Insertion sort algorithm in RISC-V assembly

The insertion sort algorithm works by comparing elements with each other. Sorting them one element at a time. In other words, it works kind of like how you would sort cards with your hands. The algorithm can be very slow and inefficient for massive data values. However, it is efficient for small data values or partially sorted data.

At first, the algorithm compares the second element of the array with the first one. If the second element is greater than the first, nothing changes. If, however they are not in the ascending order, then they will be swapped. The second pass compares the third and the second element and so on. If at some point of the insertion sort there is a smaller value than the already sorted sub-array, that element will be compared with every element until it is set to the correct place.

In the RISC-V implementation, we first define the array, which we will later sort using the Insertion sort algorithm.

Defining the array in RISC-V assembly goes:

First: The size of the array using Add immediate instruction.

addi a2, x0, 10	this instruction effectively sets the value of register a2 to 10. In this context, the immediate 10 represents the size of the array.

Defining the array:

```
addi t0, x0, 32
sw t0, 0(a0)
```

```
  addi t0, x0, 8

  sw t0, 4(a0)

# … And so on
```

The values for the elements of the array are given by the immediate values 32 and 8, which are loaded into the register t0 with addi instruction. The they are stored in the array using the sw instruction. The offsets of 0 and 4 are used to specify the location of each element within the array.

The instruction (store word) sw t0, 0(a0)

stores the content of the register t0 into the memory address provided by the register a0. The offset is 4, so therefore we define all the elements of the array by increasing that by 4.

After defining the array, we need to keep count of the numbers being compared and sorted. We need to know what memory access contains next value to be compared.

We do that by:

```
addi s2, x0, 4
```

The register s2 is used to keep track of the memory address that should be accessed to get the next value to be compared in the sorting process. The value 4 is added to the register s2 because the array elements are stored in consecutive memory locations with an offset of 4 bytes between each element.

And by:

```
addi s1, x0, 1
```

The register s1 is used to keep track of how many numbers have been sorted and to exit the sorting loop when all numbers are in the correct order.

Then we call the jal (jump and link) to jump to Insertion_sort and finally jump to Exit.

jal x0, Insertion_sort

The Insertion_sort itself jumps to various other methods (explained in detail below) if needed.

Check_order:

The Check_order method is used to check if the numbers in the array are in order. It does this by comparing the value of the s1 register with the value in the a2 register, which is the size of the array. If s1 is equal to a2, it means that all the elements in the array have been sorted, so the method jumps to the Exit label to terminate the sorting algorithm.

If s1 is not equal to a2, it is because not all the elements in the array have been sorted, so the method calls the Insertion_sort method recursively to continue the sorting process. So s1 register is used to keep track of the number of elements that have been sorted so far, while the s2 register is used to keep track of the memory address of the element being compared.

Change_places:

The Change_places method is used to swap the places of two elements in the array. It does this by using the a1 register to access the memory locations of the two elements being compared. It also uses registers t1 and t2 for storing the values of those elements.

The method loads the value of the element at the memory location pointed to by a1 into the register t1 and the value of the element at the memory location pointed to by a1 - 4 into the register t2.

Next, the method compares the value in the register t1 with the value in the register t2 using the lbt (branch if less than) instruction. If the value of t1 is less than the value of t2, the method swaps the values of t1 and t2 by storing the value of t2 in the memory location pointed to by a1 and the value of t1 in the memory location pointed to by a1 - 4.

Finally, the method checks if the element that was originally in the first memory location is now in the correct location by comparing the value of the register a1 with the value of the register x0 (is always zero). If a1 is equal to x0, it means that the element has been placed in the first memory location, so the method jumps to the Increase_counter method to update the s2 and s1 registers and continue the sorting process. If a1 is not equal to x0, it means that the element has not been placed in the correct location yet, so the method calls the Sort method to continue the sorting process for that element.

Sort:

The Sort method is used to continue the sorting process for an element that has not yet been placed in the correct location in the array. It works in a similar way to the Insertion_sort method, but it does not update the value of the register a1 to match the value of the register s2.

First, the method loads the value of the element at the memory location pointed to by a1 into the t1 register and the value of the element at the memory location pointed to by a1 - 4 into the t2 register.

Next, the method compares the value in the t1 register with the value in the t2 register using the BLT (branch if less than) instruction. If the value in t1 is less than the value in t2, the method swaps the values of t1 and t2 by storing the value of t2 in the memory location pointed to by a1 and the value of t1 in the memory location pointed to by a1 - 4.

Finally, the method updates the s2 and s1 registers to reflect the fact that an element has been sorted, and then calls the Check_order mehtod to see if all the elements in the array are in order. If they are, the sorting algorithm terminates. If they are not, the Insertion_sort method is called recursively to continue the sorting process.

Increase_counter:

The Increase_counter method is used to update s2 and s1 registers when an element has been placed in the correct location of the array.

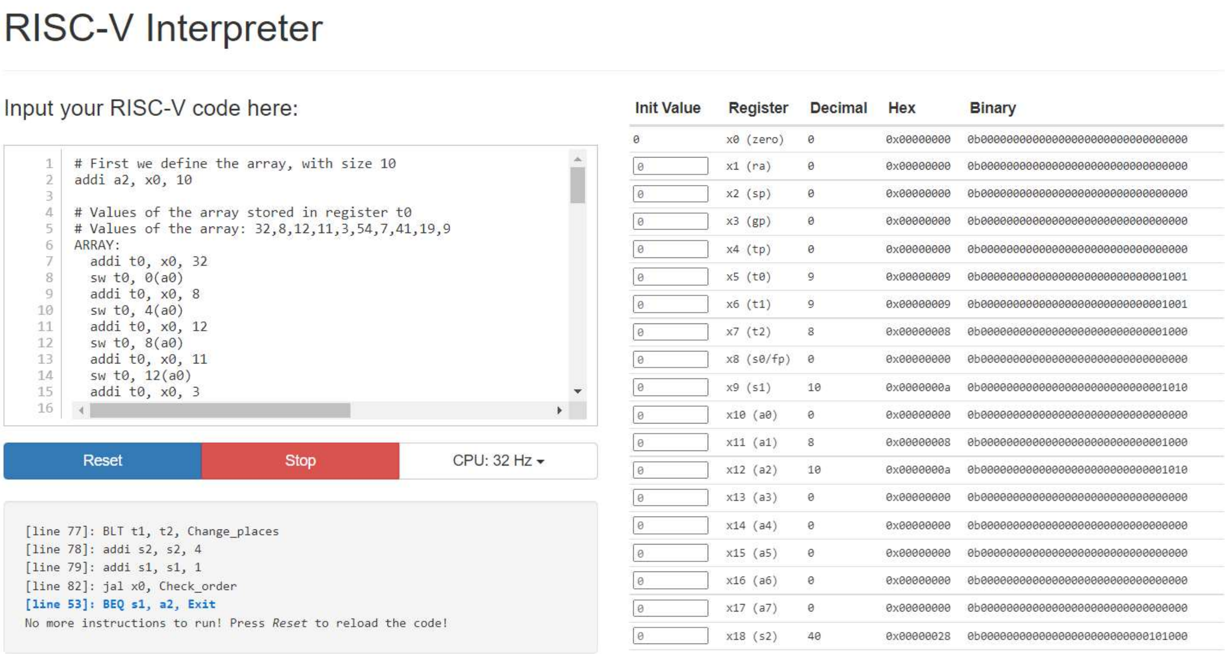
It does this by simply adding 4 to the value of the register s2 and 1 to the value of the register s1. After the registers have been updated, the method calls the Check_order method to see if all the elements in the array are in order. If they are, the sorting algorithm stops. If not, the Insertion_sort method is called recursively, and sorting continues.

Exit:

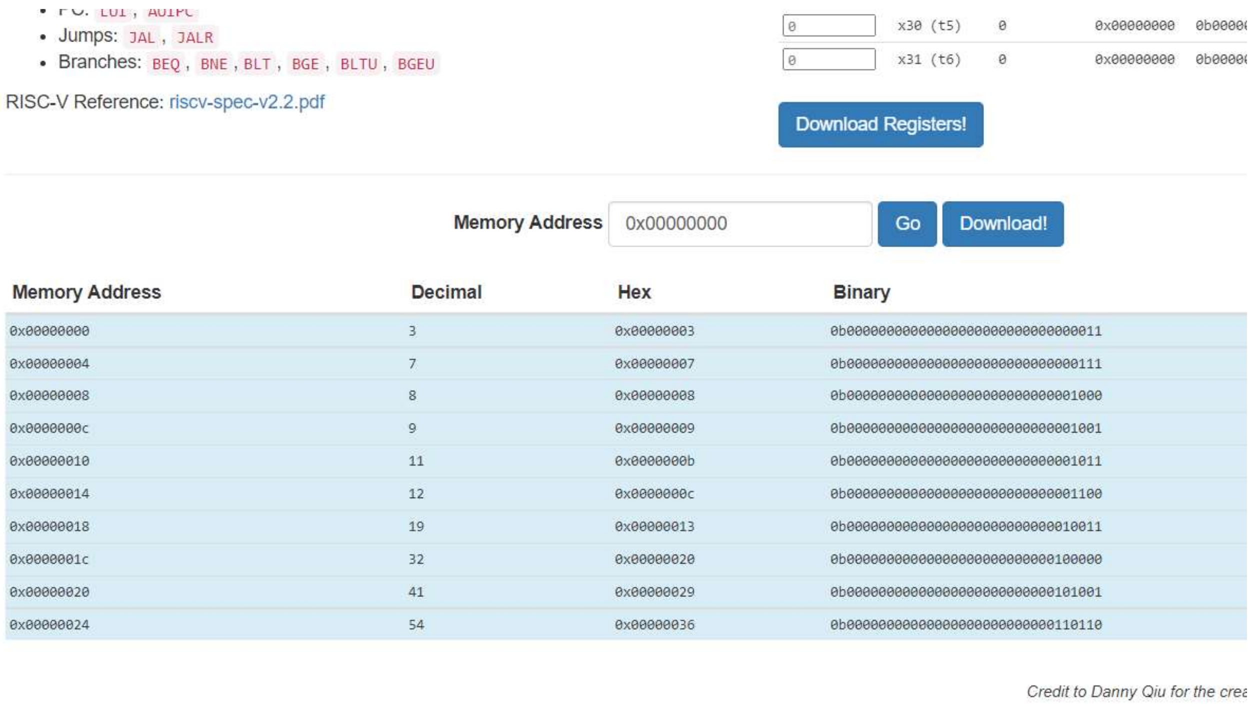
Does not contain any instructions or perform any actions. It is simply a label that marks the end of the sorting algorithm.

The takeout

When testing the algorithm with RISC-V Interpreter (https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#) it seems to work like I intended. I kept testing with the Interpreter since I didn't have anywhere else to run the code.



The array after running the insertion sort algorithm below:



Memory Address	Decimal	Hex	Binary
0x00000000	3	0x00000003	0b00000000000000000000000000000011
0x00000004	7	0x00000007	0b00000000000000000000000000000111
0x00000008	8	0x00000008	0b00000000000000000000000000001000
0x0000000c	9	0x00000009	0b00000000000000000000000000001001
0x00000010	11	0x0000000b	0b00000000000000000000000000001011
0x00000014	12	0x0000000c	0b00000000000000000000000000001100
0x00000018	19	0x00000013	0b00000000000000000000000000010011
0x0000001c	32	0x00000020	0b00000000000000000000000000100000
0x00000020	41	0x00000029	0b00000000000000000000000000101001
0x00000024	54	0x00000036	0b00000000000000000000000000110110

Credit to Danny Qiu for the crea

Making this algorithm in RISC-V assembly language seemed a bit odd at the beginning, since I have not even coded anything in C before this. Since the programming language C is closer to machine language than lets' say for example Java, I thought it was a good starting point. With that the first thing was to fully understand how the algorithm could be implemented in C.

By making the methods one at the time was the key to making the project have any logic. I have a little experience of programming languages like Microsoft's Visual Basic which uses the "goto" method to jump to a specific line.

I would say that that the main point or the takeout of this project was understanding how the assembly language works. Understanding it makes a difference when writing efficient code on high level programming languages.