Key Scheduling Algorithm (KSA)

DLX Assembly Implementation

Written by Noam Yakar

The goal of KSA is generating a pseudo-random permutation of a 256-byte vector (S) based on a 16-byte key (K). KSA is known as the first step in the RC4 stream cipher algorithm.

In KSA, the first step is initializing a 256-byte vector (S) with the identity permutation, meaning the values in the array are equal to their index.
The next step is shuffling the array in a pseudo-random manner, making it a permutation array.

The annotated Assembly program is presented in the following pages. Hereby presented the C implementation of the algorithm. Though it seems quite short, the Assembly implementation is a little bit longer...

```
int i;
for (i = 0; i < 256; i++){
S[i] = i;
}

int j = 0;
for (i = 0; i < 256; i++){
j = mod(j + S[i] + key[mod(i,16)],256);
swap(S[i],S[j]);
}
```

Table of Contents

The DLX Assembly code.. **Page 2**

Full explanations of the program.. **Page 7**

Program's output validation.. **Page 10**

The DLX Assembly implementation

```
beqz R0 begin
pc = 0x50

begin: addi R10 R0 0xFFFF   # R10 = 0xFFFFFFFF
addi R12 R0 0xFF            # R12 = 0xFF
addi R15 R0 0x3             # R15 = 0x3
addi R2 R0 0xFFFF           # R2 = -1 (the i of the second for loop)
addi R3 R0 0x4              # R3 = 4
addi R4 R0 0x8              # R4 = 8, the S for the first loop
addi R20 R0 0x7             # R20 = 7, the S-1 for the second loop
addi R1 R0 0x0              # R1 = 0, this is j
addi R7 R0 0x4000           # R7 = 0x4000
slli R7 R7                  # R7 << 1
slli R7 R7                  # R7 << 1 -> R7 = 0x00010000
addi R7 R7 0x203            # R7 = 0x00010203 (S[0] S[1] S[2] S[3])

addi R8 R0 0x4040           # R8 = 0x4040
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1
slli R8 R8                  # R8 << 1 -> R8 = 0x04040000
addi R8 R8 0x404            # R8 = 0x04040404

LOOP1: sgei R6 R4 0x48      # if (S >= 8 + 64) - last address is 71
bnez R6 LOOP2               # finished first loop, branch to second loop
sw R7 R4 0x0                # S[i] = i
add R7 R7 R8               # S[i] -> S[i + 4]
addi R4 R4 0x1              # S = S + 1
beqz R0 LOOP1               # go to start of LOOP1

LOOP2: addi R2 R2 0x1       # i = i + 1
addi R20 R20 0x1            # S = S + 1 (word address)
seqi R5 R3 0x8              # R5=1 if R3=8 -> need to make R3=4
beqz R5 0x1                 # PC = PC + 2
addi R3 R0 0x4              # R3 = 4

```

```
lw R7 R20 0x0            # R7 = S[i] S[i + 1] S[i + 2] S[i + 3]
add R18 R0 R7            # R18 = S word, R18 is input reg for GETBYTE
addi R16 R0 0x0          # R16 = 0, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R9 R1 R27            # R9 = j + S[i], result of GETBYTE is in R27
lw R13 R3 0              # R13 = key[i%16] key[(i+1)%16] key[(i+2)%16] key[(i+3)%16]
add R18 R0 R13           # R18 = key word, R18 is input reg for GETBYTE
addi R16 R0 0x0          # R16 = 0, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R14 R9 R27           # R14 = j + S[i] + key[i%16], result of GETBYTE is in R27
and R1 R14 R12           # j = mod(j + S[i] + key[mod(i,16)] , 256)
addi R22 R0 SWAP         # R22 = address of SWAP
jalr R22                 # jalr to SWAP

lw R7 R20 0x0            # R7 = S[i] S[i + 1] S[i + 2] S[i + 3]
addi R2 R2 0x1           # i = i + 1
add R18 R0 R7            # R18 = S word, R18 is input reg for GETBYTE
addi R16 R0 0x1          # R16 = 1, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R9 R1 R27            # R9 = j + S[i + 1], result of GETBYTE is in R27
add R18 R0 R13           # R18 = key word, R18 is input reg for GETBYTE
addi R16 R0 0x1          # R16 = 1, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R14 R9 R27           # R14 = j + S[i + 1] + key[(i+1)%16], result of GETBYTE is in R27
and R1 R14 R12           # j = mod(j + S[i + 1] + key[mod((i + 1),16)] , 256)
addi R22 R0 SWAP         # R22 = address of SWAP
jalr R22                 # jalr to SWAP

lw R7 R20 0x0            # R7 = S[i] S[i + 1] S[i + 2] S[i + 3]
addi R2 R2 0x1           # i = i + 1
add R18 R0 R7            # R18 = S word, R18 is input reg for GETBYTE
addi R16 R0 0x2          # R16 = 2, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R9 R1 R27            # R9 = j + S[i + 2], result of GETBYTE is in R27
add R18 R0 R13           # R18 = key word, R18 is input reg for GETBYTE
addi R16 R0 0x2          # R16 = 2, R16 is input num for GETBYTE
addi R22 R0 GETBYTE      # R22 = address of GETBYTE
jalr R22                 # jalr to GETBYTE
add R14 R9 R27           # R14 = j + S[i + 2] + key[(i + 2)%16], result of GETBYTE is in R27
and R1 R14 R12           # j = mod(j + S[i + 2] + key[mod((i + 2),16)] , 256)
addi R22 R0 SWAP         # R22 = address of SWAP
jalr R22                 # jalr to SWAP
```

```
lw R7 R20 0x0               # R7 = S[i] S[i + 1] S[i + 2] S[i + 3]
addi R2 R2 0x1              # i = i + 1
add R18 R0 R7               # R18 = S word, R18 is input reg for GETBYTE
addi R16 R0 0x3             # R16 = 3, R16 is input num for GETBYTE
addi R22 R0 GETBYTE         # R22 = address of GETBYTE
jalr R22                    # jalr to GETBYTE
add R9 R1 R27               # R9 = j + S[i + 3], result of GETBYTE is in R27
add R18 R0 R13              # R18 = key word, R18 is input reg for GETBYTE
addi R16 R0 0x3             # R16 = 3, R16 is input num for GETBYTE
addi R22 R0 GETBYTE         # R22 = address of GETBYTE
jalr R22                    # jalr to GETBYTE
add R14 R9 R27              # R14 = j + S[i + 3] + key[(i + 3)%16], result of GETBYTE is in R27
and R1 R14 R12              # j = mod(j + S[i + 3] + key[mod((i + 3),16)] , 256)
addi R22 R0 SWAP            # R22 = address of SWAP
jalr R22                    # jalr to SWAP

addi R3 R3 0x1              # R3 = R3 + 1
sgei R25 R2 0xFF            # set R25 to 1 if i >= 255
beqz R25 LOOP2              # go to start of LOOP2
halt                        # halt - finish program

SWAP: add R28 R0 R31        # save the return address
add R23 R2 R0               # R23 = i
and R16 R23 R15             # R16 = 2 LSB digits of i
add R8 R0 R16               # R8 = R16, save R16 for later
srli R17 R23                # R17 = R23 >> 1 (i>>1)
srli R17 R17                # R17 >> (i>>1)
addi R5 R17 0x8             # R5 is the line address of S[i]
add R19 R0 R5               # R19 = R5, save R5 for later
lw R18 R5 0x0               # R18 = S[i] S[i+1] S[i+2] S[i+3]
addi R22 R0 GETBYTE         # R22 = address of GETBYTE
jalr R22                    # jalr to GETBYTE
add R4 R0 R27               # temp = R4 = S[i]

add R23 R1 R0               # R23 = j
and R16 R23 R15             # R16 = 2 LSB of j
add R21 R0 R16              # R21 = R16, save R16 for later
srli R17 R23                # R17 = R23 >> 1 (j>>1)
srli R17 R17                # R17 >> (j>>1)
addi R5 R17 0x8             # R5 is the line address of S[j]
lw R18 R5 0x0               # R18 = S[j] S[j+1] S[j+2] S[j+3]
addi R22 R0 GETBYTE         # R22 = address of GETBYTE
jalr R22                    # jalr to GETBYTE
add R6 R0 R27               # R6 = S[j]

lw R18 R19 0x0              # R18 = S[i] S[i+1] S[i+2] S[i+3]
add R16 R0 R8               # use the saved R16, R8, the inner location of S[i]
add R11 R0 R6               # R11 = S[j], R11 is input byte of PUTBYTE
addi R22 R0 PUTBYTE         # R22 = address of PUTBYTE
jalr R22                    # jalr to PUTBYTE
sw R18 R19 0x0              # store the new R18
```

```
lw R18 R5 0x0              # R18 = S[j] S[j+1] S[j+2] S[j+3]
add R16 R0 R21             # use the saved R16, R21, the inner location of S[j]
add R11 R0 R4              # R11 = temp, R11 is input byte of PUTBYTE
addi R22 R0 PUTBYTE        # R22 = address of PUTBYTE
jalr R22                   # jalr to PUTBYTE
sw R18 R5 0x0              # store the new R18
jr R28                     # go back to where SWAP was called

PUTBYTE: add R29 R0 R31    # save the return address
addi R22 R0 GETBMASK       # R22 = address of GETBMASK
jalr R22                   # jalr to GETBMASK
xor R26 R26 R10            # reverse bitmask, 1->0, 0->1, R26 is result of GETBMASK
and R18 R18 R26            # empty a byte in R18, ex: R18 = [ S[i] 0 S[i+2] S[i+3] ]
add R27 R0 R11             # R27 = S[j], R27 is input reg of MOVLEFT
addi R22 R0 MOVLEFT        # R22 = address of MOVLEFT
jalr R22                   # jalr to MOVLEFT
or R18 R27 R18             # insert byte
jr R29                     # go back to where PUTBYTE was called

GETBYTE: add R29 R0 R31    # save the return address
addi R22 R0 GETBMASK       # R22 = address of GETBMASK
jalr R22                   # jalr to GETBMASK
and R27 R26 R18            # mask the wanted byte, make other bytes zero
addi R22 R0 MOVRIGHT       # R22 = address of MOVRIGHT
jalr R22                   # jalr to MOVRIGHT
jr R29                     # go back to where GETBYTE was called
```

```
GETBMASK: addi R26 R0 0xFF  # R26 = 3rd byte bitmask
seqi R25 R16 0x3            # R25 = 1 if (R16 == 3), else 0
beqz R25 0x1                # if (R25 == 0) we don't want byte 3, skip next line
jr R31                      # we want byte 3, got it, now can return
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
seqi R25 R16 0x2            # R25 = 1 if (R16 == 2), else 0
beqz R25 0x1                # if (R25 == 0) we don't want byte 2, skip next line
jr R31                      # we want byte 2, got it, now can return
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
seqi R25 R16 0x1            # R25 = 1 if (R16 == 1), else 0
beqz R25 0x1                # if (R25 == 0) we don't want byte 1, skip next line
jr R31                      # we want byte 1, got it, now can return
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
slli R26 R26                # R26 = R26 << 1
jr R31                      # we want byte 0, got it, now can return
```

```
MOVRIGHT: sgti R25 R16 0x0  # R25 = 1 if (R16 > 0), else 0
bnez R25 0x8                # if (R25 != 0) skip next 8 lines
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
sgti R25 R16 0x1            # R25 = 1 if (R16 > 1), else 0
bnez R25 0x8                # if (R25 != 0) skip next 8 lines
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
sgti R25 R16 0x2            # R25 = 1 if (R16 > 2), else 0
bnez R25 0x8                # if (R25 != 0) skip next 8 lines
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
srli R27 R27                # R27 = R27 >> 1
jr R31                      # go back to where MOVRIGHT was called

```

```
MOVLEFT: seqi R25 R16 0x3    # R25 = 1 if (R16 == 3), else 0
beqz R25 0x1                 # if R25 == 0, skip a line
jr R31                       # we want byte 3, got it, now can return
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
seqi R25 R16 0x2             # R25 = 1 if (R16 == 2), else 0
beqz R25 0x1                 # if R25 == 0, skip a line
jr R31                       # we want byte 2, got it, now can return
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
seqi R25 R16 0x1             # R25 = 1 if (R16 == 1), else 0
beqz R25 0x1                 # if R25 == 0, skip a line
jr R31                       # we want byte 1, got it, now can return
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
slli R27 R27                 # R27 = R27 << 1
jr R31                       # we want byte 0, got it, now can return
```

Explanation of the program

The program contains two loops (LOOP1, LOOP2) and auxiliary functions.

In the beginning there's "pc = 0x50" that tells the compiler to map the rest of the code at a starting address 0x50, so there won't be an overlap between the code and data (input / output keys).

Under the Label "begin" there are initializations of register values that are needed for the rest of the program.

LOOP1: This loop implements the first FOR-loop of the algorithm. It initializes the S vector using two registers that were initialized in "begin": R7 = 0x00010203 and R8 = 0x04040404. In each iteration we add them together and store the result in memory. The loop ends after filling 64 words in addresses 0x8 – 0x47.

LOOP2: This loop implements the second FOR-loop of the algorithm. It contains four blocks, each one implements the FOR-loop content with a slight change, so in total there are 64 iterations of LOOP2.

The reason for using four blocks is that in the beginning of an iteration, one of the 4 input key words are loaded to R13 and used for the rest of the iteration. Each block takes a different byte from R13 and that's how the operation Key(mod(i, 16)) works. The extraction of a single byte from a 4-byte word is done by the function GETBYTE that will be explained later.

Another thing that needs to be achieved in a block is S(i). An S word is loaded to R7 in the beginning of the block and a byte is taken from it using the function GETBYTE.

The final j index of a block is stored in R1 and is used for calculating the new j of the next block.

After calculating j + S(i) + Key(mod(i, 16)), the modulo-256 operation is done by taking the 8 LSB bits of this number (using AND with 0xFF). Now that the final j has been achieved, the SWAP operation can take place.

At the end of a loop iteration, R3 is incremented by 1. The value of R3 indicates the word address of the key needed for the current iteration. Hence, it can only have values between 4-7. At the beginning of the loop there's a check if it reached 8, if so - it is set back to 4.

Another thing that happens at the end of a loop iteration is the check if R2 (index i) is greater or equal to 255. If so – we reach halt and the program ends. Else, start another iteration.

SWAP:

Input registers: R1 (index j), R2 (index i)

Output registers: R27

Functionality: The function swaps between S(i) and S(j) using the indexes i (R2) and j (R1).

The SWAP function contains 3 blocks: one for getting S(i), one for getting S(j), and one for the actual swapping.

To find the address at which S(i) is stored, the index i is divided by 4 and then added an offset of 8. Using this address, the word that contains S(i) is loaded to R18. The index of the specific byte to extract is a number between 0-3. It is achieved by taking the two LSB bits of i, and stored in R16. It's also saved in R8 for later usage. Then there's a call to the function GETBYTE that uses registers R18 and R16. The result, which is called "temp", is stored in R4.

In a similar manner S(j) is extracted and stored in R6. The index of the byte in the word is saved in R21 for later usage.

Now the actual swapping takes place. The word containing S(i) is loaded to R18. The saved content of R8 is stored in R16 and S(j) is stored in R11. Registers R18, R16 and R11 are used by the function PUTBYTE that puts S(j) in the required place in the word. After building this word, it is stored back in memory.

Then, the word containing S(j) is loaded to R18. The saved content of R21 is stored in R16 and S(i) is stored in R11. The function PUTBYTE is called, and the result is stored back in memory.

GETBYTE:

Input registers: R18, R16

Output registers: R27

Functionality: This function takes an inner byte from R18, based on a number between 0-3 in R16, and puts it in R27.

It calls the function GETBMASK that returns a bitmask containing 1's at the byte specified by R16. Then it performs bitwise AND between the word (R18) and the bitmask (R26). The result is a "byte among zeros", for example: [0 byte 0 0]. To have this byte at the LSB, it needs to be right shifted. This operation is done by the function MOVERIGHT that will be explained about later. The result is in R27.

PUTBYTE:

Input registers: R11, R16

Output registers: R18

Functionality: This function takes a byte from R11, and puts it as inner byte in R18, whereas the location is based on a number between 0-3 in R16.

It calls the function GETBMASK that returns a bitmask containing 1's at the byte specified by R16. This bitmask is reversed (1->0, 0->1) using bitwise XOR with R10 that contains only 1's.

Then the place where the new byte will be inserted in R18 is emptied by a bitwise AND between the reversed bitmask and R18.

The next step is to get the newly inserted byte to be in the correct place by shifting it left, which is done by the function MOVLEFT that will be explained about later. The result is a "byte among zeros". The insertion is done by a bitwise OR between this result and the one-byte-empty word in R18 from before.

<u>GETBMASK:</u>

Input registers: R16

Output registers: R26

Functionality: This function puts a bitmask in R26 according to a number between 0-3 in R16.

At the beginning of the function, R26 = 0xFF, which is a bitmask for the byte number 3. If this is indeed the bitmask needed (test instruction seqi) the function returns. Else, the bitmask is shifted 8 times using slli. The same process continues for the numbers 2, 1.

<u>MOVRIGHT:</u>

Input registers: R27, R16

Output registers: R27

Functionality: This function moves right a "byte among zeros" in R27, for example: [0 byte 0 0], according to a number between 0-3 in R16.

The goal is to move the byte to the LSB part. The indication of where the byte exists in the word is the value of R16. If R16 is greater than 0, it means the byte is not in the MSB part and we can skip the 8 lines that perform right shifts. If R16 is greater than 1, we can skip another 8 lines of right shifts. If R16 is greater than 2, the byte is in the LSB part, and no right shifts are needed.

<u>MOVLEFT:</u>

Input registers: R27, R16

Output registers: R27

Functionality: This function moves left a byte in R27, according to a number between 0-3 in R16. The result is a "byte among zeros", for example: [0 byte 0 0].

If R16 equals 3, the byte should stay at the LSB part and the function returns. Else, the byte is shifted 8 times using slli. The same test happens with the numbers 2, 1.

Validation

The program was tested on a DLX processor that was planned and built by me in the course "Computer Structure Advance Lab".

A random key was selected and used in a C implementation to get the expected output vector S. This output was then compared to the output generated by the assembly program that ran on the processor.

Key: "3cbb1690cf81be02d71244a2836b78a1"

Below are the input key (K) and the expected output (S):

KEY0	KEY1	KEY2	KEY3	KEY4	KEY5	KEY6	KEY7	KEY8	KEY9	KEY10	KEY11	KEY12	KEY13	KEY14	KEY15
0x3C	0xBB	0x16	0x90	0xCF	0x81	0xBE	0x02	0xD7	0x12	0x44	0xA2	0x83	0x6B	0x78	0xA1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x84	0xF8	0x31	0x1C	0x76	0x2B	0xED	0xC9	0x56	0xC3	0x11	0x9C	0x4D	0x3F	0x1D	0x9F
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0xFF	0xF0	0x78	0x71	0xAC	0x88	0x30	0x2F	0x52	0x63	0xA7	0x18	0xA3	0x8B	0x54	0x25
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0x37	0x07	0x0E	0x0A	0xB5	0x97	0xCB	0x90	0x4E	0xCA	0x38	0x82	0x95	0xAF	0xF2	0xAA
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0x16	0x6D	0x4A	0x47	0x39	0x7D	0xBA	0xDC	0xF9	0xC2	0x6F	0x3A	0x81	0x46	0xEA	0x33
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
0x89	0x85	0xB0	0x92	0x35	0xB2	0x60	0xA1	0xE0	0x91	0xB7	0x7F	0x6C	0x41	0x7B	0x58
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
0xA4	0x06	0x4F	0x0F	0x64	0xD7	0xA8	0x61	0xD1	0x53	0x99	0x44	0x03	0x67	0x5F	0x45
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
0x6E	0xC1	0x5A	0x8C	0x04	0xC4	0x93	0x75	0x2C	0x01	0x0C	0xB3	0xA2	0x05	0x65	0xE2
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
0x23	0x21	0x9B	0x9E	0x12	0xE1	0x62	0x74	0xF5	0x80	0x10	0x50	0x1B	0xB1	0xFC	0x3E
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
0xA0	0xD5	0xAD	0x1A	0xFA	0x51	0x79	0x40	0xD9	0x28	0x9A	0x4B	0x55	0xEC	0x17	0xBB
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
0x8A	0x72	0x83	0x3D	0x13	0xDF	0x32	0xDB	0x2E	0x14	0x1E	0xBD	0x20	0xD8	0x00	0xD3
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
0x2A	0xA5	0x94	0x36	0x8F	0xA9	0xB6	0xF7	0xAE	0xC5	0x98	0x1F	0x48	0xD4	0xDE	0xC7
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
0xBC	0x02	0xB9	0x68	0x0B	0xF1	0x70	0x8E	0x87	0x29	0xBF	0xCF	0x7C	0xF3	0xCC	0x5E
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
0x73	0x66	0x57	0xD2	0x77	0x19	0xE6	0x5B	0xCE	0xEB	0x9D	0x5D	0xD0	0x8D	0xA6	0xE5
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
0xEE	0x27	0x6A	0x08	0x7E	0xFD	0x42	0x6B	0xE4	0xE8	0xFB	0xCD	0xBE	0x3B	0xF6	0xB4
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
0xC8	0x09	0x2D	0xE3	0x5C	0x43	0x96	0x0D	0xD6	0x34	0x4C	0x3C	0xDA	0xC0	0xF4	0x15
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
0xFE	0xC6	0x69	0x24	0x49	0x26	0xAB	0x59	0x22	0xDD	0x86	0xEF	0x7A	0xB8	0xE7	0xE9

Below are the post-run memory dumps that show the output array S: