EG 211 Computer Architecture Assignment – 1:

1.RAJANALA SAI DHEERAJ (IMT2022093)

2.MANNEPALLI KRISHNA SATHWIK (IMT2022045)

> The program we chose is sorting in ascending order.

> Algorithm in C++ (SELECTION SORT)

```
void selectionSort(int arr[], int n)
{
	int i, j, min_idx;
	for (i = 0; i < n - 1; i++) {
		min_idx = i;
		for (j = i + 1; j < n; j++){
			if (arr[j] < arr[min_idx])
				min_idx = j;
			}
		if (min_idx != i)
			swap(arr[min_idx], arr[i]);
	}
}
```

MARS COMPILER OUTPUT:

```
Enter No. of integers to be taken as input: 8
Enter starting address of inputs(in decimal format): 268501216
Enter starting address of outputs (in decimal format): 268501280
Enter the integer: 12
Enter the integer: 14
Enter the integer: 2
Enter the integer: 7
Enter the integer: 5
Enter the integer: 11
Enter the integer: 4
Enter the integer: 8
2
4
5
7
8
11
12
14

-- program is finished running --
```

- **THIS IS A SCREENSHOT OF THE COMPILATION OF SORTING "EIGHT" INTEGERS INTO ACSENDING ORDER.**

Explanation for sorting algorithm in Mips assembly program language:

1. **Copying part:**

- $t4 and $t5 stores address locations of input and output memory locations from $t2 and $t3 respectively.
- Loop2 runs from 0 to n and copies input values from input memory location to output memory location.

2. Sorting part:

- 1. Initialization: - $s0 is set to 0 to act as an iterator i for the outerloop. $s1 is set to 1 act as iterator j i.e. iterator for innerloop. $t7 is set to n - 1, where n is the total number of integers in the array.

  $s5 is initialized to base address of the array $t3 and is incremented such as to match address pointed by i.

  $t6 is initialized to base address of the array $s5 and is incremented such as to match address pointed by j.

  $t8 is initialized to base address of the array $s5 and is incremented such as to match address pointed by min_ind.

- 2. Outer Loop (Mainloop1): - It compares the value in $s0 (initialized as the index i) with $t7 (n-1) to determine if the outer loop should continue. - If the condition is met, it proceeds to the inner loop to find the minimum element in the unsorted portion of the array.

- 3. Inner Loop (Loop2): - It uses $s1 as an iterator (`j`) to traverse the unsorted portion of the array. - Compares the elements at the indices Arr[min_ind] [$t4] and Arr[j] [$t5] and updates the index of the minimum element min_ind [$t9] to [$s0] if a smaller element is found.

- 4. Cycleloop: - It iterates address from $t3 to location of new min_ind (by iterating $t9 number of times) and stores address location in $s4 ,this locations is later used to in swap.

- 5. Swap: - If a smaller element is found in the inner loop, a swap operation is performed between new Arr[min_ind] "[$s4]" and Arr[i] "[$s6]" to put it in the sorted position of the unsorted array.

- 6. Loop Control and Incrementing: - After each iteration of the inner loop, the loop control and index variables ($t6, $s1) are updated accordingly. - The program jumps back to outerloop until the entire array is sorted in outerloop [$t8, $s0, $s5] are updated accordingly to maintain addresses of each variable accordingly in the registers.

- 7. sorting_end: - The program exits when the outer loop completes (endouterloop) as all elements are sorted and stored in the memory location started from "$t3".

  In summary, this MIPS assembly code uses nested loops to i mplement the Selection Sort algorithm, which repeatedly finds

the minimum element in the unsorted part of the array and swaps with the first unsorted element. This process continues until the entire array is sorted.

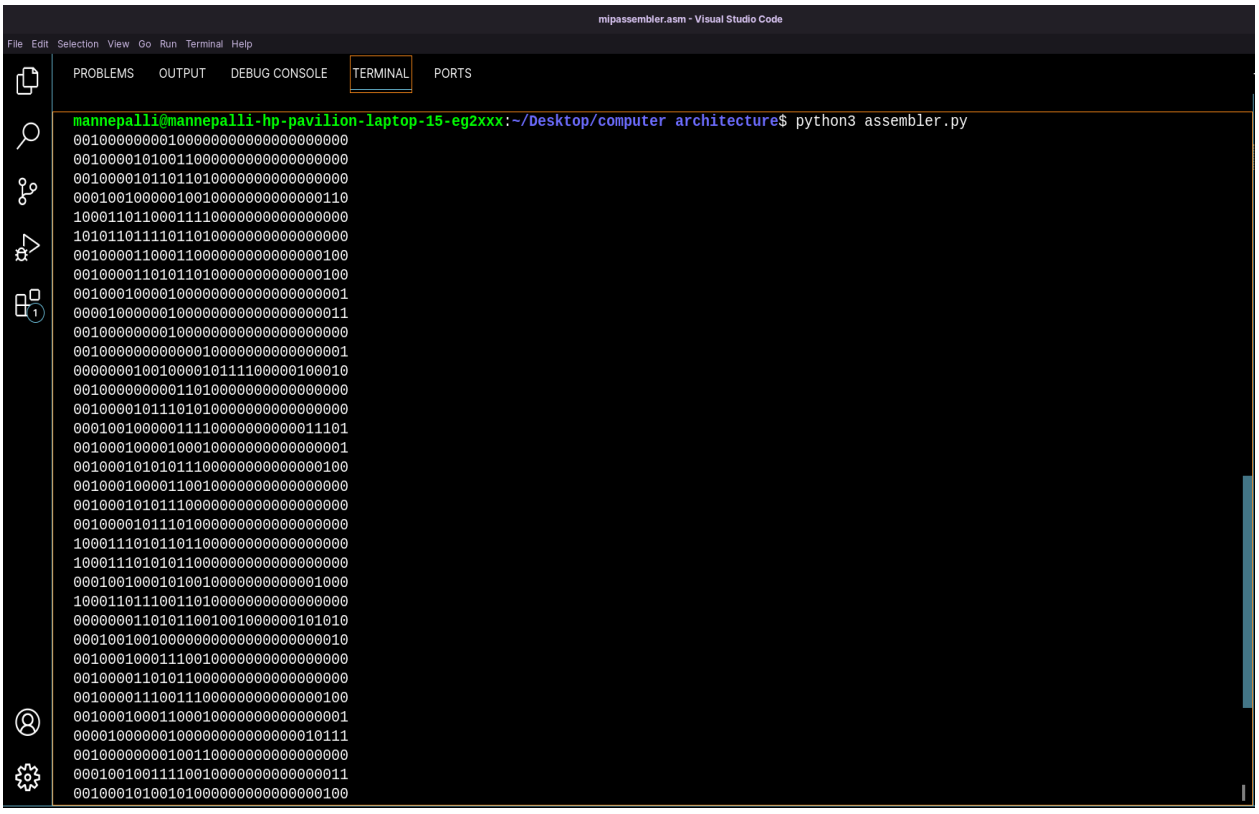
Explanation for the Assembler:

1. Defined dictionaries for R-format and I-format and J-format instructions separately. Another dictionary for registers and labels.
2. Implements an assembler that takes an instruction and generates 32-bit machine code for the respective instruction.
3. Reads a MIPS assembly file(mipassembler.asm), processes it and extracts labels and instructions.
4. I stored the labels and their addresses for instructions beq and j instructions. The addresses of those labels are stored in a dictionary register-numbers along with the registers.

Brief overview of how the code works:

- After defining dictionaries there is a function that converts decimal to binary and it converts the decimal input provided into required no of bits in binary.

- For the instructions such as beq and jump the addresses of labels are already stored in the dictionary.
- Now comes the if, elseif conditions that checks the instruction specifically. That is whether it's Addi or beq or j or Subi or slt or etc.... and prints the required 32-bit machine code for each instruction in mipassembler.asm.



The bottom most line here and the topmost line in the below screenshot are same

```
00100010100101000000000000000100
00100010011100110000000000000001
00001000000100000000000000100001
00010011001100000000000000000011
10001110100101110000000000000000
10101110110101000000000000000000
10101110111010100000000000000000
00100011000110000000000000000100
00100010000100000000000000000001
00100010101101010000000000000100
mannepalli@mannepalli-hp-pavilion-laptop-15-eg2xxx:~/Desktop/computer architecture$
```

The above pasted screenshots are our assembler's output.

Now we will be pasting the screenshot of running our code in MARS (MIPS Assembler and Runtime Simulator). There will be 45 lines in both of our outputs. But the no of instructions is 44. This is happening due to the split of Subi instruction which is splitted into Addi and sub.

```
00100000000010000000000000000000
00100001010011000000000000000000
00100001011011010000000000000000
00010010000010010000000000000110
10001101100011110000000000000000
10101101101011110000000000000000
00100001100011000000000000000100
00100001101011010000000000000100
00100010000100000000000000000001
00001000000100000000000000000011
00100000000100000000000000000000
00100000000000010000000000000001
00000001001000010111100000100010
00100000000011010000000000000000
00100001011101010000000000000000
00010010000011110000000000011101
00100010000100010000000000000001
00100010101011100000000000000100
00100010000110010000000000000000
00100010101110000000000000000000
00100001011101000000000000000000
10001110101101100000000000000000
10001110101011000000000000000000
00010010001010010000000000001000
10001101110011010000000000000000
00000001101011001001000000101010
00010010010000000000000000000010
00100010001110010000000000000000
00100001101011000000000000000000
00100001110011100000000000000100
00100010001100010000000000000001
00001000000100000000000000010111
00100000000100110000000000000000
00010010011110010000000000000011
00100010100101000000000000000100
00100010011100110000000000000001
00001000000100000000000000100001
00010011001100000000000000000011
10001110100101110000000000000000
10101110100101100000000000000000
10101110101101110000000000000000
00100011000110000000000000000100
00100010000100000000000000000001
00100010101101010000000000000100
00001000000100000000000000001111
~
~
~
```