# Detailed Report on 5 Stage Pipeline Stall Detector

This C program is designed to identify potential pipeline stalls in an assembly program and modify it to be stall-free by inserting no-operation (nop) instructions. The program takes a .txt file containing assembly program as input and outputs a modified program with added nops to avoid stalls in the terminal.

## Code Structure:

Here is an analysis of the coding approach and steps taken for the code to work correctly:

### Header Files and Constants:

The program begins with standard C library inclusion and the definition of constants, specifying maximum array dimensions and sizes. These constants help maintain clarity and manage array sizes efficiently.

### reg_num(char *arr):

This function is responsible for mapping a given register name to its corresponding register number. It supports both the ABI register names (e.g., "t0") and their alternative "x" names (e.g., "x5"). It helps in decoding registers within assembly instructions. This function takes a string containing a register name as input.

### sliceString(char *str, int start, int end):

The 'sliceString' function extracts substrings from character arrays. This function is used to extract specific portions of assembly instructions and returns a substring of the original string between these indices. In the function the start index is inclusive but the end index is exclusive.

### decode(char *arr, int *arr2, int type):

The 'decode' function is responsible for decoding registers within an assembly instruction. It analyses the input assembly instruction string, extracts the register names, and maps them to their corresponding register numbers using the 'reg_num' function. The type parameter determines whether the instruction is of R-type (type 1) or a load instruction (type 2) or store instruction (type 4) or other I type instructions (type 3).

### nop_counter(int arr[], int arr2[], int type):

The 'nop_counter' function calculates the number of no-operation (nop) instructions needed based on register dependencies and the type of instruction. It determines whether nops are required to avoid pipeline stalls. The type being 1 considers there is no instruction between 2 instructions having register dependencies , but for any other value considers there is exactly 1 instruction between 2 instructions having register dependencies.

## get_type(char arr[][MAXCOLS], int row):

This function determines the type of instruction based on its mnemonic. It returns 2 for load instructions, 3 for other I type instructions except 'jalr', 4 for store instructions and 1 for other instructions. It's used to identify the instruction type which is helpful throughout the code especially for 'decode' function.

## nops(int row, char arr[][MAXCOLS], char arr2[][5]):

The 'nops' function is responsible for detecting and resolving pipeline stalls by inserting nop (No-Operation) instructions. It examines the dependencies between the current instruction and the previous two instructions (if they exist) and inserts nops accordingly to avoid pipeline stalls.. The number of inserted nops is recorded in the clock_cycles_1 and clock_cycles_2 variables, based on whether data forwarding is available.
If it finds hazards, such as register dependencies, it inserts the appropriate number of nop instructions into arr2. The specific conditions for nop insertion vary depending on the instruction types and dependencies.

## Main Function:

The main function is the entry point of the program. It coordinates the overall execution of the pipeline stall detection and simulation. Here's a brief explanation of what the main function does:

1. The code begins by prompting the user for an input file containing the assembly code. It reads the file and stores the instructions in a two-dimensional array named `arr`. The code is designed to handle files with a maximum of 50 lines, with each line being less than 24 characters.
2. It initializes arrays to store assembly instructions and modified instructions with nops. It also sets clock cycle counters (clock_cycles_1 and clock_cycles_2) to 4.
3. The program then proceeds to analyse the assembly code line by line, starting from the last line and working its way up to the first line. For each line:
    - It calls the nops function to identify potential pipeline stalls and insert nops if necessary.
    - It increments clock cycle counters for the two scenarios: no data forwarding and no hazard detection (clock_cycles_1) and data forwarding but no hazard detection (clock_cycles_2).
4. After analysing all lines, the program prints the results, including the modified assembly program and the total clock cycles required for both scenarios.

In conclusion, this program is a tool for identifying and mitigating potential pipeline stalls in assembly programs. By detecting dependencies between instructions, it ensures efficient execution and minimizes stalls. By analysing the modified assembly output, users can better understand how stalls impact the performance of their programs and explore potential optimizations.