Overall Test Plan

For our testing plan the first phase is going to be testing each of the individual components of the Call Stack Module. We will test normal and abnormal conditions and user interactions for the C code compiler, the Assembly code compiler, and the corresponding animation of the stack and registers. This is to ensure each of the components of our module are working well individually and not interfering with one another. Our first phase is simply to check that each one is doing its specific task before we move on to integration testing and functionality between how the components interact with each other. We will create user interactions and test assembly and c code to input into the compilers to test the animation usability and correctness. We will also test incorrectly formatted and inaccurate c code to ensure that the program is giving correct error messages and not attempting to run the incorrect code. Another phase of the testing plan is going to be testing the connection between the frontend and the server in the backend to ensure it is correctly running the conversion script for the C code to Assembly. We will create various test cases with different C code and assembly instructions to ensure the conversion script is able to handle specific commands to add them into the stack and register animations. We will also test the connection between the client and server to see the load it can handle as well as the response rate to ensure it is not getting overloaded especially once we are getting into user profiles for both students and professors. The last phase will be testing the integration between the modules, compilers and animations to ensure that they all are working seamlessly and there aren't any major issues or concerns. The majority of our testing will be front end interactions with compilers and animations.

Test Case Descriptions

VC 1.1 Compiles Valid C Code
VC 1.2 This test will ensure that the compiler recognizes and correctly compiles inputted C code
VC 1.3 The user will input valid C code into the editor, compile it, and see that it does not spit out an error and compiles to correct assembly code
VC 1.4 Inputs: The inputs for this test is correctly implemented C code
VC 1.5 Outputs: Correct and valid converted assembly code
VC 1.6 Normal
VC 1.7 Blackbox
VC 1.8 Functional
VC 1.9 Unit Test
VC 1.10 Results: Correct converted C code to assembly output in the terminal,user types in a simple c code function such as hello world and then hits the generate assembly code button, then the user sees the correctly compiled assembly version of the code.

EM 1.1 Error Message for Invalid C Code
EM 1.2 This test will ensure that the compiler recognizes incorrectly formatted/improper C code

EM 1.3 The user will input invalid C code into the editor, attempt to compile it, and see that the compiler outputs the error message in the terminal
EM 1.4 Inputs: The inputs for this test is incorrectly compiled C code
EM 1.5 Outputs: Error message being printed to the terminal saying the C code is invalid
EM 1.6 Abnormal
EM 1.7 Blackbox
EM 1.8 Functional
EM 1.9 Unit Test
EM 1.10 Results: Error message properly displayed in the terminal. The user will input an improperly formatted C code and once they try to generate the assembly the pop error message will display for the user to let them know it is inaccurate.

CC 1.1 C Code Conversion Properly Runs in the Stack Animation
CC 1.2 This test will ensure that the stack animation properly pulls the C code from the compiler and adds it into the animation
CC 1.3 The user will input valid C code into the editor, compile it, and see that the compiled code gets added to the stack animation properly
CC 1.4 Inputs: The inputs for this test is correctly compiled C code
CC 1.5 Outputs: The output for this test is correctly animated C code in the stack
CC 1.6 Normal
CC 1.7 Blackbox
CC 1.8 Functional
CC 1.9 Integration
CC 1.10 Results: The animation correctly runs through the stack with the correctly inputted C code

BF 1.1 Next Button Functionality
BF 1.2 This test will ensure that the Next button on the stack home page properly moves forward with the stack animation
BF 1.3 The user will input valid C code into the editor, compile it, and click through the next button to see that it moves through the stack animator
BF 1.4 Inputs: The inputs for this test is correctly compiled C code and the user clicking through the next button
TEST 1.5 Outputs: The outputs for this test are the proper animations of showing what is in each place in the stack for the converted C to assembly code
BF 1.6 Normal
BF 1.7 Blackbox
BF 1.8 Functional
BF 1.9 Integration
BF 1.10 Results: The next button correctly clicks through the stack animation showing the proper code/elements in the stack with correct functionality and is very user friendly and instant response time so there is no lagging between user input and the animation actually moving forward

BF 2.1 Back Button Functionality
BF 2.2 This test will ensure that the Back button on the stack home page properly moves backward with the stack animation
BF 2.3 The user will input valid C code into the editor, compile it, and click through the back button to see that it moves through the stack animator and is matching with the code
BF 2.4 Inputs: The inputs for this test is correctly compiled C code and the user clicking through the back button
BF 2.5 Outputs: The outputs for this test are the proper animations of showing what is in each place in the stack for the converted C to assembly code when you are moving backward through the code
BF 2.6 Normal
BF 2.7 Blackbox
BF 2.8 Functional
BF 2.9 Integration
BF 2.10 Results: The back button correctly clicks backward through the stack animation showing the proper code/elements in the stack and follows the reverse order to maintain consistency and accuracy within the stack and registers.

BF 3.1 Skip To End Button Functionality
BF 3.2 This test will ensure that the Skip To End button on the stack home page properly skips to the end of the stack with corresponding animation
BF 3.3 The user will input valid C code into the editor, compile it, and click the skip to end button to see that it moves to the end of the stack animator and is matching with the code
BF 3.4 Inputs: The inputs for this test is correctly compiled C code and the user clicking the skip to end button
BF 3.5 Outputs: The outputs for this test are the proper animations of showing what is in each place at the end of the stack for the converted C to assembly code when you go to the end of the code
BF 3.6 Normal
BF 3.7 Blackbox
BF 3.8 Functional
BF 3.9 Integration
BF 3.10 Results: The skip to end button correctly clicks to the end of the code and shows the proper animation in the stack and updates the values accordingly so nothing is missed for the final product.

BF 4.1 Reset Functionality
BF 4.2 This test will ensure that the Reset button on the stack home page properly clears all information in the stack animation when clicked
BF 4.3 The user will input valid C code into the editor, compile it, and click the next button to start stepping through the animation and after two clicks of next will click the reset button
BF 4.4 Inputs: The inputs for this test is correctly compiled C code, the user clicking the next button twice, and then the user clicking the reset button

BF 4.5 Outputs: The outputs for this test are the proper animations of showing the stack table being cleared once the reset button is clear and starting over on the compiler
BF 4.6 Normal
BF 4.7 Blackbox
BF 4.8 Functional
BF 4.9 Integration
BF 4.10 Results: The skip to end button correctly clicks to the end of the code and shows the proper animation in the stack and the transition is quick and immediate so there is no lag ensuring the user doesn't have to wait or try to click the button multiple times leading to accidentally skipped steps and not being able to follow the animation.

BF 5.1 Call Stack Start Button Functionality
BF 5.2 This test will ensure that the Call Stack Start Button on the Dashboard page correctly navigates the user to the Call Stack Animation page
BF 5.3 The user will start up the Hackademia site and load the home dashboard page and then click on the start button in the call stack module
BF 5.4 Inputs: The inputs for this test is correctly compiled C code, the user clicking the next button twice, and then the user clicking the reset button
BF 5.5 Outputs: The outputs for this test is the user being able to correctly navigate to the call stack module page
BF 5.6 Normal
BF 5.7 Blackbox
BF 5.8 Functional
BF 5.9 Integration
BF 5.10 Results: The start button navigates the user to the stack module home page and the transition is quick and immediate so there is no lag ensuring the user doesn't have to wait or try to click the button multiple times.

BF 6.1 Dashboard Button Functionality
BF 6.2 This test will ensure that the Dashboard Button will correctly navigate to the home page showcasing the different modules
BF 6.3 The user will start up the Hackademia site, navigate to the Call Stack Module, then click on the dashboard button
BF 6.4 Inputs: The inputs for this test is loading the Hackademia site, clicking on the stack module, clicking on the dashboard button
BF 6.5 Outputs: The outputs for this test is the user being able to correctly navigate from a specific module page back to the dashboard home page
BF 6.6 Normal
BF 6.7 Blackbox
BF 6.8 Functional
BF 6.9 Integration

BF 6.10 Results: The dashboard button properly navigated from the specified to the dashboard home screen and the transition is quick and immediate so there is no lag ensuring the user doesn't have to wait or try to click the button multiple times.

PT 1.1 Large C Code Compilation
PT 1.2 This test will evaluate the system's performance when compiling a large C program
PT 1.3 The user will input a large C program with dozens of lines of code and compile it to measure the compilation time
PT 1.4 Inputs: Multiple lined C code
PT 1.5 Outputs: Compiles correctly and measures compilation time, no server timeout
PT 1.6 Boundary
PT 1.7 Blackbox
PT 1.8 Performance
PT 1.9 Integration
PT 1.10 Results: No server timeout errors, compiles in less than 1 second resulting in an extremely responsive program that the rest of the animation and compilation also follow for very user friendly usage.

PT 2.1 High Memory Usage Handling
PT 2.2 This test will evaluate and verify that the system handles cases where the input consumes high memory
PT 2.3 The user inputs a C program with large memory allocation and compile it, check to see if system compiles correctly and doesn't crash
PT 2.4 Inputs: An array that attempts to allocate a very large amount of memory
PT 2.5 Outputs: Successful compilation & memory usage does not exceed acceptable limits
PT 2.6 Boundary
PT 2.7 Whitebox
PT 2.8 Performance
PT 2.9 Integration
PT 2.10 Results: The program compiles correctly and is able to handle the memory load

PT 3.1 Animation Performance Under High Complexity
PT 3.2 This test will evaluate the performance of the stack animation when processing complex C code with nested function calls and large stacks
PT 3.3 The user will input a C program with multiple nested functions and recursion and monitor the animation's performance and handling
PT 3.4 Inputs: Nested recursion functions
PT 3.5 Outputs: Animation reads each function call and returns the step smoothly, no freezing or time lag in the animation, less than 0.5 seconds for each processing step
PT 3.6 Boundary
PT 3.7 Blackbox
PT 3.8 Performance
PT 3.9 Integration

PT 3.10 Results: The animations did not lag and handled the more complex code without issue. After testing with large c code functionality the animations are still working swiftly and not lagging behind when highlighting and updating the stack and register values.

UT 1.1 Edge Case for Empty Functions
UT 1.2 This test will ensure that empty functions are correctly handled by the compiler and visualized in the animation without errors
UT 1.3 The user will input C code with an empty function and compile it to verify the system doesn't produce errors and the stack reflects the empty function
UT 1.4 Inputs: Empty c function
UT 1.5 Outputs: The stack animation shows the empty function
UT 1.6 Boundary
UT 1.7 Blackbox
UT 1.8 Functional
UT 1.9 Unit
UT 1.10 Results: The compiler doesn't spit out errors or crash and the stack animation reflects the empty function. When the stack animation is empty it simply doesn't do anything; it just remains in a hanging state until given further instruction.

UT 2.1 User Input Validation
UT 2.2 This test will verify that the compiler supports proper formatting and prevents input and compilation of invalid characters
UT 2.3 The user tests the editor functionality by typing, adding, deleting, and pasting C code to ensure it handles the interactions properly
UT 2.4 Inputs: Typing unsupported characters in the compiler
UT 2.5 Outputs: Unsupported characters trigger an error message and don't compile
UT 2.6 Abnormal
UT 2.7 Blackbox
UT 2.8 Functional
UT 2.9 Unit
UT 2.10 Results: The compiler correctly triggers the error message and the user cannot properly compile and run the animation. When the user compiles and generates improper assembly code, as it steps through the stack and registers a pop up box with an error message shows up when it hits an unsupported operand or instruction stating following values may be inaccurate.

FE 1.1 Syntax Highlighting Functionality
FE 1.2 The purpose of the test is to verify that the code editor correctly highlights the C code and corresponding assembly as you run through the animation
FE 1.3 The user will input valid C code and compile it while clicking through the animation to check the highlighting of the program
FE 1.4 Inputs: Valid C code, user clicking through the animation
FE 1.5 Outputs: Highlighting of the corresponding C code and assembly lines
FE 1.6 Normal
FE 1.7 Blackbox

FE 1.8 Functional
FE 1.9 Integration
FE 1.10 Results: The animation properly highlights the corresponding C code and assembly when running through the scenario. The user can click through the stack and registers using the navigation buttons at the top and properly shows the corresponding stack and register display for the appropriate steps.

Test Case Matrix

Name	Normal/Abnormal	Blackbox/ Whitebox	Functional/Performance	Unit/integration
VC1	Normal	Blackbox	Functional	Unit
EM1	Normal	Blackbox	Functional	Unit
CC1	Normal	Blackbox	Functional	Integration
BF1	Normal	Blackbox	Functional	Integration
BF2	Normal	Blackbox	Functional	Integration
BF3	Normal	Blackbox	Functional	Integration
BF4	Normal	Blackbox	Functional	Integration
BF5	Normal	Blackbox	Functional	Integration
BF6	Normal	Blackbox	Functional	Integration
PT1	Boundary	Blackbox	Performance	Integration
PT2	Boundary	Whitebox	Performance	Integration
PT3	Boundary	Blackbox	Performance	Integration
UT1	Boundary	Blackbox	Functional	Unit
UT2	Abnormal	Blackbox	Functional	Unit
FE1	Normal	Blackbox	Functional	Integration