a6600000000000000000000000000000000000000000000000000028e44ea450de400000000000000000000000000000000000000000000000000028e44ea450877

Created contract that has to return success, hence reach the stop



stop at 2b6

loop from 95 - 1a5

calldata:

0x6e6ecf96 + 2 words + 0 wei

0x6e6ecf96

0000000000000000000000000000000000000000000000000000000000000020

0000000000000000000000000000000000000000000000000000000000000011

0000000000000000000000000000000000000000000000000000000000000021

0000000000000000000000000000000000000000000000000000000000000580

000000000000000000000000000000000000000000000000000000000000053b

000000000000000000000000000000000000000000000000000000000000144e

00000000000000000000000000000000000000000000000000000000000051f2

0000000000000000000000000000000000000000000000000000000000000b08

00000000000000000000000000000000000000000000000000000000000050e3

0000000000000000000000000000000000000000000000000000000000000952

000000000000000000000000000000000000000000000000000000000000dc41

00000000000000000000000000000000000000000000000000000000000036a6

000000000000000000000000000000000000000000000000000000000000cee8

0000000000000000000000000000000000000000000000000000000000003ee6

00000000000000000000000000000000000000000000000000000000000002e1

0000000000000000000000000000000000000000000000000000000000007a66

0000000000000000000000000000000000000000000000000000000000000593

This calldata passes the first and the second loop. The second loop ends at 0x1ff

0x6e6ecf960000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000001100000000000000000000000000000000000000000000000000000000000000210000000000000000000000000000000000000000000000000000000000000580000000000000000000000000000000000000000000000000000000000000053b000000000000000000000000000000000000000000000000000000000000144e00000000000000000000000000000000000000000000000000000000000051f20000000000000000000000000000000000000000000000000000000000000b0800000000000000000000000000000000000000000000000000000000000050e30000000000000000000000000000000000000000000000000000000000000952000000000000000000000000000000000000000000000000000000000000dc4100000000000000000000000000000000000000000000000000000000000036a6000000000000000000000000000000000000000000000000000000000000cee80000000000000000000000000000000000000000000000000000000000003ee600000000000000000000000000000000000000000000000000000000000002e10000000000000000000000000000000000000000000000000000000000007a660000000000000000000000000000000000000000000000000000000000000593

Constraints:

// word2 + i <= 31 -> word2 <= 0x11

// i + 15 < word2+i -> word2 > 0x0f

// i + 0x11 < word2+i+1 -> word2 > 0x10

// mem(0x480 + (i << 5)) == mem(0x0480 + (i + 0x11 << 5)))

0x0460 -> word2

0x0480 -> word3

Memory layout with 0x11 as second word

000000000000000000000000000000000000000000000000000000000000001a

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000011

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000 0x400

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000012

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000 0x600

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000021 0x6a0

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000000000000000000000000000000000010f1

0000000000000000000000000000000000000000000000000000000000000533

0000000000000000000000000000000000000000000000000000000000000021

000000000000000000000000000000000000000000000000000000000000004d

00000000000000000000000000000000000000000000000000000000000004d2

00000000000000000000000000000000000000000000000000000000000010e1

0000000000000000000000000000000000000000000000000000000000000000

Current progress

object "contract" {

    code { }

    object "runtime" {

        code {

                mstore(0x40, 0x80)

                let \_0 := iszero(callvalue())

                require(not(\_0))

                let \_1 := lt(calldatasize(), 0x4)

                if not(\_1){

                    let \_2 := shr(0xe0, calldataload(0x0))

                    let \_3 := eq(0x6356ce4f, \_2)

                    switch \_2

                        case 0x6356ce4f{

                            func\_0x6356ce4f()

                        }

                        default { }

                }

                func\_ROOT4146650865()

            function func\_0x6356ce4f() {

                    let \_4 := add(0x4, sub(calldatasize(), 0x4))

                    let \_5 := iszero(slt(sub(\_4, 0x4), 0x40))

                    require(not(\_5))

                    let \_6 := calldataload(add(0x4, 0x0))                       // pointer first bytes

                    let \_7 := iszero(gt(\_6, 0xffffffffffffffff))

                    require(not(\_7))

                    let \_8 := add(0x4, \_6)

                    let \_9 := slt(add(\_8, 0x1f), \_4)

                    require(not(\_9))

                    let \_10 := calldataload(\_8)                                 // length first bytes

                    let \_11 := add(\_8, 0x20)

                    let \_12 := iszero(gt(\_10, 0xffffffffffffffff))

                    if \_12{

                        let \_13 := mload(0x40)

                        let \_14 := add(\_13, and(add(add(and(add(\_10, 0x1f), not(0x1f)), 0x20), 0x1f), not(0x1f)))

                        let \_15 := iszero(or(gt(\_14, 0xffffffffffffffff), lt(\_14, \_13)))

                        if \_15{

                            mstore(0x40, \_14)

                            mstore(\_13, \_10)

                            let \_16 := add(\_13, 0x20)

                            let \_17 := iszero(gt(add(\_11, \_10), \_4))

                            require(not(\_17))

                            calldatacopy(\_16, \_11, \_10)

                            mstore(add(\_16, \_10), 0x0)

                            let \_18 := calldataload(add(0x4, 0x20))

                            let \_19 := iszero(gt(\_18, 0xffffffffffffffff))

                            require(not(\_19))

                            let \_20 := add(0x4, \_18)

                            let \_21 := slt(add(\_20, 0x1f), \_4)

                            require(not(\_21))

                            let \_22 := calldataload(\_20)

                            let \_23 := add(\_20, 0x20)

                            let \_24 := iszero(gt(\_22, 0xffffffffffffffff))

                            if \_24{

                                let \_25 := mload(0x40)

                                let \_26 := add(\_25, and(add(add(and(add(\_22, 0x1f), not(0x1f)), 0x20), 0x1f), not(0x1f)))

                                let \_27 := iszero(or(gt(\_26, 0xffffffffffffffff), lt(\_26, \_25)))

                                if \_27{

                                    mstore(0x40, \_26)

                                    mstore(\_25, \_22)

                                    let \_28 := add(\_25, 0x20)

                                    let \_29 := iszero(gt(add(\_23, \_22), \_4))

                                    require(not(\_29))

                                    calldatacopy(\_28, \_23, \_22)

                                    mstore(add(\_28, \_22), 0x0)

                    // Here starts the actual implementation of the function, all other code are just checks for the calldata parameters (bytes memory var1, bytes memory var2)

                                    let \_30 := mload(0x40)           // Load the free memory pointer

                                    mstore(0x40, add(0x40, \_30))     // Update the fmp with 2 more words because a string is saved first the length and then the actual bytes

                                    mstore(\_30, 0xd)             // First store the length of the string (0xd) Almost there! are 13 characters, hence 0xd

                                    mstore(add(0x20, \_30), 0x616c6d6f73742074686572652100000000000000000000000000000000000000)      // Save the actual string "Almost there!"

                                    let \_31 := mload(add(\_25, 0x20)) // 32 bytes second calldata bytes

                                    let \_32 := mload(add(\_30, 0x20)) // almost there!

                                    let \_33 := eq(\_32, \_31)      // require the first 32 bytes of the second bytes to be "almost there!"

                                    require(not(\_33))

                                    let \_34 := mload(0x40)

                                    mstore(0x40, add(0x720, \_34))

                                    mstore(\_34, 0x6ed)

                                    codecopy(add(0x20, \_34), 0x475, 0x6ed)

                                    // At this moment the memory layout is the following:

                                    //  first bytes argument -> length + actual bytes

                                    //  second bytes argument -> length + actual bytes

                                    //  saved "almost there!" -> length + actual bytes

                                    //  saved a chunk of bytes-> length + actual bytes

                                    let \_35 := mload(\_34)

                                    let \_36 := create(0x0, add(\_34, 0x20), \_35)  // address of the contract created

                                    let \_37 := mload(0x40)   // current fmp

                                    let \_38 := mload(\_13)    // 0x80 initial fmp

                                    let \_39 := 0x0                      // num1 = 0

                                    let \_40 := 0x0                      // num2 = 0

                                    for {

                                        let \_41 := lt(\_39, \_38)         // condition1  num1 < 0x80

                                        let \_42 := 0x20                 // num3 = 0x20

                                        let \_43 := add(\_40, \_42)        // num4 = num2 + num3 -> 0x20

                                    }

                                    not(iszero(\_41))      // \_39 needs to be greater or equal to \_38 to exit loop?

                                    { } // No incremental actions

                                    {

                                        \_39 := \_43                                    // num1 = num4

                                        \_41 := lt(\_39, \_38)                           //

                                        let \_44 := not(iszero(\_41))

                                        \_40 := \_43

                                        let \_45 := mload(add(add(\_13, 0x20), \_40))

                                        mstore(add(\_37, \_40), \_45)

                                        \_42 := 0x20

                                        \_43 := add(\_40, \_42)

                                        \_39 := \_43

                                        \_40 := \_43

                                    }

                                    mstore(add(\_37, \_38), 0x0)

                                    let \_46 := mload(0x40)

                                    let \_47 := call(gas(), and(0xffffffffffffffffffffffffffffffffffffffff, \_36), 0x0, \_46, sub(add(\_37, \_38), \_46), \_46, 0x0)

                                    // The calldata starts at the current fmp and loads (\_37 + \_38 - \_46)

                                    let \_48 := returndatasize()

                                    let \_49 := eq(\_48, 0x0)

                                    switch \_48

                                        case 0x0{ }

                                        default {

                                            let \_50 := mload(0x40)

                                            mstore(0x40, add(\_50, and(add(returndatasize(), 0x3f), not(0x1f))))

                                            mstore(\_50, returndatasize())

                                            returndatacopy(add(\_50, 0x20), 0x0, returndatasize())

                                        }

                                    if \_47{

                                        stop()

                                    }

                                    if not(\_47){

                                        mstore(0x0, 0x4e487b7100000000000000000000000000000000000000000000000000000000)

                                        mstore(0x4, 0x1)

                                        revert(0x0, 0x24)

                                    }

                // End of the implementation

                                }

                                if not(\_27){

                                    mstore(0x0, 0x4e487b7100000000000000000000000000000000000000000000000000000000)

                                    mstore(0x4, 0x41)

                                    revert(0x0, 0x24)

                                }

                            }

                            if not(\_24){

                                mstore(0x0, 0x4e487b7100000000000000000000000000000000000000000000000000000000)

                                mstore(0x4, 0x41)

                                revert(0x0, 0x24)

                            }

                        }

                        if not(\_15){

                            mstore(0x0, 0x4e487b7100000000000000000000000000000000000000000000000000000000)

                            mstore(0x4, 0x41)

                            revert(0x0, 0x24)

                        }

                    }

                    if not(\_12){

                        mstore(0x0, 0x4e487b7100000000000000000000000000000000000000000000000000000000)

                        mstore(0x4, 0x41)

                        revert(0x0, 0x24)

                    }

                }

            function func\_ROOT4146650865() {

                    revert(0x0, 0x0)

                }

            function require(condition) {

                    if iszero(condition){

                        revert(0x0, 0x0)

                    }

                }

        }

    }

}

[00]    PUSH1   00          // [i]

[02]    PUSH1   0f          // [15, i]

[04]    SWAP1               // [i, 15]

i=0  while i<15

[05]    JUMPDEST

[06]    DUP1                // [i, i, 15]

// store the counter at memory position 0x0940

[07]    PUSH2   0940

[0a]    MSTORE

// word at memory 0x0460 must be less than or equal to 31 or 0x1f

// word2 + 1 <= 31

[0b]    PUSH2   0460

[0e]    MLOAD               // [word2, i, 15]

[0f]    PUSH1   1f          // [1f, word2, i, 15]

[11]    DUP2                // [word2, 1f, word2, i, 15]

[12]    GT                  // [word2 > 1f, word2, i, 15]

[13]    PUSH2   06aa        // [word2 > 1f, word2, i, 15]

[16]    JUMPI

[17]    PUSH2   0940        // [0x0940, word2, i, 15]

[1a]    MLOAD               // [i, word2, i, 15]

[1b]    PUSH2   0940        // [0x0940, i, word2, i, 15]

[1e]    MLOAD               // [i, i, word2, i, 15]

[1f]    PUSH1   03          // [3, i, i, word2, i, 15]

[21]    DUP2                // [i, 3, i, i, word2, i, 15]

[22]    AND                 // [i & 3, i, i, word2, i, 15]

[23]    SWAP1               // [i, i & 3, i, word2, i, 15]

[24]    POP                 // [i & 3, i, word2, i, 15]

[25]    PUSH1   05          // [5, i & 3, i, word2, i, 15]

[27]    DUP2                // [i & 3, 5, i & 3, i, word2, i, 15]

// word at position 0x940

[28]    GT                  // [i & 3 > 5, i & 3, i, word2, i, 15]

[29]    PUSH2   06aa

[2c]    JUMPI               // [i & 3, i, word2, i, 15]

[2d]    PUSH1   05          // [5, i & 3, i, word2, i, 15]

[2f]    SHL                 // [(i & 3) << 5, i, word2, i, 15]

[30]    PUSH2   0880        // [0x880, (i & 3) << 5, i, word2, i, 15]

[33]    ADD                 // [0x880 + (i & 3) << 5, i, word2, i, 15]

// this has a pattern

// 0    0x0880

// 1    0x08a0

// 2    0x08c0

// 3    0x08e0

// 4    0x0880

// 5    0x08a0

// 6    0x08c0

// 7    0x08e0

// 8    0x0880

// 9    0x08a0

// a    0x08c0

// b    0x08e0

// c    0x0880

// d    0x08a0

// e    0x08c0

[34]    MLOAD               // [memoryPos, i, word2, i, 15]

[35]    PUSH2   0940        // [0x940, memoryPos, i, word2, i, 15]

[38]    MLOAD               // [i, memoryPos, i, word2, i, 15]

[39]    XOR                 // [xor(i, memoryPos), i, word2, i, 15]

[3a]    DUP1                // [xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[3b]    DUP3                // [i, xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[3c]    MUL                 // [i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[3d]    DUP2                // [xor(i, memoryPos), i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[3e]    ISZERO              // [xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[3f]    DUP4                // [i, xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[40]    DUP4                // [xor(i, memoryPos), i, xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[41]    DUP4                // [i \* xor(i, memoryPos), xor(i, memoryPos), i, xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[42]    DIV                 // [i \* xor(i, memoryPos) / xor(i, memoryPos), i, xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[43]    EQ                  // [i \* xor(i, memoryPos) / xor(i, memoryPos) == i, xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[44]    OR                  // [i \* xor(i, memoryPos) / xor(i, memoryPos) == i || xor(i, memoryPos) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[45]    ISZERO              // [(i \* xor(i, memoryPos) / xor(i, memoryPos) == i || xor(i, memoryPos) == 0) == 0, i \* xor(i, memoryPos), xor(i, memoryPos), i, word2, i, 15]

[46]    PUSH2   06aa

[49]    JUMPI

// 0x0880 -> 10f1

// 0x08a0 -> 0533

// 0x08c0 -> 21

// 0x08e0 -> 4d

// 0x0900 -> 04d2

// 0x0920 -> 10e1

// Require any of those 2 conditions

// (i \* xor(i, memoryPos) / xor(i, memoryPos) == i      this should always hold and actually does

// xor(i, memoryPos) == 0

[4a]    SWAP1               // [xor(i, memoryPos), i \* xor(i, memoryPos), i, word2, i, 15]

[4b]    POP                 // [i \* xor(i, memoryPos), i, word2, i, 15]

[4c]    SWAP1               // [i, i \* xor(i, memoryPos), word2, i, 15]

[4d]    POP                 // [i \* xor(i, memoryPos), word2, i, 15]

[4e]    PUSH2   0940

[51]    MLOAD               // [i, i \* xor(i, memoryPos), word2, i, 15]

[52]    PUSH1   02          // [2, i, i \* xor(i, memoryPos), word2, i, 15]

[54]    DUP2                // [i, 2, i, i \* xor(i, memoryPos), word2, i, 15]

[55]    ADD                 // [i + 2, i, i \* xor(i, memoryPos), word2, i, 15]

[56]    DUP2                // [i, i + 2, i, i \* xor(i, memoryPos), word2, i, 15]

[57]    DUP2                // [i + 2, i, i + 2, i, i \* xor(i, memoryPos), word2, i, 15]

[58]    LT                  // [i + 2 < i, i + 2, i, i \* xor(i, memoryPos), word2, i, 15]

[59]    PUSH2   06aa

[5c]    JUMPI

// i + 2 >= i this should always hold

[5d]    SWAP1               // [i, i + 2, i \* xor(i, memoryPos), word2, i, 15]

[5e]    POP                 // [i + 2, i \* xor(i, memoryPos), word2, i, 15]

[5f]    PUSH1   06          // [6, i + 2, i \* xor(i, memoryPos), word2, i, 15]

[61]    DUP2                // [i + 2, 6, i + 2, i \* xor(i, memoryPos), word2, i, 15]

[62]    MOD                 // [(i + 2) % 6, i + 2, i \* xor(i, memoryPos), word2, i, 15]

[63]    SWAP1               // [i + 2, (i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[64]    POP                 // [(i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[65]    PUSH1   05          // [5, (i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[67]    DUP2                // [(i + 2) % 6, 5, (i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[68]    GT                  // [(i + 2) % 6 > 5, (i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[69]    PUSH2   06aa

[6c]    JUMPI               // [(i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

// (i + 2) % 6 <= 5 this should always hold

[6d]    PUSH1   05          // [5, (i + 2) % 6, i \* xor(i, memoryPos), word2, i, 15]

[6f]    SHL                 // [(i + 2) % 6 << 5, i \* xor(i, memoryPos), word2, i, 15]

[70]    PUSH2   0880        // [0x880, (i + 2) % 6 << 5, i \* xor(i, memoryPos), word2, i, 15]

[73]    ADD                 // [0x880 + (i + 2) % 6 << 5, i \* xor(i, memoryPos), word2, i, 15]

[74]    MLOAD               // [memoryPos2, i \* xor(i, memoryPos), word2, i, 15]                                             memoryPos2 => 0x880 + (((i + 2) % 6) << 5)

[75]    DUP1                // [memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[76]    DUP3                // [i \* xor(i, memoryPos), memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[77]    ADD                 // [i \* xor(i, memoryPos) + memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[78]    DUP3                // [i \* xor(i, memoryPos), i \* xor(i, memoryPos) + memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[79]    DUP2                // [i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos), i \* xor(i, memoryPos) + memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[7a]    LT                  // [i \* xor(i, memoryPos) + memoryPos2 < i \* xor(i, memoryPos), i \* xor(i, memoryPos) + memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[7b]    PUSH2   06aa

[7e]    JUMPI               // [i \* xor(i, memoryPos) + memoryPos2, memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

// i \* xor(i, memoryPos) + memoryPos2 >= i \* xor(i, memoryPos) this should always hold

[7f]    SWAP1               // [memoryPos2, i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[80]    POP                 // [i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos), word2, i, 15]

[81]    SWAP1               // [i \* xor(i, memoryPos), i \* xor(i, memoryPos) + memoryPos2, word2, i, 15]

[82]    POP                 // [i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[83]    PUSH2   0940

[86]    MLOAD               // [i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[87]    DUP1                // [i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[88]    DUP3                // [i \* xor(i, memoryPos) + memoryPos2, i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[89]    ADD                 // [i \* xor(i, memoryPos) + memoryPos2 + i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[8a]    DUP3                // [i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos) + memoryPos2 + i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[8b]    DUP2                // [i \* xor(i, memoryPos) + memoryPos2 + i, i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos) + memoryPos2 + i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[8c]    LT                  // [i \* xor(i, memoryPos) + memoryPos2 + i < i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos) + memoryPos2 + i, i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[8d]    PUSH2   06aa

[90]    JUMPI               // [i \* xor(i, memoryPos) + memoryPos2 + i, i, i \* xor(i, memoryPos) + memoryPos2, word2, i, 15]

// i \* xor(i, memoryPos) + memoryPos2 + i >= i \* xor(i, memoryPos) + memoryPos2 this should always hold

[91]    SWAP1               // [i, i \* xor(i, memoryPos) + memoryPos2 + i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[92]    POP                 // [i \* xor(i, memoryPos) + memoryPos2 + i, i \* xor(i, memoryPos) + memoryPos2, , word2, i, 15]

[93]    SWAP1               // [i \* xor(i, memoryPos) + memoryPos2, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[94]    POP                 // [i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[95]    PUSH2   0940

[98]    MLOAD               // [i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[99]    PUSH1   0f          // [15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[9b]    DUP2                // [i, 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[9c]    ADD                 // [i + 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[9d]    DUP2                // [i, i + 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[9e]    DUP2                // [i + 15, i, i + 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[9f]    LT                  // [i + 15 < i, i + 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[a0]    PUSH2   06aa

[a3]    JUMPI               // [i + 15, i, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

// i + 15 >= i  this should always hold

[a4]    SWAP1               // [i, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[a5]    POP                 // [i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[a6]    PUSH2   0460

[a9]    MLOAD               // [word2, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[aa]    DUP2                // [i + 15, word2, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[ab]    LT                  // [i + 15 < word2, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[ac]    ISZERO              // [i + 15 < word2 == 0, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[ad]    PUSH2   06aa

[b0]    JUMPI               // [i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

// i + 15 < word2+i

[b1]    PUSH1   05          // [5, i + 15, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[b3]    SHL                 // [i + 15 << 5, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[b4]    PUSH2   0480        // [0x480, i + 15 << 5, i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[b7]    ADD                 // [0x480 + (i + 15 << 5), i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[b8]    MLOAD               // [mem(0x480 + (i + 15 << 5)), i \* xor(i, memoryPos) + memoryPos2 + i, word2, i, 15]

[b9]    XOR                 // [xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[ba]    DUP2                // [word2, xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[bb]    PUSH1   05          // [5, word2, xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[bd]    SHL                 // [word2 << 5, xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[be]    PUSH2   0480        // [0x480, word2 << 5, xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[c1]    ADD                 // [0x480 + (word2 << 5), xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i), word2, i, 15]

[c2]    MSTORE              // [word2, i, 15]           store at position 0x480 + (word2 << 5)      this value xor(mem(0x480 + (i + 15 << 5))   ,    i \* xor(i, memoryPos) + memoryPos2 + i)

[c3]    PUSH1   01          // [1, word2, i, 15]

[c5]    DUP2                // [word2, 1, word2, i, 15]

[c6]    ADD                 // [word2 + 1, word2, i, 15]

[c7]    PUSH2   0460

[ca]    MSTORE              // [word2, i, 15]           stored at the memory where the word2 was located word2 + 1

[cb]    POP                 // [i, 15]

[cc]    PUSH2   0940

[cf]    MLOAD               // [i, i, 15]

[d0]    PUSH1   11          // [0x11, i, i, 15]

[d2]    DUP2                // [i, 0x11, i, i, 15]

[d3]    ADD                 // [i + 0x11, i, i, 15]

[d4]    DUP2                // [i, i + 0x11, i, i, 15]

[d5]    DUP2                // [i + 0x11, i, i + 0x11, i, i, 15]

[d6]    LT                  // [i + 0x11 < i, i + 0x11, i, i, 15]

[d7]    PUSH2   06aa

[da]    JUMPI               // [i + 0x11, i, i, 15]

// i + 0x11 >= i  this should always hold

[db]    SWAP1               // [i, i + 0x11, i, 15]

[dc]    POP                 // [i + 0x11, i, 15]

[dd]    PUSH2   0460

[e0]    MLOAD               // [word2+1, i + 0x11, i, 15]

[e1]    DUP2                // [i + 0x11, word2+1, i + 0x11, i, 15]

[e2]    LT                  // [i + 0x11 < word2+1, i + 0x11, i, 15]

[e3]    ISZERO              // [i + 0x11 < word2+1 == 0, i + 0x11, i, 15]

[e4]    PUSH2   06aa

[e7]    JUMPI               // [i + 0x11, i, 15]

// i + 0x11 < word2+i+1

[e8]    PUSH1   05          // [5, i + 0x11, i, 15]

[ea]    SHL                 // [i + 0x11 << 5, i, 15]

[eb]    PUSH2   0480        // [0x0480, i + 0x11 << 5, i, 15]

[ee]    ADD                 // [0x0480 + (i + 0x11 << 5), i, 15]

[ef]    MLOAD               // [mem(0x0480 + (i + 0x11 << 5)), i, 15]

[f0]    PUSH2   0940

[f3]    MLOAD               // [i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[f4]    PUSH2   0460

[f7]    MLOAD               // [word2+1, i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[f8]    DUP2                // [i, word2+1, i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[f9]    LT                  // [i < word2+1, i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[fa]    ISZERO              // [i < word2+1 == 0, i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[fb]    PUSH2   06aa

[fe]    JUMPI               // [i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

// i < word2+i+1 this should always hold

[ff]    PUSH1   05          // [5, i, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[101]   SHL                 // [i << 5, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[102]   PUSH2   0480        // [0x480, i << 5, mem(0x0480 + (i + 0x11 << 5)), i, 15]

[105]   ADD                 // [0x480 + (i << 5), mem(0x0480 + (i + 0x11 << 5)), i, 15]

[106]   MLOAD               // [mem(0x480 + (i << 5)), mem(0x0480 + (i + 0x11 << 5)), i, 15]

[107]   XOR                 // [xor(mem(0x480 + (i << 5)),    mem(0x0480 + (i + 0x11 << 5))), i, 15]

[108]   PUSH2   06aa

[10b]   JUMPI               // [i, 15]

// mem(0x480 + (i << 5)) == mem(0x0480 + (i + 0x11 << 5)))

[10c]   PUSH1   01          // [1, i, 15]

[10e]   ADD                 // [i + 1, 15]

[10f]   DUP2                // [15, i + 1, 15]

[110]   DUP2                // [i + 1, 15, i + 1, 15]

[111]   XOR                 // [XOR(i + 1, 15), i + 1, 15]

[112]   PUSH2   0095

[115]   JUMPI               // [i + 1, 15]

// to leave loop i+1 == 15

Second loop:

[1a8]   PUSH1   00

[1aa]   PUSH1   0f

[1ac]   SWAP1

[1ad]   JUMPDEST

[1ae]   DUP1

[1af]   PUSH2   0940

[1b2]   MSTORE

[1b3]   PUSH2   0940

[1b6]   MLOAD

[1b7]   PUSH1   11

[1b9]   DUP2

[1ba]   ADD

[1bb]   DUP2

[1bc]   DUP2

[1bd]   LT

[1be]   PUSH2   06aa

[1c1]   JUMPI

[1c2]   SWAP1

[1c3]   POP

[1c4]   PUSH2   0460

[1c7]   MLOAD

[1c8]   DUP2

[1c9]   LT

[1ca]   ISZERO

[1cb]   PUSH2   06aa

[1ce]   JUMPI

[1cf]   PUSH1   05

[1d1]   SHL

[1d2]   PUSH2   0480

[1d5]   ADD

[1d6]   MLOAD

[1d7]   PUSH2   0940

[1da]   MLOAD

[1db]   PUSH2   0460

[1de]   MLOAD

[1df]   DUP2

[1e0]   LT

[1e1]   ISZERO

[1e2]   PUSH2   06aa

[1e5]   JUMPI

[1e6]   PUSH1   05

[1e8]   SHL

[1e9]   PUSH2   0480

[1ec]   ADD

[1ed]   MLOAD

[1ee]   XOR

[1ef]   PUSH2   06aa

[1f2]   JUMPI

[1f3]   PUSH1   01

[1f5]   ADD

[1f6]   DUP2

[1f7]   DUP2

[1f8]   XOR

[1f9]   PUSH2   01ad

[1fc]   JUMPI

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////           Second loop              ////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

[1a8]   PUSH1   00          // [0]

[1aa]   PUSH1   0f          // [f, 0]

[1ac]   SWAP1               // [0, f]

[1ad]   JUMPDEST

[1ae]   DUP1                // [i, i, f]

[1af]   PUSH2   0940

[1b2]   MSTORE              // store i at memory 0x940

[1b3]   PUSH2   0940        // [0x940, i, f]

[1b6]   MLOAD               // [i, i, f]

[1b7]   PUSH1   11          // [0x11, i, i, f]

[1b9]   DUP2                // [i, 0x11, i, i, f]

[1ba]   ADD                 // [i + 0x11, i, i, f]

[1bb]   DUP2                // [i, i + 0x11, i, i, f]

[1bc]   DUP2                // [i + 0x11, i, i + 0x11, i, i, f]

[1bd]   LT                  // [i + 0x11 < i, i + 0x11, i, i, f]

[1be]   PUSH2   06aa

[1c1]   JUMPI               // [i + 0x11, i, i, f]

// i + 0x11 >= i this should always hold

[1c2]   SWAP1               // [i, i + 0x11, i, f]

[1c3]   POP                 // [i + 0x11, i, f]

[1c4]   PUSH2   0460        // [0x460, i + 0x11, i, f]

[1c7]   MLOAD               // [word2, i + 0x11, i, f]

[1c8]   DUP2                // [i + 0x11, word2, i + 0x11, i, f]

[1c9]   LT                  // [i + 0x11 < word2, i + 0x11, i, f]

[1ca]   ISZERO              // [i + 0x11 < word2 == 0, i + 0x11, i, f]

[1cb]   PUSH2   06aa

[1ce]   JUMPI               // [i + 0x11, i, f]

// i + 0x11 < word2

[1cf]   PUSH1   05          // [5, i + 0x11, i, f]

[1d1]   SHL                 // [(i + 0x11) << 5, i, f]

[1d2]   PUSH2   0480        // [0x480, (i + 0x11) << 5, i, f]

[1d5]   ADD                 // [0x480 + (i + 0x11) << 5, i, f]

[1d6]   MLOAD               // [mem(0x480 + (i + 0x11) << 5), i, f]

[1d7]   PUSH2   0940        // [0x940, mem(0x480 + (i + 0x11) << 5), i, f]

[1da]   MLOAD               // [i, mem(0x480 + (i + 0x11) << 5), i, f]

[1db]   PUSH2   0460        // [0x460, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1de]   MLOAD               // [word2, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1df]   DUP2                // [i, word2, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1e0]   LT                  // [i < word2, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1e1]   ISZERO              // [i < word2 == 0, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1e2]   PUSH2   06aa

[1e5]   JUMPI               // [i, mem(0x480 + (i + 0x11) << 5), i, f]

// i < word2

[1e6]   PUSH1   05          // [5, i, mem(0x480 + (i + 0x11) << 5), i, f]

[1e8]   SHL                 // [i << 5, mem(0x480 + (i + 0x11) << 5), i, f]

[1e9]   PUSH2   0480        // [0x480, i << 5, mem(0x480 + (i + 0x11) << 5), i, f]

[1ec]   ADD                 // [0x480 + (i << 5), mem(0x480 + (i + 0x11) << 5), i, f]

[1ed]   MLOAD               // [mem(0x480 + (i << 5)), mem(0x480 + (i + 0x11) << 5), i, f]

[1ee]   XOR                 // [xor(mem(0x480 + (i << 5)),     mem(0x480 + (i + 0x11) << 5)), i, f]

[1ef]   PUSH2   06aa

[1f2]   JUMPI               // [i, f]

// mem(0x480 + (i << 5)) == mem(0x480 + (i + 0x11) << 5)

[1f3]   PUSH1   01          // [1, i, f]

[1f5]   ADD                 // [i + 1, f]

[1f6]   DUP2                // [f, i + 1, f]

[1f7]   DUP2                // [i + 1, f, i + 1, f]

[1f8]   XOR                 // [xor(i + 1, f), i + 1, f]

[1f9]   PUSH2   01ad

[1fc]   JUMPI               // [i + 1, f]

// to leave loop it is required that i == f