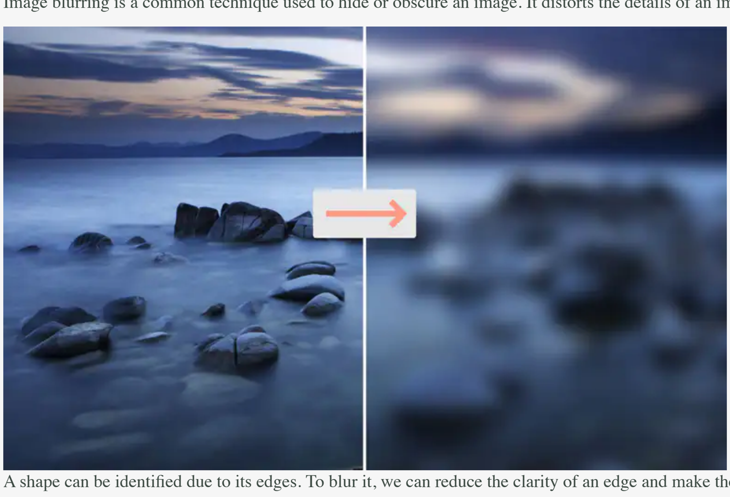
CMPUT 229 - Computer Organization and Architecture I

Lab 3: Image Blurring

Introduction

[illegible]



[illegible]

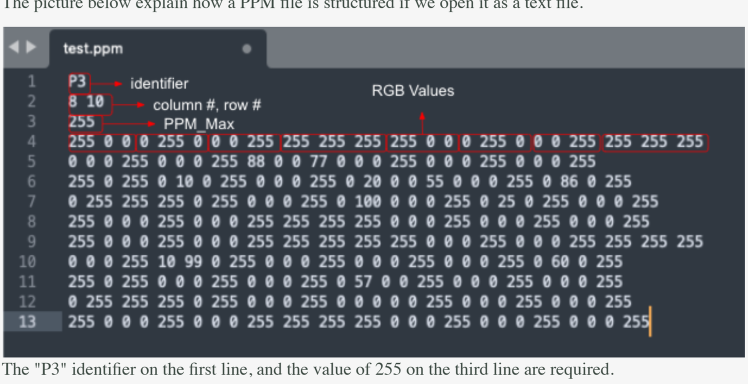
Background

Box Blur

[illegible]

PPM File

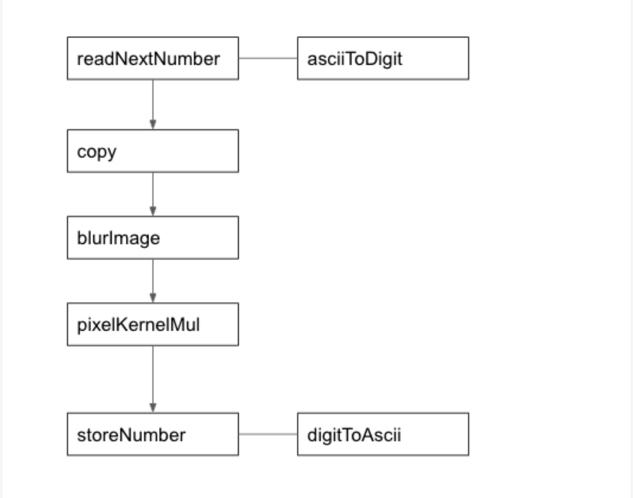
[illegible]



[illegible]

Assignment

[illegible]

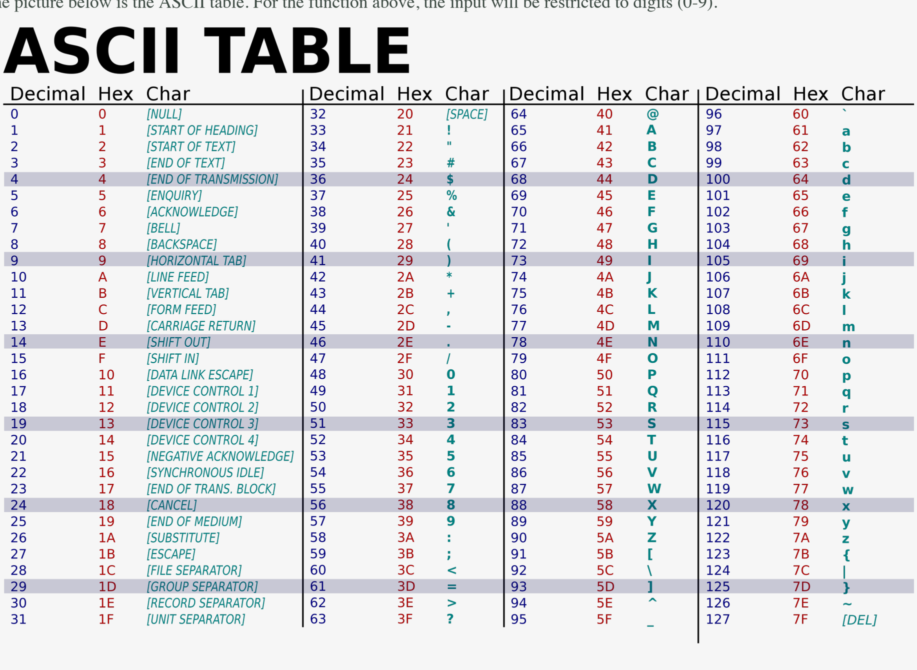


[illegible]

Part 1: Helper functions

```
digitToAscii:
  [illegible]

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
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	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)
  a1: 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
```

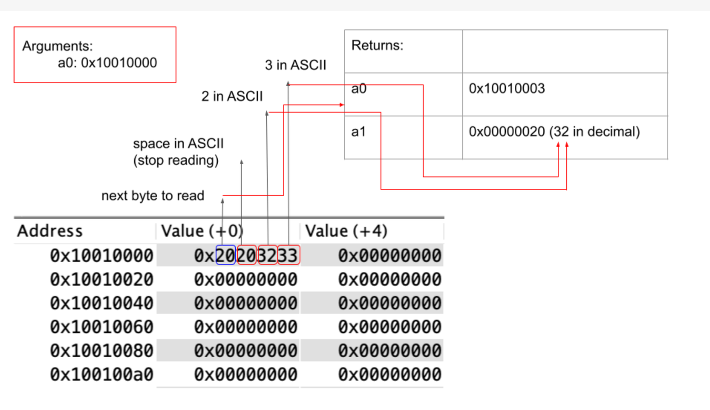
Part 2: Parsing the file

`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:

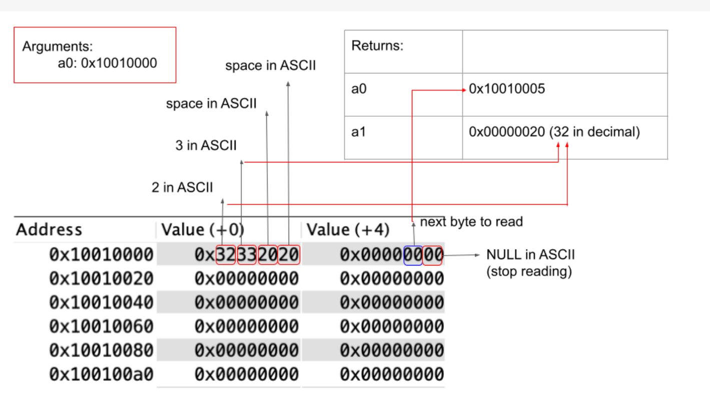
- NULL `(0x00)`
- '' `(0x20)`
- '\n' `(0x0A)`
- '\r' `(0x0D)`
- '\t' `(0x09)`

```
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
  a1: the number represented as an integer
```

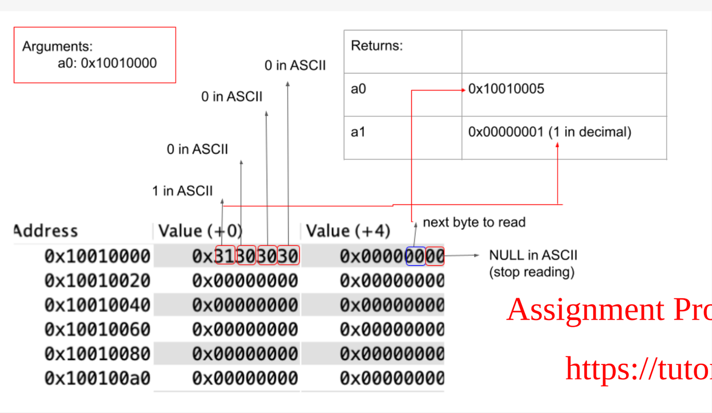
Here are some cases to consider.



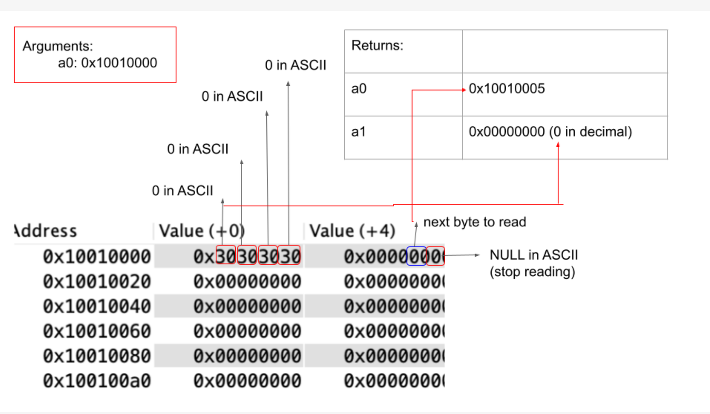
[illegible]



[illegible]



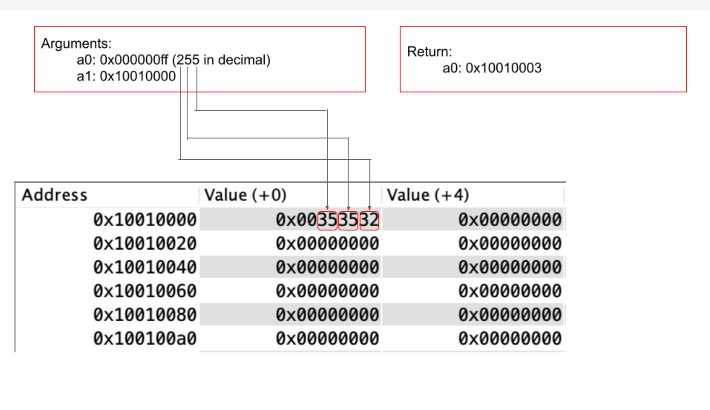
In this example, the return of `a0` should be `0x10010005` and `a1` should be `0x00000001` since we are ignoring the leading zeros (`0x30` in hex for ASCII).



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
  a1: 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.



Equivalently, if we translate the hexadecimal to ASCII,

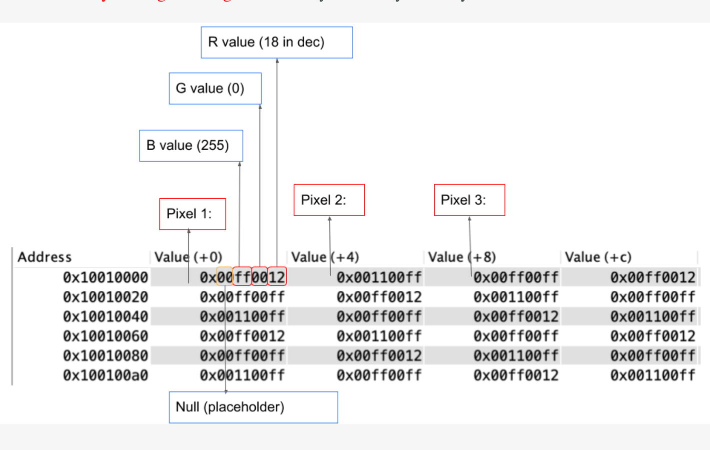
Address	Value (+0)	Value (+4)
0x10010000	\0 5 5 2	\0 \0 \0 \0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0

And the return of `a0` should be `0x10010003` 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)
  a1: 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.

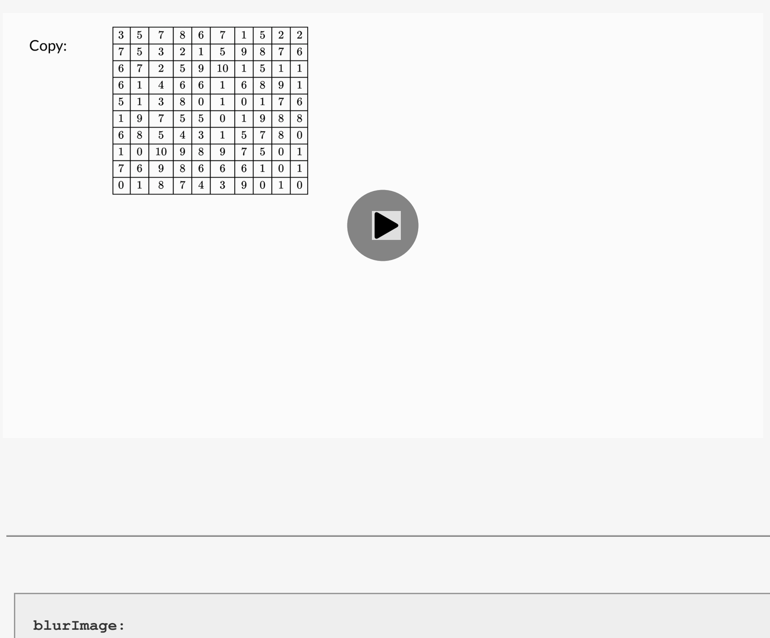



```
+---+---+---+---+---+---+-----------+
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 255 |
+---+---+---+---+---+---+-----------+
| R | G | B |Nul| R | G | ... | Nul |
+---+---+---+---+---+---+-----------+
<--------------->
        |
     1 pixel
```

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 $^{1}/_{49}$. It looks like the picture below.

$$\frac{1}{49}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

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`.




```
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
    a1: 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++) {
  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.correct.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.

Assumptions and Notes

- 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 doesn't run properly, use `beq zero` instead.
- If you use system calls or `ebreak` for debugging, remove them before submitting the lab. Do not print anything in the file `blur.s`.
- 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 program. You can use this for debugging.

Check My Lab

Link to CheckMyLab

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 (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.

Resources

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

Marking Guide

- 80% for correct program functionality
  - 5% for correctly implementing digitToAscii.
  - 5% for correctly implementing copy.
  - 5% for correctly implementing asciiToDigit.
  - 20% for correctly implementing readNextNumber.
  - 15% for correctly implementing storeNumber.
  - 15% for correctly implementing pixelKernelMul.
  - 15% for correctly implementing blurImage.
- 20% for program style
- Here is the mark sheet used for grading

Submission

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

- **Do not** add a `main` label to this file.
- **Do not** modify the line `.include "common.s"`.
- **Keep** the file `blur.s` in the `Code` folder of the git repository.
- **Push** your repository to GitHub before the deadline. Just committing will not upload your code. Check online to ensure your solution is submitted.