[Exercise 1] Image Convolution in Assembly

A digital image is a two-dimensional array (a matrix) of pixels, where each pixel is characterized by its position in the image (row, $column$) and its value $P_{row,column}$.

Linear image processing operations such as blurring and edge detection are usually performed by *convolving* the entire image with a small square matrix called *kernel*. Depending on the values of kernel elements, different effects may be obtained in the final image. Image convolution for a kernel of size 3×3 is illustrated in Figures 1 and 2. It is performed by taking sub-images of the same dimension as the kernel, convolving them with the kernel, and replacing the value of the center pixel of each sub-image with the corresponding convolution result:

- Figure 1: convolution between a sub-image centered around pixel $P_{1,3}$, and a kernel. The result of the convolution replaces the old value of pixel $P_{1,3}$ in the final image.
- Figure 2: convolution between a sub-image centered around pixel $P_{1,7}$, and the kernel. The result of the convolution replaces the old value of pixel $P_{1,7}$ in the final image. Note that all the pixels beyond the image boundaries (pixels that do not exist in the image) are assumed to have a value equal to zero.

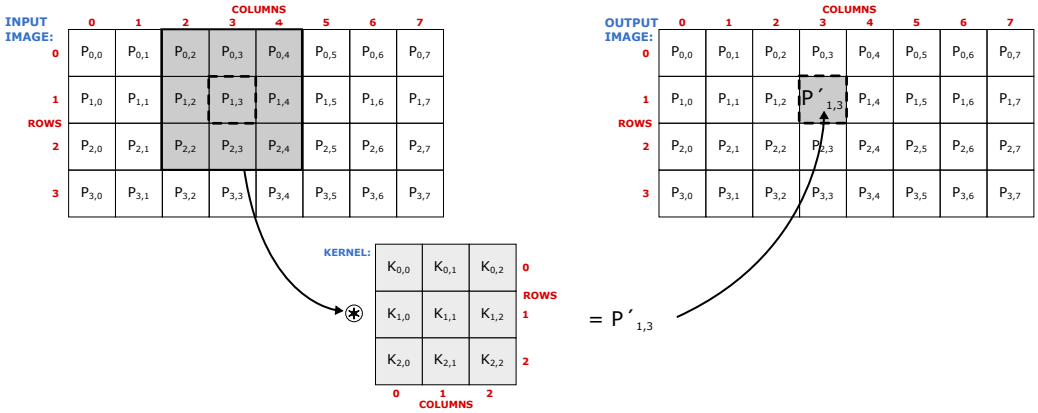


Figure 1: Convolution applied to the image pixel $P_{1,3}$.

The convolution between two square matrices having the same dimensions $n \times n$ is calculated as follows:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,n-1} \\ \dots & \dots & \dots & \dots \\ A_{n-1,0} & A_{n-1,1} & \dots & A_{n-1,n-1} \end{bmatrix} \circledast \begin{bmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,n-1} \\ B_{1,0} & B_{1,1} & \dots & B_{1,n-1} \\ \dots & \dots & \dots & \dots \\ B_{n-1,0} & B_{n-1,1} & \dots & B_{n-1,n-1} \end{bmatrix} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{i,j} \cdot B_{i,j}$$

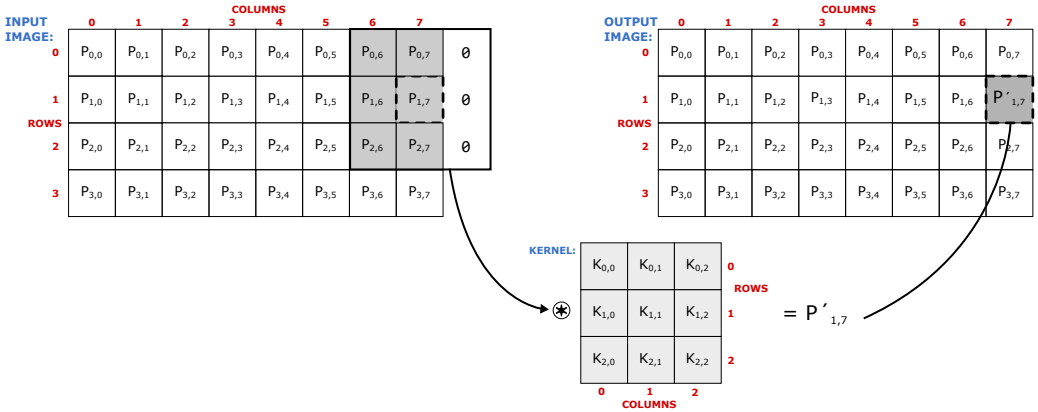
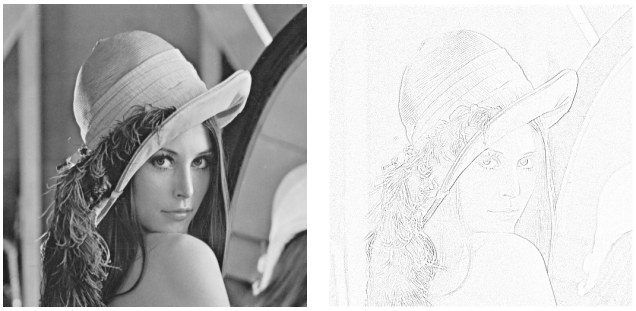


Figure 2: Convolution applied to the image pixel $P_{1,7}$.

For example, convolution between a sub-image and a kernel of size 3×3 equals:

$$\begin{bmatrix} 34 & 54 & 0 \\ 52 & 97 & 0 \\ 94 & 129 & 0 \end{bmatrix} \circledast \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$= (34 \cdot 0 + 54 \cdot 1 + 0 \cdot 0) + [52 \cdot 1 + 97 \cdot (-4) + 0 \cdot 1] + (94 \cdot 0 + 129 \cdot 1 + 0 \cdot 0) = -153.$$



(a) input (b) output

Figure 3: Example of the effects of the convolution: (a) input image and (b) image after convolution with a 3×3 kernel given above.

In this exercise, you are required to write an assembly program that performs convolution between an input image, starting from the memory address `IMAGE_IN`, and a kernel, starting from the memory address `KERNEL`. The resulting image has the same size and memory layout as the input image and should be stored in memory starting from the address `IMAGE_OUT`. Images have `ROWS` rows and `COLS` columns. Kernel matrix has `KERNEL_SIZE` rows and the same number of columns. `KERNEL_SIZE` is an odd number, while `ROWS` and `COLS` are both a power of two.

An image is represented as a two-dimensional array of 8-bit signed integers. Each pixel is identified by its row and its column. The memory layout of the image having `ROWS`

rows and COLS columns is shown in Figure 4. In this figure, values inside rectangles are the offsets of the corresponding image pixels with respect to the beginning of the IMAGE_IN array in memory. For example, the pixel at (row, column) = (0, 0) is stored at address IMAGE_IN, the pixel at (row, column) = (0, 1) at address IMAGE_IN+1, the pixel at (row, column) = (1, 0) at address IMAGE_IN+COLS, etc.

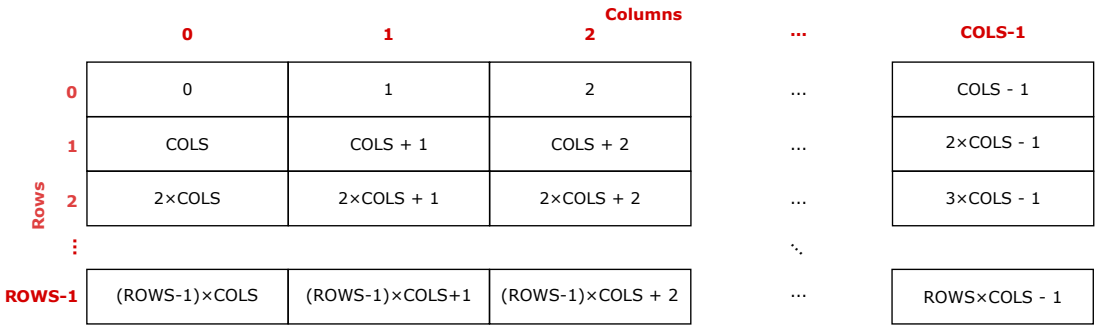


Figure 4: Memory layout of images and kernel.

A kernel is stored following the same data layout in memory. Kernel elements are 8-bit numbers. You may assume that IMAGE_IN, IMAGE_OUT, KERNEL, KERNEL_SIZE, ROWS, and COLS are all defined as constants (.equ) at the beginning of the program, and that the input image and the kernel are already initialized in memory.

Instructions:

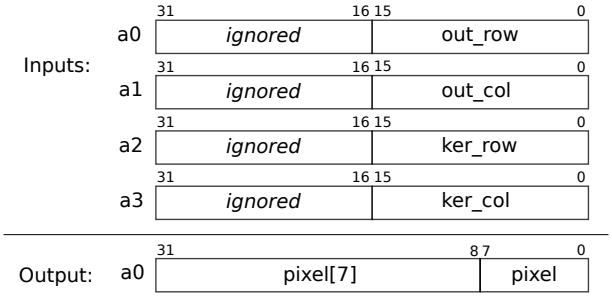
- Depending on the question, you may be allowed to use load-byte and store-byte instructions, or you may have to use only load-word and store-word instructions. This is one more reason to **read each question carefully.**
- You are **NOT** allowed to use any multiplication instruction.
- Your code should conform to the assembly coding conventions.

a) The convolution between a sub-image and the kernel, to produce a value of the output image pixel at the coordinates (out_row, out_col), can be calculated by using the following pseudo-code:

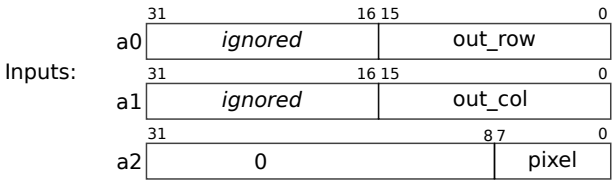
```
conv = 0;
for (ker_row = 0; ker_row < KERNEL_SIZE; ker_row++) {
  for (ker_col = 0; ker_col < KERNEL_SIZE; ker_col++)  {
    in_row = out_row + ker_row - (KERNEL_SIZE - 1) / 2;
    in_col = out_col + ker_col - (KERNEL_SIZE - 1) / 2;
    if pixel coordinates within image boundary:
       in_pixel = in_image(in_row, in_col);
    else
```

```
9             in_pixel = 0;
10          conv = conv +  in_pixel * kernel(ker_row, ker_col);
11        }
12      }
13      out_image(out_row, out_col) = conv;
```

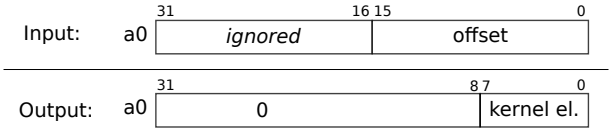
Write the function `get_image_value` that takes the coordinates of the output pixel, `out_row` and `out_col`, of the current kernel coefficient, `ker_row`, `ker_col`, and retrieves the corresponding pixel from the input image (sign-extended). That is, the function should implement the functionality of lines 4–9 in the above pseudo-code. When writing this function, **you are allowed** to use the load-byte instructions. Additionally, you may also assume to have available constants `LOG2ROWS` and `LOG2COLS`, equal to $\log_2(\texttt{ROWS})$ and $\log_2(\texttt{COLS})$, respectively. They should enable computing the memory address of the image pixel without using any multiplication instruction (since their use is not allowed). Inputs and output of `get_image_value` are defined as follows:



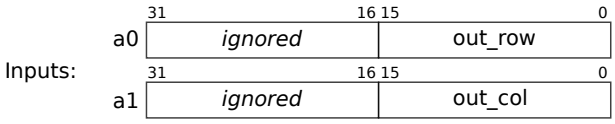
b) Write the function `update_image` that takes the coordinates of the output pixel, `out_row` and `out_col`, the result of the convolution, and updates the corresponding pixel in the output image. In other words, the function implements the pseudo-code line 13. When writing this function, **you are allowed** to use the store-byte instructions. The function has no return value; its inputs are defined as follows:



c) Write the function `get_kernel_value` that takes the memory offset of a kernel element and returns its value. For example, if `offset=2`, the function returns the value of the kernel element at address `KERNEL+2`. Here, **you are NOT allowed** to use the load-byte instructions. Instead, you have to use the load-word instructions. Assume **little-endian** byte order. Note that `KERNEL` **may not be aligned on a word boundary**. The input and the output of `get_kernel_value` are defined as follows:



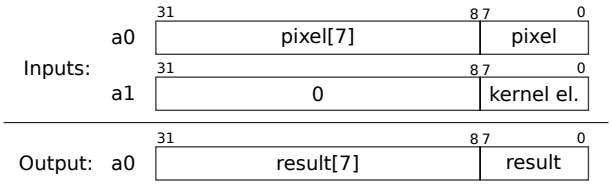
d) Write the function `convolution` that generates the entire output image. You can assume a function `kernel_conv` is available, which performs a full convolution to compute one output pixel. Essentially, this function performs the functionality of the entire pseudo-code. The function `kernel_conv` has no return value; its inputs are defined as follows:

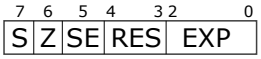


The function `convolution` should call `kernel_conv` for each output pixel; it takes no input arguments and has no return values.

e) Assume now that the kernel elements use the 8-bit encoding described in Figure 5. Such encoding can represent zero or power-of-two positive and negative values. As a consequence, multiplication of an 8-bit signed pixel by a kernel element only requires shifts, sign-inversions, and zeroing.

Implement the function `mult` that performs multiplication of an 8-bit signed integer (image pixel), sign-extended on 32 bits, with an encoded 8-bit kernel element. The result is an 8-bit signed integer; ignore any overflows or underflows. Inputs and output of `mult` are defined as follows:





$$\text{kernel element} = \begin{cases} (-1)^S \cdot 2^{(-1)^{SE} \cdot \underline{\text{EXP}}} & \text{if } Z = 0 \\ 0 & \text{if } Z = 1 \end{cases}$$

S	sign: 0 = positive, 1 = negative
Z	zero: 0 = non-zero, 1 = zero
SE	sign of the exponent: 0 = positive, 1 = negative
RES	reserved (to be ignored)
EXP	exponent

Figure 5: Encoding used for the kernel elements.

[Solution 1]

a)

```
#get_image_value
#a0 = out_row
#a1 = out_col
#a2 = ker_row
#a3 = ker_col
#a0 = IMAGE_IN[out_row + ker_row - (KERNEL_SIZE - 1) / 2]
#              [out_col + ker_col - (KERNEL_SIZE - 1) / 2],
#     zero if outside image size
get_image_value:

#Zero the ignored bits
li t0, 0xFFFF
and a0, a0, t0
and a1, a1, t0
and a2, a2, t0
and a3, a3, t0

li   t0, KERNEL_SIZE
addi t0, t0, -1
#t0 = (KERNEL_SIZE - 1) / 2
srli t0, t0, 1
#a0 = out_row + ker_row
add a0, a0, a2
#a1 = out_col + ker_col
add a1, a1, a3
#a0 = out_row + ker_row - (KERNEL_SIZE - 1) / 2 -> in_row
sub a0, a0, t0
#a1 = out_col + ker_col - (KERNEL_SIZE - 1) / 2 -> in_col
sub a1, a1, t0
#t6 = 0 (default value)
add t6, zero, zero

#Check if a0 and a1 are within the image bounds
blt a0, zero, get_image_value_end
blt a1, zero, get_image_value_end

li t0, ROWS
bge a0, t0, get_image_value_end

```

```
li t0, COLS
bge a1, t0, get_image_value_end

#If yes, read the pixel
#a0 = in_row << LOG2COLS = in_row * COLS
slli a0, a0, LOG2COLS
#a1 = in_row * COLS + in_col
add a1, a1, a0
lb t6, IMAGE_IN(a1)
#sign extension
andi t0, t6, 128
beq t0, zero, get_image_value_end

addi t0, zero, -256
or t6, t6, t0

get_image_value_end:
addi a0, t6, 0
ret
```

b)

```
#update_image
#a0 = out_row
#a1 = out_col
#a2 = value to write to IMAGE_OUT[out_row][out_col]
update_image:

#Zero the ignored bits
li t0, 0xFFFF
and a0, a0, t0
and a1, a1, t0

slli a0, a0, LOG2COLS    #a0 = out_row << LOG2COLS = out_row * COLS
add a0, a0, a1           #a0 = out_row * COLS + out_col
#   -> offset
sb a2, IMAGE_OUT(a0)
ret
```

c)

```
#get_kernel_value
#a0 = offset
#a0 = KERNEL[offset]
get_kernel_value:

#Zero the ignored bits
li t0, 0xFFFF
and a0, a0, t0

addi a0, a0, KERNEL
lw t0, 0(a0)              #t0 = word containing KERNEL[offset]
andi a0, a0, 3            #a0 = byte_addr_offset within 4-byte word
slli a0, a0, 3            #a0 = byte_addr_offset * 8 = shift
srl t0, t0, a0            #t0 = t0 >> shift
andi t0, t0, 0xFF         #clear all bits except for the LSByte
ret
```

d)

```
#convolution
#Iterate over ROWS rows and COLS columns, starting from IMAGE_IN
#and writing to IMAGE_OUT.
#No arguments, no return values
convolution:
addi sp, sp, -12
sw ra, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
add s1, zero, zero        #s1 = out_row

convolution_row_loop_begin:
li t0, ROWS
bgeu s1, t0, convolution_row_loop_end

add s2, zero, zero        #s2 = out_col

convolution_col_loop_begin:
li t0, COLS
bgeu s2, t0, convolution_col_loop_end

add a0, zero, s1
add a1, zero, s2
jal ra, kernel_conv
```

```
addi s2, s2, 1
j convolution_col_loop_begin

convolution_col_loop_end:
addi s1, s1, 1
j convolution_row_loop_begin

convolution_row_loop_end:
lw ra, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
addi sp, sp, 12
ret

kernel_conv:
#Not defined
ret
```

e)

```
#mult
#a0 = image pixel (8-bit signed)
#a1 = kernel value (8-bit custom encoding)
#a0 = image pixel * kernel value (8-bit signed)
mult:
andi t0, a1, 64          #Z
bne t0, zero, mult_clear

andi t0, a1, 32          #SE
andi t1, a1, 7           #EXP
bne t0, zero, mult_right

sll t6, a0, t1
j mult_flip

mult_right:
sra t6, a0, t1

mult_flip:
andi t0, a1, 128         #S
beq t0, zero, mult_end

sub t6, zero, t6
j mult_end
```

```
25
26 mult_clear:
27 add t6, zero, zero
28
29 mult_end:
30 addi a0, t6, 0
31 ret
```

[Exercise 2] Encryption in Assembly

Consider the encryption algorithm shown in Figure 6. It takes a 32-bit input word W_{IN} and an array of keys K_i, $0 \leq i < N_K$, to produce a 32-bit encrypted word W_{OUT} after N_K iterations.

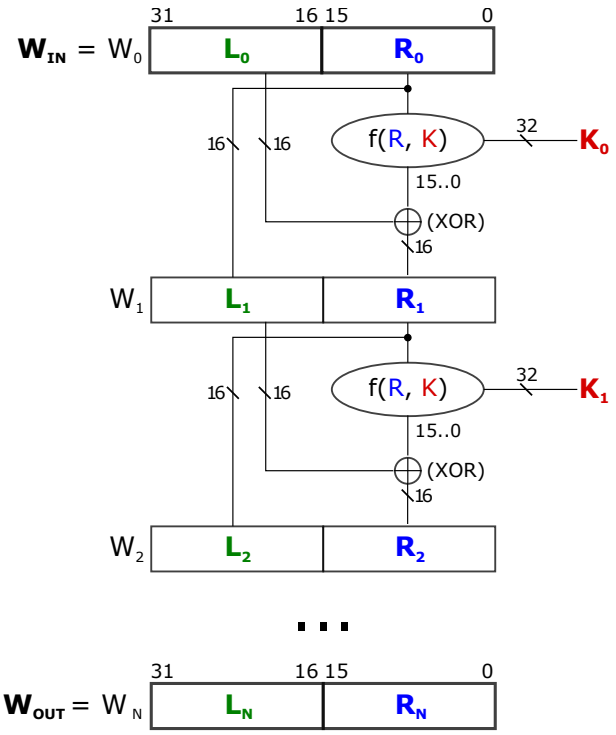


Figure 6: Encryption of one 32-bit word. L_i and R_i denote the upper 16 bits (the left part) and the lower 16 bits (the right part) of the 32-bit word W_i. The input word W_{IN} is considered encrypted after N_K cycles of computation, where N_K is the number of 32-bit keys. The result W_{OUT} is therefore equal to W_N. XOR stands for exclusive-or.

The function $f(R, K)$ is defined as follows:

$$f(R, K) = (\texttt{0x0000} \ \& \ R) + (\texttt{0x0000} \ \& \ K_{23..16} \ \& \ K_{7..0})$$

where $\&$ denotes the binary concatenation operator (as in Verilog) and $K_{m..n}$ selects bits m downto n of K (equivalent to `K[m:n]` in Verilog). Note that the input R is on 16 bits, K is on 32 bits, and the function returns a 32-bit value.

For example, given N_K = `4`, K_0 = `0x33221100`, K_1 = `0x77665544`, K_2 = `0xBBAA9988`, and K_3 = `0xFFEEDDCC`, and for W_{IN} = `0x12345678`, the algorithm proceeds as follows:

i	W_{i-1}	L_{i-1}	R_{i-1}	K_{i-1}	$f(R,K)$	$W_i = R_{i-1}$ & $(f(R,K)_{15..0}$ XOR $L_{i-1})$
1	0x12345678	0x1234	0x5678	0x33221100	0x5678 + 0x2200 = 0x00007878	0x5678 & 0x6A4C = 0x56786A4C
2	0x56786A4C	0x5678	0x6A4C	0x77665544	0x6A4C + 0x6644 = 0x0000D090	0x6A4C & 0x86E8 = 0x6A4C86E8
3	0x6A4C86E8	0x6A4C	0x86E8	0xBBAA9988	0x86E8 + 0xAA88 = 0x00013170	0x86E8 & 0x5B3C = 0x86E85B3C
4	0x86E85B3C	0x86E8	0x5B3C	0xFFEEDDCC	0x5B3C + 0xEECC = 0x00014A08	0x5B3C & 0xCCE0 = 0x5B3CCCE0

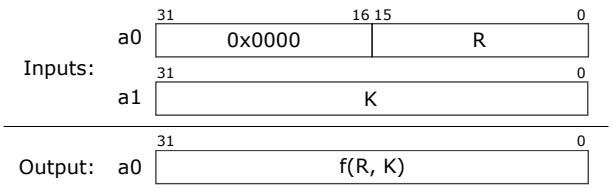
and eventually returns $W_{OUT} = W_4 =$ 0x5B3CCCE0.

In this exercise, you are to write an assembly program that reads an array of 8-bit ASCII chars in groups of four, i.e., **word by word**, encrypts every loaded word, and stores the result to memory, which is **byte addressed**. You may assume that the number of chars in the array is a multiple of four.

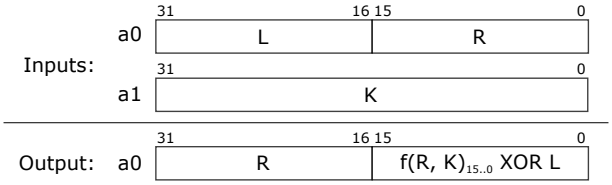
Instructions:

- To access the memory you are allowed to use **only load-word and store-word** instructions.
- Your code should conform to the assembly coding conventions.

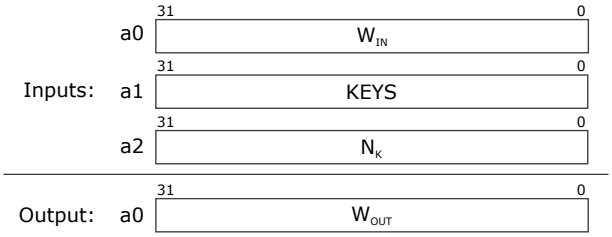
a) Write the function FRK that computes $f(R,K)$. Inputs and output of FRK are defined as follows:



b) Write the function `ENC_STEP` that calls `FRK` and performs one iteration of the encryption algorithm. Inputs and output of `ENC_STEP` are defined as follows:



c) Write the function `ENC_WORD` that encrypts a single 32-bit word by calling `ENC_STEP` N_K times (N_K equals the number of 32-bit keys K). Inputs and output of `ENC_WORD` are defined as follows:



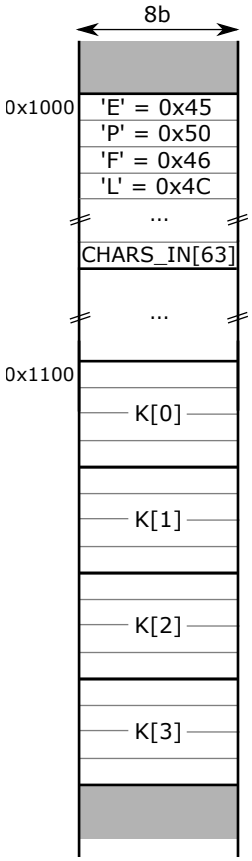
d) Write the `main` function of the encryption program. The program encrypts **word by word** of an array of `N_C` 8-bit ASCII chars that starts at the address `CHARS_IN`, where `CHARS_IN` is word-aligned. The program writes the encrypted words in an output array starting from address `CHARS_OUT`, where `CHARS_OUT` is also word-aligned. The `N_K` 32-bit keys are in memory starting from the address `KEYS`, which is also word-aligned. You may assume that `CHARS_IN`, `N_C`, `CHARS_OUT`, `KEYS`, and `N_K` are all defined as constants at the beginning of the program, and that the input arrays of chars and keys are already initialized in memory.

For example, given the following preamble at the beginning of the code:

```
# number of chars in the input array
.equ N_C        64
# number of keys in the array of keys
.equ N_K        4

.data
# starting address of the array of input chars
CHARS_IN:
 ...
# starting address of the array of 32-bit keys
KEYS:
 ...
# starting address of the array of encrypted chars
CHARS_OUT:
 ...
```

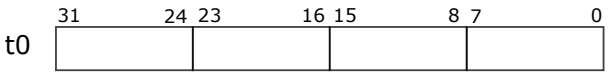
The figure below illustrates a possible state of the memory just before reaching the `main` function:



e) Assuming the memory state shown above and a **big-endian** processor, what is the content of register `t0` after executing the following instruction in RISC-V assembly:

```
1 lw t0, 0x1000(zero)
```

You may use ASCII chars to represent values of individual bytes. Here is an example of the drawing that you should do and fill in:



[Solution 2] Encryption in Assembly

```
# number of chars in the input array
.equ N_C        4
# number of keys in the array of keys
.equ N_K        4

.data
# starting address of the array of input chars
CHARS_IN:
.word 0x12345678
# starting address of the array of 32-bit keys
KEYS:
.word 0x33221100
.word 0x77665544
.word 0xBBAA9988
.word 0xFFEEDDCC
# starting address of the array of encrypted chars
CHARS_OUT:
.word 0x0, 0x0, 0x0, 0x0

.text
encryption:
    jal     ra, main
    li      a0, 10              # triggers an ecall to end
    ecall                       # (exit system call)

## (a)

FRK:
    andi    t0, a1, 0xFF        # t0 = K(7 .. 0)
    srli    t1, a1, 16          # t1 = K(31 .. 16)
    andi    t1, t1, 0xFF        # t1 = K(23 .. 16)
    slli    t1, t1, 8           # t1 = K(23 .. 16) && "00000000"
    or      t0, t0, t1          # t0 = K(23 .. 16) && K(7 .. 0)
    add     a0, a0, t0          # v0 = R + K(23 .. 16) && K(7 .. 0)
    ret

## (b)

ENC_STEP:
    # push on the stack
    addi    sp, sp, -12
```

```
42      sw      ra, 0(sp)
43      sw      s0, 4(sp)
44      sw      s1, 8(sp)
45
46      li      t0, 0xFFFF
47      and     s0, a0, t0          # s0 = R
48      srli    s1, a0, 16          # s1 = L
49
50      addi    a0, s0, 0           # a0 = R
51      addi    a1, a1, 0           # a1 = K
52
53      jal     ra, FRK             # a0 = f(R,K)
54
55      xor     a0, s1, a0          # a0 = f(R,K) xor L
56      li      t0, 0xFFFF
57      and     a0, a0, t0          # a0 = (f(R,K) xor L) (15 .. 0)
58      slli    s0, s0, 16          # s0 = R && "0x0000"
59      or      a0, a0, s0          # a0 = R && (f(R,K) xor L) (15 .. 0)
60
61      # pop from the stack
62      lw      ra, 0(sp)
63      lw      s0, 4(sp)
64      lw      s1, 8(sp)
65      addi    sp, sp, 12
66
67      ret
68
69  ## (c)
70
71  ENC_WORD:
72      # push the stack
73      addi    sp, sp, -16
74      sw      ra, 0(sp)
75      sw      s0, 4(sp)
76      sw      s1, 8(sp)
77      sw      s2, 12(sp)
78
79      addi    s0, a1, 0           # s0 = KEYS
80      add     s1, zero, a2        # s1 = N_K
81      addi    s2, zero, 0         # s2 = 0
82
83  ENC_WORD_loop:
84      lw      a1, 0(s0)           # a1 = K = MEMORY[s0]
85
```

```
    jal     ra, ENC_STEP          # a0 = enc_step(a0)

    addi    s2, s2, 1             # s2 = s2 + 1
    beq     s2, s1, ENC_WORD_end  # if (s2 == N_K) goto ENC_WORD_end

    addi    s0, s0, 4             # s0 = s0 + 4
    j       ENC_WORD_loop

ENC_WORD_end:
    # pop the stack
    lw      ra, 0(sp)
    lw      s0, 4(sp)
    lw      s1, 8(sp)
    lw      s2, 12(sp)
    addi    sp, sp, 16

    ret

## (d)

main:
  #push the stack
  addi    sp, sp, -20
  sw      ra, 0(sp)
  sw      s0, 4(sp)
  sw      s1, 8(sp)
  sw      s2, 12(sp)
  sw      s3, 16(sp)

  # s0 = N_C / 4 (number of 32-bit words)
  addi    s0, zero, N_C
  srli    s0, s0, 2

  addi    s1, zero, 0           # s1 = 0
  la      s2, KEYS              # s2 = KEYS
  li      s3, N_K               # s3 = N_K

main_loop:
  beq     s0, zero, main_end    # if (s0 == 0) goto main_end
  la      t2, CHARS_IN
  add     t2, t2, s1

  lw      a0, 0(t2)             # a0 = CHARS_IN[s1]
  addi    a1, zero, s2
```

```
130   addi    a2, zero, s3
131
132   jal     ra, ENC_WORD
133
134   la      t2, CHARS_OUT
135   add     t2, t2, s1
136   sw      a0, 0(t2)             # CHARS_OUT[s1] = a0
137
138   addi    s0, s0, -1            # s0 = s0 - 1
139   addi    s1, s1, 4             # s1 = s1 + 4
140
141   j       main_loop
142
143 main_end:
144   #pop the stack
145   lw      ra, 0(sp)
146   lw      s0, 4(sp)
147   lw      s1, 8(sp)
148   lw      s2, 12(sp)
149   lw      s3, 16(sp)
150   addi    sp, sp, 20
151
152   ret
```

e)