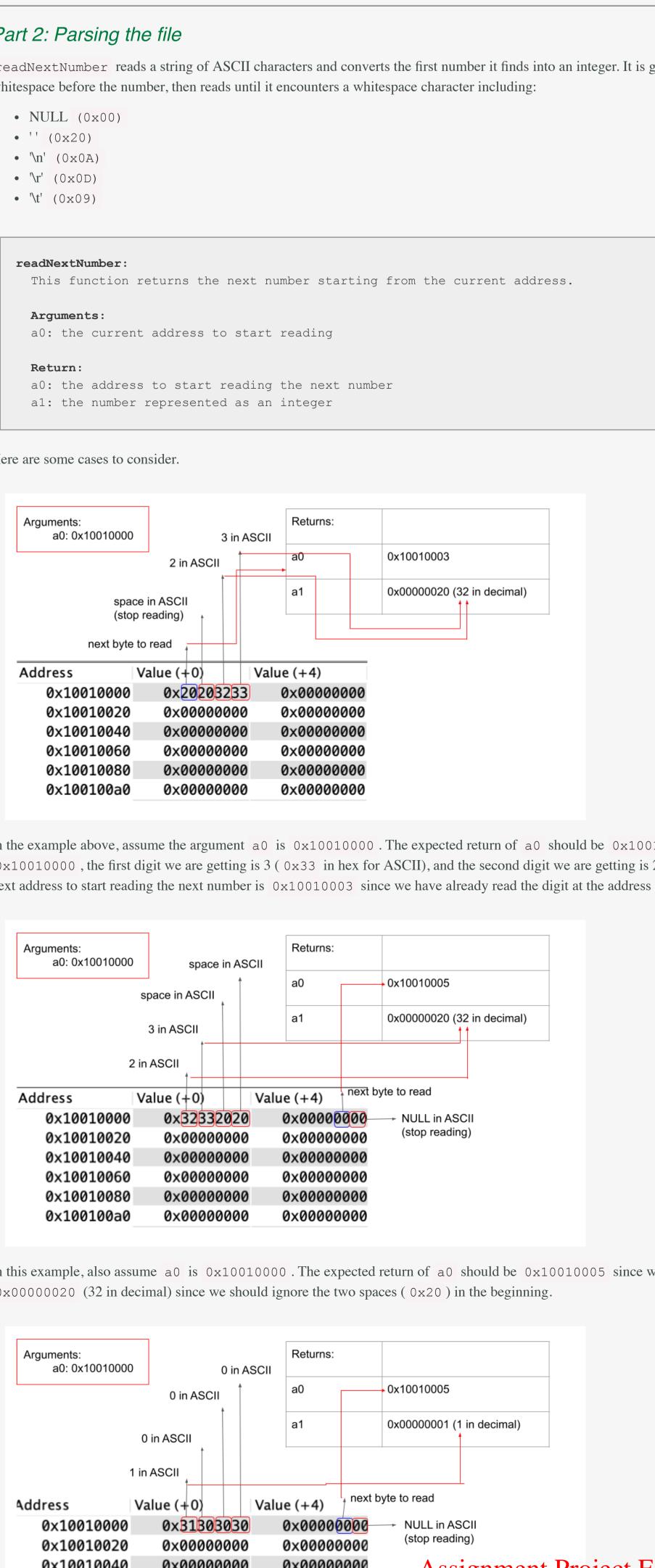
UNIVERSITY OF ALBERTA CMPUT 229 - Computer Organization and Architecture I Lab 3: Image Blurring Introduction Image blurring is a common technique used to hide or obscure an image. It distorts the details of an image, making it less clear. A shape can be identified due to its edges. To blur it, we can reduce the clarity of an edge and make the transition from one pixel to another more smooth. Background **Box Blur** This lab uses box blur, also known as a box linear filter. It will apply a filter in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. A 3 by 3 box blur ("radius 1") can be represented with the following matrix: The radius can be calculated as matrixSize / 2 (floor division). For example, a 5 by 5 box blur has a radius of 2. PPM File This lab uses the PPM image format because in this format RGB values are stored as ASCII characters, which makes it very easy to manipulate. To view a PPM image on your machine, try to use this link. The picture below explain how a PPM file is structured if we open it as a text file. test.ppm **RGB Values** column #, row # 0 0 0 255 10 99 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 60 0 255 11 255 0 0 0 255 0 0 0 255 255 255 255 0 0 0 255 0 0 0 255 0 0 0 255 The "P3" identifier on the first line, and the value of 255 on the third line are required. A PPM file begins with an identifier has the following format: • An identifier, also known as "magic number", on the first line: • identifier = "P3" for an ASCII PPM file; • identifier = "P2" fo an ASCII PGM file. • image size: number of columns followed by number of rows. • a max value (PPM\_MAX), which indicates the maximum value of any pixel component (R, G, B, or gray value) may have. This value may be any positive integer, although values of 15 and 255 are quite typical. • color values for all the column × row pixels. The values are ordered by rows from top to bottom. Within each row, the pixels are listed left to right. For a PPM image, each pixel value is listed as the values of its three components, Red, Green and Blue. These values are represented as ASCII characters in the memory in RISC-V. A whitespace (i.e., typically a space or newline character) must separate these integers. The file can have any number of values on each line. For example, it is possible that the file has all the pixel values (R, G and B, separated by spaces) on a single line. Assignment The goal of this lab is to write a program in RISC-V that will blur an image provided. You must write the following functions digitToAscii, copy, asciiToDigit, readNextNumber pixelKernelMul, storeNumber, blurImage with the specifications discussed later. asciiToDigit readNextNumber copy blurlmage pixelKernelMul storeNumber digitToAscii The code provided in common.s operates as follows: 1. Read each color value from the image file as a string of ASCII characters. For each value read: 1. Call readNextNumber which uses asciiToDigit to convert each ASCII string into an integer value. 2. Store each integer value in memory. 2. Call copy to make an in-memory copy of the values. This copy will be used to computed the color values for the blurred image. 3. Call blurImage which uses pixelKernelMul to blur the image. 4. Call storeNumber which uses digitToAscii to convert the blurred image back to ASCII characters. Part 1: Helper functions digitToAscii: This function returns the ASCII value of the digit. Arguments: a0: a single digit represented as an integer, between 0 and 9 Return: a0: the ASCII value of the digit, between 48 (0x30) and 57 (0x39) The picture below is the ASCII table. For the function above, the input will be restricted to digits (0-9). **ASCII TABLE** |Decimal Hex Char |Decimal Hex Char | Decimal Hex Char Decimal Hex Char 32 [NULL] [SPACE] 64 96 97 [START OF HEADING] 21 65 41 1 1 33 61 22 66 98 2 2 [START OF TEXT] 34 42 62 b 3 3 35 23 67 43 C 99 [END OF TEXT] 63 C 4 36 24 68 44 D 100 [END OF TRANSMISSION] 64 d 37 5 5 [ENQUIRY] 25 69 45 Ε 101 65 e 6 38 26 & 70 46 F 102 [ACKNOWLEDGE] 66 f [BELL] 39 27 71 47 G 103 7 67 8 40 28 72 48 н 104 [BACKSPACE] 68 9 9 [HORIZONTAL TAB] 41 29 73 49 105 69 42 2A 74 10 Α [LINE FEED] 4A 106 6A 11 В 43 2B 75 4B K 107 [VERTICAL TAB] 6B k 12 C [FORM FEED] 2C 76 4C 108 6C 45 2D 77 4D 13 D M 109 6D [CARRIAGE RETURN] m 14 Е 46 2E 78 4E N 110 [SHIFT OUT] 6E n 15 F 47 2F 79 4F 0 111 6F [SHIFT IN] 0 16 10 [DATA LINK ESCAPE] 48 30 0 80 50 112 70 31 49 1 81 51 Q 113 17 11 [DEVICE CONTROL 1] 71 32 18 12 [DEVICE CONTROL 2] 50 2 82 52 R 114 72 19 13 51 33 83 53 S 115 73 [DEVICE CONTROL 3] 52 34 54 20 14 4 84 Т 116 74 [DEVICE CONTROL 4] 21 15 [NEGATIVE ACKNOWLEDGE] 53 35 5 85 55 U 117 75 36 6 86 56 V 22 16 54 118 76 [SYNCHRONOUS IDLE] 37 7 87 23 17 [END OF TRANS. BLOCK] 55 57 W 119 77 56 38 58 24 18 [CANCEL] 88 120 78 X 25 19 57 39 9 89 59 Y 121 79 [END OF MEDIUM] У 26 1A [SUBSTITUTE] 58 3A 90 5A Z 122 7A 27 1B [ESCAPE] 59 3B 91 5B 123 7B 28 1C 60 3C 92 5C 124 [FILE SEPARATOR] < 7C 29 3D 93 1D 5D 125 7D [GROUP SEPARATOR] 30 1E [RECORD SEPARATOR] 62 3E > 94 5E 126 7E 63 31 1F 3F 95 [UNIT SEPARATOR] 5F 127 7F [DEL] copy: This function copies all the RGB values to another address. Arguments: a0: address of the start of RGB values (where to copy from) al: address of the start of where to copy to a2: length in words asciiToDigit: This function returns the digit for the given ASCII value Arguments: a0: the ASCII value of the digit, between 48 (0x30) and 57 (0x39) Return: a0: a single digit represented as an integer, between 0 and 9 Part 2: Parsing the file readNextNumber reads a string of ASCII characters and converts the first number it finds into an integer. It is guaranteed to only have ASCII numbers between 0 and 255. The function skips any whitespace before the number, then reads until it encounters a whitespace character including: • NULL (0x00) • '' (0x20) • ' n' (0x0A)•  $'\r'$  (0x0D) • '\t' (0x09) readNextNumber: This function returns the next number starting from the current address. Arguments: a0: the current address to start reading Return: a0: the address to start reading the next number al: the number represented as an integer Here are some cases to consider. Returns: Arguments: a0: 0x10010000 3 in ASCII 0x10010003 2 in ASCII а1 0x00000020 (32 in decimal) space in ASCII (stop reading) next byte to read Value (+0) Value (+4) Address 0×10010000 0x20203233 0x00000000 0x10010020 0x00000000 0x00000000 0x10010040 0x00000000 0x00000000 0x10010060 0x00000000 0x00000000 0x10010080 0x00000000 0x00000000 0x100100a0 0x00000000 0×00000000 In the example above, assume the argument a0 is 0x10010000. The expected return of a0 should be 0x10010003 and a1 should be 0x00000020 since starting at the memory address of



0x10010000, the first digit we are getting is 3 (0x33 in hex for ASCII), and the second digit we are getting is 2 (0x32 in hex for ASCII). Therefore, the result of a1 is 0x20 (32 in decimal). And the next address to start reading the next number is 0x10010003 since we have already read the digit at the address of 0x10010000 (digit 3), 0x10010001, (digit 2) and 0x10010002 (space). In this example, also assume a0 is 0x10010000. The expected return of a0 should be 0x10010005 since we have processed all the addresses before that. However, the return of a1 should also be  $0 \times 00000020$  (32 in decimal) since we should ignore the two spaces ( $0 \times 20$ ) in the beginning. 0×00000000 Assignment Project Exam Help 0x10010040 0x00000000 0×10010060 0x00000000 0×00000000 0x10010080 0x00000000 0×00000000 https://tutorcs.com 0x100100a0 0×00000000 0×00000000 In this example, the return of a0 should be 0x10010005 and a1 should be 0x00000001 since we are ignoring the leading zeros (0x30 in hex for ASCII). Returns: Arguments: a0: 0x10010000 0 in ASCII a0 0x10010005 0 in ASCII а1 0x00000000 (0 in decimal) 0 in ASCII 0 in ASCII next byte to read Value (+0) Value (+4) ١ddress 0×0000000 **NULL in ASCII** 0x10010000 0x30303030 (stop reading) 0×10010020 0x00000000 0x00000000 0×10010040 0x00000000 0x0000000 0×10010060 0×00000000 0×00000000 0x10010080 0x00000000 0x00000000 0x100100a0 0x00000000 0×00000000 Finally, in this example, the return of a0 should also be 0x10010005 and a1 should be 0x00000000. storeNumber: This function converts an integer value between 0 and 255 to ASCII and then stores their ASCII to the address. Store the leftmost digit first, consider using the function digitToAscii.

Arguments: a0: the number represented as an integer al: address for the number to be stored Return: a0: the next address that's available to be stored Assume the argument a0 is 0x000000ff (255 in decimal) and a1 is 0x10010000. After calling the function, we should see the number is stored in the memory as shown in the diagram below. Arguments: Return: a0: 0x000000ff (255 in decimal) a0: 0x10010003 a1: 0x10010000 Address Value (+0) Value (+4) 0x10010000 0x00353532 0x00000000 0x00000000 0x10010020 0×00000000 0x00000000 0x10010040 0×00000000 0x10010060 0×00000000 0×00000000 0×10010080 0x00000000 0x00000000 0x100100a0 0×00000000 0×00000000 Equivalently, if we translate the hexadecimal to ASCII, Address Value (+0) Value (+4) 0x10010000 \0 5 2 5 \0 \0 \0 \0 0x10010020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 0x10010040 0x10010060 \0 \0 \0 \0 \0 \0 \0 \0 0x10010080 \0 \0 \0 \0 \0 \0 \0 \0 0x100100a0 \0 \0 \0 \0 \0 \0 \0 \0 And the return of a0 should be  $0 \times 10010003$  since that's the address for the next number to be stored. Part 3: Blurring pixelKernelMul: This function calculates the average for each of the R, G, B values and stores the value. Arguments: a0: address of the start of RGB values, store the calculated RGB values in this region (a0 is the base address) al: address of the start of the copy of RGB values a2: row # of the current pixel to blur a3: col # of the current pixel to blur a4: total row (may not be used) a5: total col After parsing the image file, each word in RISC-V memory represents a pixel value. Each pixel value contains 4 bytes, including R, G, B values and a placeholder (Null). Note that each of the R, G, B values is stored as a 1-byte unsigned integer in memory since they can only be from 0 to 255. R value (18 in dec) G value (0)

B value (255) Pixel 2: Pixel 3: Pixel 1: Address Value (+0) Value (+4) Value (+8) Value (+c) 0x00ff0012 0x001100ff 0×10010000 0x00ff00ff 0x00ff0012 0×10010020 0x00ff00ff 0x00ff0012 0x001100ff 0x00ff00ff 0x001100ff 0x00ff00ff 0x00ff0012 0x001100ff 0×10010040 0×10010060 0x00ff0012 0x001100ff 0x00ff00ff 0x00ff0012 0×10010080 0x00ff00ff 0x00ff0012 0x001100ff 0x00ff00ff 0x001100ff 0x100100a0 0x00ff00ff 0x001100ff 0x00ff0012 Null (placeholder) +---+ | 0 | 1 | 2 | 3 | 4 | 5 | ... | 255 | +---+ | R | G | B | Nul | R | G | ... | Nul | +---+---+ <----> 1 pixel For the simplicity of the lab, we are only going to blur the center of the image and not deal with the corners and edges. We are using the kernel with a radius of 3. The matrix will be a 7 (calculated from  $2 \times 3$ + 1) by 7 matrix (all filled by 1) and then multiply by  $\frac{1}{49}$ . It looks like the picture below. We are going to apply the filter for each of the R, G, B values separately for all pixels (except the ones on the edges and corners). The video below explains the calculation process. We are going to take all the 49 surrounding pixels and then calculate their average. Do NOT modify the memory given by a1. It is a copy of the original image. Store the result of the blur in a0. Copy:

blurImage: This function blurs the image using pixelKernelMul on all possible pixels. It will run pixelKernelMul on all pixels except for the ones on the edges and corners. The C code is provided below. Arguments: a0: kernel size al: total rows

a2: total columns a3: address of the start of RGB values, store the calculated RGB values in this region (a0 is the base address) a4: address of the start of the copy of RGB values C code for blurImage: const int kernel radius = 3; for (int i = kernel radius; i < row - kernel radius; i++) {</pre> for (int j = kernel radius; j < col - kernel radius; j++) { pixelKernelMul(...);x

Writing a Solution Notice the directive .include "common.s" in the solution template file provided. This directive will cause RARS to execute the code in the file common.s before executing any code in the solution. You must follow the template exactly in blur.s by completing each function. You can make any other helper functions, as long as they do not share a name with any labels already used in the common.s file. Read the code in the common.s file to understand how the whole program works. In RISC-V assembly every function must have a return statement, this return statement is the instruction jr ra. You must include this instruction at the end of your function. Return values must be stored in the argument registers. Testing your Lab Sample test cases are in Code/tests/. The correct outputs for these tests are in the files with the same name but .correct.out extension.

When testing, provide the path to test file at program arguments in the RARS simulator. Alternatively, test from the terminal with rars LABFILE pa TESTFILE where TESTFILE is the path to the file that contains the test case that you want to test and LABFILE is the path to blur.s. After running the program, it will check the correctness of the functions. The check does not guarantee that the solution will receive full marks for the functions, you still need to do your own testings. The provided tests do not check the correctness of blurImage. This function must be checked manually as discussed in the next 2 paragraphs. The program also generates an output file (blurred image) with the same name as the test file but with .out extension. The Linux/UNIX command diff can be used to compare the outputs. For instance, the command diff IMAGE.out IMAGE.out, where IMAGE.out is the path to the file that contains your image and IMAGE.correct.out is the path to the correct output image, will print the differences between the two outputs. If diff does not print anything, then the two outputs are identical. You are encouraged to test your functions individually. Do NOT put anything in .data section in blur.s. The test case test2.ppm may take up to 3 minutes to run. Thus, only run this test case after the solution successfully executes the smaller test cases.

This lab is supported in CheckMyLab. To get started, navigate to the Image Blurring lab in CheckMyLab found in the dashboard. From there, students can upload their test cases in the My test cases table

Assumptions and Notes

anything in the file blur.s.

Check My Lab

Link to CheckMyLab

Resources

Marking Guide

• 20% for program style

Submission

• 80% for correct program functionality

• Here is the mark sheet used for grading

• **Do not** add a main label to this file.

to ensure your solution is submitted.

• Do not modify the line .include "common.s".

• Keep the file blur.s in the Code folder of the git repository.

doesn't run properly, use beq zero instead.

program. You can use this for debugging.

• Slides used for in-class introduction of the lab (.pdf)(.pptx)

• 5% for correctly implementing digitToAscii.

• 5% for correctly implementing asciiToDigit.

• 15% for correctly implementing storeNumber.

• 15% for correctly implementing blurImage.

• 20% for correctly implementing readNextNumber.

• 15% for correctly implementing pixelKernelMul.

There is a single file to be submitted for this lab. blur.s should contain the code for your solution.

• Push your repository to GitHub before the deadline. Just committing will not upload your code. Check online

• 5% for correctly implementing copy.

• We will not test your lab with a saturation number greater than 255.

• We will only test your lab with the images that has the exact same format as the ones under Code/tests/.

• Some students reported issues using the beqz instruction in RARS. If you find your code is buggy or

• If you use system calls or ebreak for debugging, remove them before submitting the lab. Do not print

• The pseudo instruction ebreak lets you set breakpoints in your code. When your debugger reaches the

instruction ebreak, it will stop before the next instruction, allowing you to inspect the state of your

(see below). Additionally, students can also upload their 'blur.s' file in the My solutions table, which will then be tested against all other valid test cases.