

# Assignment-1 Report

Team- Subham Agarwala(IMT2022110)

Chandrima Nandi(IMT2022062)

For this assignment, we implemented **Bubble Sort** as the sorting algorithm for Question 1. The code is written in MIPS Assembly language and executed using MARS Simulator. For Question 2 we used Python to make an assembler that converts our assembly language program to machine code (32-bit binary).

## Question 1:

Code:

```
#t1- number of inputs
#t2- starting address of input
#t3- starting address of output

addi $t6,$t6,1 #t6 initialised to 1 to check if register used in slt has value 1
addi $s6,$zero,0 #s6-i=0
addi $s0,$t1,-1 #s0=n-1
addu $s2,$zero,$t2 #s2 will have address of input
addu $s3,$zero,$t3 #s3 will have address of output

#this section will copy the input array to the output address provided
copy:
    slt $t0,$s0,$s6 #checks (n-1)<i, i starting from 0 to run n times
    beq $t0,$t6,sorting #when i>(n-1) i.e loop has run n times, we start sorting
    lw $t4,0($s2) #t4=input[i]
    sw $t4,0($s3) #output[i]=t4
    addi $s2,$s2,4 #moving input and output address one address ahead
    addi $s3,$s3,4
    addi $s6,$s6,1 #i++
    j copy #looping

sorting:
    addi $s6,$zero,-1 #s6=-1, s3=j. Starting j from -1 because it value increases before first jump to inner loop
    addi $s0,$t1,-2 #s0=n-2

outer_loop: slt $t0,$s0,$s6 #checks if i>n-2 (running loop n-1 times)
            beq $t0,$t6,print #when all numbers are iterated over and checked, we move on to print the result
            addi $s6,$s6,1
            addi $s3,$zero,0 #initializing j to 0
            addu $s2,$zero,$t2 #s2 will be used to access elements of array, reset to starting address of output at every iteration of outer loop
            j inner_loop

inner_loop: sub $t1,$s0,$s6 #t1=n-i-1
            slt $t0,$s1,$s3 #checks if i>n-1-j
            beq $t0,$t6,outer_loop
            lw $t4,0($s2) #t4=output[j]
            lw $t5,4($s2) #t5=output[j+1]
            slt $t0,$t5,$t4 #checks if output[j]>output[j+1]
            beq $t0,$t6,swap #if yes then the values are swapped
            return_here: #returns to this line after values are swapped
                    addi $s2,$s2,4

            addi $s3,$s3,1 #j++
            j inner_loop #looped back to inner_loop

swap: sw $t5,0($s2) #output[j]=output[j+1]
      sw $t4,4($s2) #output[j+1]=output[j]
      j return_here

print:
#endfunction
```

In the code, the integers to sort received as input from the user are copied to the output address for manipulation leaving the input unchanged. The code is well documented to explain every step in the code. Bubble sort has been used as the

sorting algorithm which iterates over every element and sorts them in ascending order.

**Result (executed in MARS):**

The screenshot displays the MARS MIPS simulator interface, which is divided into several panels:

- Text Segment:** A table showing assembly instructions with their addresses, codes, basic forms, and source comments.
 

Bkpt	Address	Code	Basic	Source
	4194304	0x0c10004c	jal 4194608	14: jal print inp statement
	4194308	0x0c10003f	jal 4194556	15: jal input int
	4194312	0x000c4821	addu \$9,\$0,\$12	16: move \$t1,\$t4
	4194316	0x0c100051	jal 4194628	20: jal print inp int statement
	4194320	0x0c10003f	jal 4194556	21: jal input int
	4194324	0x000c5021	addu \$10,\$0,\$12	22: move \$t2,\$t4
	4194328	0x0c100056	jal 4194648	26: jal print out int statement
	4194332	0x0c10003f	jal 4194556	27: jal input int
	4194336	0x000c5821	addu \$11,\$0,\$12	28: move \$t3,\$t4
	4194340	0x000ac021	addu \$24,\$0,\$10	32: move \$t8,\$t2
	4194344	0x0000b821	addu \$23,\$0,\$0	33: move \$s7,\$zero #i = 0
- Labels:** A list of labels and their corresponding addresses.
 

Label	Address
template.asm	
loop1	4194348
loop1end	4194376
copy	4194400
sorting	4194432
outer_loop	4194440
inner_loop	4194464
return_here	4194492
swap	4194504
print	4194516
- Data Segment:** A table showing memory addresses and their values at various offsets.
 

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268501088	540682612	1953383680	1931506277	1953653108	543649385	1919181921	544437093	1864394351
268501120	1970304117	673215348	1679847017	1835623269	1713400929	1634562671	540682612	1953383680
268501152	1948283493	1763730792	1734702190	540701285	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	5	4	3	2	1	0	0	0
268501248	1	2	3	4	5	0	0	0
268501280	0	0	0	0	0	0	0	0
268501312	0	0	0	0	0	0	0	0
268501344	0	0	0	0	0	0	0	0
- Mars Messages:** A log of user input and program output.
 

```

Enter No. of integers to be taken as input: 5
Enter starting address of inputs(in decimal format): 268501216
Enter starting address of outputs (in decimal format): 268501248
Enter the integer: 5
Enter the integer: 4
Enter the integer: 3
Enter the integer: 2
Enter the integer: 1
1
2
3
4
5
      
```

## Question 2:

An assembler made using Python to convert the assembly code in question 1 to machine code. The 'registers' dictionary provides a mapping between the registers used in the assembly code and the binary equivalent that represents them. The 'labels' register provides necessary information when the assembler encounters a label, like the address the execution should jump to or the number of instructions that need to be skipped. The 'code' list has the entire code, storing each instruction as a different element. The 'binary' function converts decimal numbers to binary depending on whether they are positive or negative. If the number is negative it returns the 2s Complement representation.

### Output of the code:

```
00100001110011100000000000000001
00100000000101100000000000000000
00100001001100001111111111111111
00000000000010101001000000100001
00000000000010111001100000100001
00000010000101100100000000101010
0001000100001110000000000000110
10001110010011000000000000000000
10101110011011000000000000000000
001000100101001000000000000000100
001000100111001100000000000000100
00100010110101100000000000000001
000010000001000000000000000011000
00100000000101101111111111111111
00100001001100001111111111111110
00000010000101100100000000101010
00010001000011100000000000010001
00100010110101100000000000000001
00100000000100110000000000000000
00000000000010111001000000100001
00001000000100000000000000101000
00000010000101101000100000100010
00000010001100110100000000101010
00010001000011101111111111110111
10001110010011000000000000000000
100011100100110100000000000000100
00000001101011000100000000101010
00010001000011100000000000000011
001000100101001000000000000000100
00100010011100110000000000000001
00001000000100000000000000101000
10101110010011010000000000000000
101011100100110000000000000000100
00001000000100000000000000101111
```