`encodeRType` Function

This function encodes R-type instructions by following the RISC-V format. The format for R-type instructions includes fields for `opcode`, `funct3`, `funct7`, `rd`, `rs1`, and `rs2`.

- **Inputs:**
  - `instr`: The instruction mnemonic (e.g., "add", "sub").
  - `rd`: The destination register number.
  - `rs1`: The first source register number.
  - `rs2`: The second source register number.
- **Outputs:**
  - A 32-bit binary value representing the encoded instruction.
- **Steps in the Function:**
  - The `opcode` is hardcoded to `0b0110011`, which is the standard opcode for R-type instructions.
  - The function checks the value of `instr` and assigns the appropriate values for `funct3` and `funct7`. These fields differentiate between instructions like `add` and `sub`.
  - The function checks if the provided register numbers (`rd`, `rs1`, `rs2`) are within the valid range (0-31). If not, it reports an error.
  - The 32-bit instruction is then constructed by shifting and combining the values of `funct7`, `rs2`, `rs1`, `funct3`, `rd`, and `opcode`.
  - The result is returned as a 32-bit unsigned integer representing the encoded machine code.
- **Error Handling:**
  - If the `instr` is unsupported, it returns `0xFFFFFFFF`.
  - If any register value is out of range, the function reports an error and returns `0xFFFFFFFF`.

`reg_num` Function

This function translates register names (such as `x0`, `a0`, `t0`, etc.) into their corresponding numeric values (0-31).

- **Inputs:**
  1. `str`: A string representing a register name.
- **Outputs:**
  1. The corresponding register number.
- **How It Works:**
  1. The function uses a series of `if-else` statements to compare the input string to known register names.
  2. If a match is found, the corresponding register number is returned.

3. If no match is found, the function returns `-1`, indicating an invalid register.

`removeCommas` Function

This function removes commas from a string. This is useful for handling instruction formats like `add x1, x2, x3`, where register names are separated by commas.

- **Inputs:**
    1. `source`: The string containing the instruction and register names.
- **Outputs:**
    1. A string without commas.
- **Steps in the Function:**
    1. It loops through the input string and copies characters that are not commas into a new string (`result`).
    2. Once the loop completes, the result string is copied back to the original source string.

Main Program Workflow

1. **File Handling:**
    - The program opens an input assembly file (`input.s`) and an output file (`output.hex`). If the input file cannot be opened, it reports an error and exits.
2. **Reading Lines:**
    - The program reads the input file line by line using `fgets()`. For each line:
        - It removes any newline character.
        - If the line is empty, the program skips it.
3. **Parsing Instructions:**
    - The program parses each line using `sscanf()` to extract the instruction mnemonic (`instr`), and the register names (`rd`, `rs1`, `rs2`).
4. **Removing Commas:**
    - It calls the `removeCommas()` function to clean the register names.
5. **Converting Registers to Numbers:**
    - The program uses the `reg_num()` function to convert the register names into their corresponding numeric values.
6. **Encoding the Instruction:**
    - The program calls `encodeRType()` with the instruction mnemonic and the register numbers. If the instruction is unsupported, an error is printed.
7. **Writing to Output File:**
    - If the instruction is successfully encoded, the resulting machine code is printed to the console and written to the output file (`output.hex`) in hexadecimal format.
8. **Error Handling:**

    - The program gracefully handles errors related to file opening, invalid instruction formats, unsupported instructions, and invalid register names.

Error Handling and Debugging

- The program includes several layers of error handling:
    - If an invalid instruction is encountered, the program reports the error and skips to the next line.
    - If register values are out of range (not between 0 and 31), an error is printed, and the current line is skipped.
    - The program checks the number of tokens parsed from the line and skips lines that do not follow the expected format.
    - The program outputs debugging information to help trace the current state of execution, such as which line is being processed