

## Section 1

- 1) The first variable is at \$t0 which loads the address of sort. It's at 0x00400010
- 2) First line of LA instruction: lui \$1, 0x00001001  
Second line of LA instruction: ori \$8, \$1, 0x0000003c
- 3) First line of LA instruction in hex: 0x3c011001  
Second line of LA instruction in hex: 0x3428003c

## Section 2

- 1) The first string I loaded was called **sorted** with "Sort Results: ".
- 2) Its first 4 characters translate to 53 6F 72 74 0A.
- 3) I looked at the register that said \$at which said 0x10010000
- 4) Ended up at 0x74726f53
- 5) T r o s and s e R and s t l u

## Reflection

This exercise made me step back and think about how loops really work, where conceptually it is just a bunch of code jumping over other code until it meets a condition. I learned how to use jump statements for repetition, this will be useful because I have a feeling knowing how to branch and jump is going to be a core part of writing assembly code. A few questions I have regard the interaction of .s files. I'm still a little confused on how to run code on a different file from a main file (although that might be covered later). Another question is about the difference between while and do-while loops: how they look in assembly. I know we haven't learned about do-whiles but it sounds interesting to me.

How did I find the information in the first part of this report?

After putting my main code into a directory with only one test case, I assembled the code and looked at the MARS "execute" and register section.

**Text Segment**

Offset	Address	Code	Basic	Source
0x00400200	0x00400200	jal 0x00400000	29:	jal studentMain
0x00400204	0x00400204	addi \$v0, \$zero, 4	36:	addi \$v0, \$zero, 4 # print_str(TESTCASE_MSG)
0x00400208	0x00400208	lui \$t1, 0x00000100	37:	la \$a0, TESTCASE_MSG
0x0040020c	0x0040020c	ori \$t1, 0x0000004c		
0x00400210	0x00000000	syscall	38:	syscall
0x00400214	0x00000001	addi \$v0, \$zero, 1	40:	addi \$v0, \$zero, 1 # print_int(alpha)
0x00400218	0x00000100	lui \$t1, 0x00000100	41:	la \$a0, alpha
0x0040021c	0x00400034	ori \$t1, 0x00000034		
0x00400220	0x00400000	lw \$a0, 0(\$a0)	42:	lw \$a0, 0(\$a0)
0x00400224	0x00000000	syscall	43:	syscall
0x00400228	0x0000000b	addi \$v0, \$zero, 11	45:	addi \$v0, \$zero, 11 # print_char(' ')
0x0040022c	0x00000020	addi \$a0, \$zero, 0x20	46:	addi \$a0, \$zero, 0x20
0x00400230	0x00000000	syscall	47:	syscall
0x00400234	0x00000001	addi \$v0, \$zero, 1	49:	addi \$v0, \$zero, 1 # print_int(beta)
0x00400238	0x00000100	lui \$t1, 0x00000100	50:	la \$a0, bravo
0x0040023c	0x00400030	ori \$t1, 0x00000030		
0x00400240	0x00400000	lw \$a0, 0(\$a0)	51:	lw \$a0, 0(\$a0)

**Data Segment**

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x74726f53	0x73655220	0x73746c75	0x2000203a	0x43000a00	0x56207275	0x65756c61	0x00203a73
0x10010020	0x5353454c	0x524f4d00	0x00000045	0x000004d2	0xffffffff	0x00000000	0x0000000a	0x00000001
0x10010040	0x00000001	0x00000001	0x00000001	0x65540a0a	0x1637473	0x56206573	0x61697261	0x20656c62
0x10010060	0x746f7544	0x0000203a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

**Registers**

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$t9	25	0x00000000
\$t0	26	0x00000000
\$t1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
\$pc		0x00400200
\$hi		0x00000000
\$lo		0x00000000

This is the whole screen where I was able to locate addresses and understand how string loading and la works.



These buttons are the bulk of how I ran my code. The left one actually assembled it, the middle runs the current program, and the last one increments the code running by one line (or one instruction). I was able to use these to find the addresses and hex values for all of 1.1-1.2.

I noticed that for the LA instructions, the address of the label is embedded into the actual machine code for values in lui and ori. The 32 bit address is made of two 16 bit constants.