Stack usage:

My implementation consists of three main procedures: post_order, insert_iter, and new_node.

In the post_order procedure, the $ra and $s5 registers are saved on the stack because recursion overwrites their values. The stack layout is as follows: 0($sp) -> $s5 (current node pointer), 4($sp) -> $ra (return address).

The insert_iter and new_node procedures do not use the stack during their executions. insert_iter only utilizes temporary registers, while new_node simply uses the heap pointer $s2 to allocate nodes.

In summary, stack usage is limited to the post_order procedure, which recursively traverses the BST in postorder to print node keys.

Implementation details:

I first implemented the traversal algorithm in C to provide a reference for my assembly code. I then structured my assembly implementation into four main steps corresponding to reading input, creating nodes, inserting nodes iteratively, and performing postorder traversal.

Managing large input sizes presented challenges, leading me to switch from a fully recursive insertion to a part iterative approach. The post_order traversal still relies on recursion, with the stack used to preserve $ra and $s5 across calls.

As this was my first experience with QTSpim, I encountered difficulties such as repeatedly reloading the same file and tracking registers. I also faced stack overflows due to incorrect management of the stack pointer. To address these issues, I adjusted my implementation to handle larger inputs efficiently and developed a systematic method to annotate and monitor register usage throughout the program.