Fault Model

Fault #1 (stackOverflow fault):

- The stackOverflow error never gets asserted when more than 8 entries are pushed onto the stack
- The stackOverflow error gets asserted before the stack tries to push more than 8 entries.
- The stackOverflow does not stay asserted until DONE

Test #1: Create a task using the START command to push the first data entry onto the stack then use the ENTER command to enter 7 or more data entries onto the stack and check to ensure the error is asserted at the correct time. We can track that stackOverflow is unasserted while less than 8 entries are on the stack and is asserted/stays asserted until DONE after more than 8 entries are entered onto the stack.

Reasoning: It is possible that the person using the calculator tries to put more than 8 things onto the stack which would cause it to either overwrite previous stack entries or not perform the write at all and we need to make sure stackOverflow is asserted so that the user is aware of this.

Fault #2 (unexpectedDone fault):

- The unexpectedDone signal is not asserted when the DONE command is given and more than one entry is still on the stack
- The unexpectedDone signal is asserted when it should not be
- The unexpectedDone signal is asserted for longer than one cycle

Test #2: Create a task using the START command to push the first data entry onto the stack then enter a couple more random commands and finally enter the DONE command before there is only one entry on the stack. We can track that unexpectedDone is unasserted before DONE and is asserted for only one cycle after DONE command is given earlier.

Reasoning: The unexpectedDone signal could never be asserted even though the DONE command was given earlier than expected. It is also possible that the unexpectedDone is asserted at the incorrect time or for more than one cycle which goes against the specifications.

Fault #3 (dataOverflow fault):

- The dataOverflow signal is not asserted when the add ARITH_OP overflows
- The dataOverflow signal is asserted when the add ARITH_OP does not overflow
- The dataOverflow signal is not asserted when the subtract ARITH_OP overflows

- The dataOverflow signal is asserted when the subtract ARITH_OP does not overflow
- The dataOverflow signal is not asserted until DONE

Test #3: Create a task that will send random numbers onto the stack that do not overflow when added/subtracted together. Then add/subtract them using the add/subtract command and make sure the dataOverflow signal is not asserted. Then send more random numbers that do overflow and make sure the dataOverflow signal is asserted until DONE.

Reasoning: The only two ARITH_OP that can cause a dataOverflow are add and subtract so testing to make sure that when either of these two commands causes an overflow that dataOverflow is asserted and asserted until DONE is important. Also testing to make sure that the dataOverflow signal is not asserted any other time is reasonable since the calculator could mess up and assert it too soon, too late, or for too long even after DONE.

Fault #4 (protocolError fault):

- The protocolError signal is not asserted when there aren't enough items on the stack to do a specified operation
- The protocolError signal is not asserted when START appears again before DONE
- The protocolError signal is not asserted when there are 0 entries on the stack between START and DONE
- The protocolError signal is not asserted when there is an invalid command
- The protocolError signal is asserted even though none of the 4 cases that can assert it happens
- The protocolError signal is not asserted until DONE

Test #4:

- Create a task that uses the START command to put one entry onto the stack then perform all of the ARITH_OP that require two elements and see if the protocolError is asserted until DONE every time for the different cases
- Create a task that uses the START command once in the beginning and randomly another time in the sequence before DONE is asserted and make sure the error is asserted until DONE
- Create a task that uses the START command then the pop ARITH_OP to remove the one entry before DONE to see if the error is asserted until DONE
- Create a task that tries to put in a couple of random invalid commands to see if the error is asserted until DONE
- Create a task that sends in random valid commands after the START command and make sure that the error never gets asserted when none of the 4 cases happen

Reasoning: The overall reasoning for these tests is to test each of the 4 cases that will assert the protocolError and make sure that whenever any of the cases happen the protocolError should be asserted until DONE and is unasserted otherwise. If the error gets asserted for any of the 4 cases then we know the signal works as expected even when there are a combination of different protocol errors.

Fault #5 (correct fault):

- The correct signal is asserted when DONE is not on the input
- The correct signal is asserted when there are errors asserted
- The correct signal is not asserted when DONE is on input and there are no errors

Test #5: Create a task that sends a sequence of random commands after START without any errors and when DONE is used make sure the correct signal is asserted and was not asserted any time before DONE. Then try to send in a sequence of random commands after START with errors and make sure correct is not asserted.

Reasoning: Just like with many of the other signals it is possible that the correct signal is not asserted at the correct time or is not asserted at all so this test will help to ensure that the correct signal is asserted when it should be and unasserted when it shouldn't be.

Fault #6 (finish fault):

- The finished signal is not asserted when DONE is on the input
- The finished signal is asserted when DONE is not on the input

Test #6: Create a task that sends a sequence of random commands after START without errors and DONE is given to see if finished is asserted and is unasserted the whole time when DONE is not given. Then do the same thing but with a sequence that has errors and make sure the finish is asserted when DONE is given and unasserted the whole time when DONE is not given.

Reasoning: Similarly to the correct signal, it is possible that the finish signal is asserted at the wrong time or is not asserted at all so this test will ensure that finish is asserted/unasserted when it is supposed to be.

Faults #7, #8, and #9 (add, subtract, and add faults):

- The add/subtract/and operation could calculate the result of the two numbers incorrectly
- The add/subtract/and operation could calculate other elements together that are not the top two entries on the stack

- The result of the add/subtract/and could not have been added to the top of the stack
- The add/subtract/and operation keeps the top two entries on the stack with the result even though they should have been taken off after the add
- The stack size does not decrease by 1 after said operation

Test #7, #8, and #9: Create a task that puts random entries onto the stack using ENTER. Then, perform the add/subtract/and operation and check the stack to make sure the correct result is on top of it, the two entries operated on have been removed from the stack, and none of the other entries in the stack have been affected.

Reasoning: Using these tests to check the add/subtract/and operation will allow me to keep track of what I expect to be on the stack versus what the calculator does to the stack due to the add operation. So I know what the result should be and where the result should be stored on the stack meaning the calculator's stack should match that expectation if it works as intended.

Fault #10 (swap fault):

- The swap operation swaps the two entries incorrectly
- The swap operation causes the stack not to remain the same size or changes entries other than the top two

Test #10: Create a task that uses the START and ENTER command to put random values onto the stack then use the swap operation to swap the top two elements and check that they are put in their correct positions and values are still the same.

Reasoning: The only fault that reasonably could happen with the swap operation is the entries that are swapped do not get put in the correct places which is being tested by this test.

Fault #11 (negate fault):

- The negate operation did not perform the 2's complement on the top entry correctly
- The negate operation causes the stack not to remain the same size or changes entries other than the top one

Test #11: Create a task that uses START to put a random entry onto the stack and then run the negate operation and check that the result is expected while keeping other stuff on the stack the same.

Reasoning: Similarly to the swap operation, the only fault that reasonably could happen with the negate operation is the entry that is negated is not correct and does not use the 2's complement negation.

Fault #12 (pop fault): The pop operation does not remove anything from the stack or removes more than one thing from the stack.

Test #12: Create a task that pushes three random entries onto the stack then call the pop operation and make sure that only the top entry is removed from the stack.

Reasoning: It is possible that the pop operation is not removing anything from the stack or it is faulty where it accidentally removes more than one thing from the stack which is not intended.

Fault #13 (START fault): The START command could not add the starting value onto the stack or could add more than one entry to the stack.

Test #13: Create a task that uses the START command to put a random entry onto the stack then check that the value given is on top of the stack.

Reasoning: The calculator could be ignoring the START command which causes it to either not start at all or forget to put the initial value onto the stack.

Fault #14 (ENTER fault): The ENTER command does not add a value to the stack or could add more than one entry onto the stack.

Test #14: Create a task that uses the ENTER command after a START command to put a random entry onto the stack then check that the value given is on top of the stack.

Reasoning: The calculator's ENTER command could be faulty or is being ignored which could cause it to not put the values onto the stack.

Fault #15 (DONE fault): The DONE command should be stopping the calculator but does not

Test #15: Create a task that uses the START command to put a random entry onto the stack then use the DONE command and try entering other random commands to check if the calculator is ignoring them.

Reasoning: The calculator's DONE command could be allowing other inputs to be put into the calculator and onto the stack even though it should have stopped the calculator.

Observations Found

Phases:	Errors:
0	The calculator was not faulty.
1	The protocolError would be asserted when any of the 4 cases happened. However, the protocol error would not be asserted until DONE like it was supposed to. It would only get asserted for one clock cycle and then deassert.
2	When performing the subtraction operation, the calculator would perform the subtraction incorrectly and the result would be off by one.
3	When performing a calculator sequence, the finished signal would never get asserted for any sequence even when the DONE command was given.
4	● When performing the add or subtract operation on any two entries that would cause overflow the dataOverflow signal was never asserted. ● When performing the swap operation when there is only one entry on the stack the protocolError would get asserted but it would get asserted one cycle later than it should have been. ● In addition, when actually performing the swap operation when there are two elements on the stack the calculator does not swap the two elements it just ignores the operation for any two entries.
5	● When constantly entering things into the calculator's stack until the number of entries was above 8 the stackOverflow signal would get asserted but it would always get asserted one cycle earlier than it should have been asserted. I am assuming the calculator asserts it when the stack size is equal to 8

	instead of above 8. ● When performing any operation that requires more elements than there are currently on the stack the protocolError is never asserted. ● In addition, the protocolError is also not asserted when there are zero items on the stack between START and DONE
6	Could not find any errors. I think this phase does not have any faults.
7	When the START command was given twice before the DONE command was given the protocolError was never asserted.