**Basic Software Debug**

Debugging the design using the test software is a key function for all of the Phases. This document provides some recommendations about how to do this.

1. **Step and Observe Registers**

The main way to debug using software is the following (using phase2\_test as an example):

1. Run the test and determine which case fails. In phase2\_test this is easy because of the structure of incrementing x10.
2. Step to a few instructions before the failure and open the Register display window. I suggest you set the display format to hex for all registers because that will match the test values more closely, although leaving x10 as decimal will better correlate to the case numbers.
3. Determine which register should change on the instruction you are on, and what its new value should be. If the instruction is a branch or jump, determine where the next instruction should be.
4. Step the simulation, and observe which register changes or where the branch goes. If this is correct, step again until you see an unexpected behavior.
5. Once unexpected behavior is observed, the instruction which was executed is likely to be wrong.
6. **Step and Observe Memory**

If the failure occurs while testing the memory operations, open a Memory Monitor as described in the Memory Debug document.

1. Step to the first memory operation (which should be a store) and determine the address where the data should be stored and what the data should be (based on the register values in the Register display).
2. Step the store operation, and see if the correct memory location changes to the correct value. If it is wrong, either the store didn’t happen or it was to the wrong address.
3. Since incorrectly creating an address of 0 is a common problem, look at memory location 0 and see if it contains the value you are trying to write. If it does, the address calculation is wrong. You could also look through memory and see if the data has been written somewhere, but that is unlikely to be valuable at this point.
4. If the correct memory location changes but the value does not match the register you are writing, there is an error in the implementation of store.
5. If the correct value is written to the correct location, the store is likely to be right.
6. There will typically be a load instruction from the same location. Step to that point.
7. Step and observe the register selected by the RD field of load change.
8. If it changes to the correct value, the load is likely to be correct.
9. If it changes to the wrong value which is related to the correct value, the implementation of the data part of load is likely to be wrong.
10. If it changes to the wrong value which is unrelated to the correct value, the address calculation of load is likely to be wrong. If the value loaded is the value in memory location 0, that should make the address error obvious.
11. If the register doesn’t change, there is a basic error in the load implementation.
12. **Use the Disassembler**

Codasip builds a disassembler when the Disassembler (ia) operation is executed in the Task Window. Once a simulation is started, click the i-> button in the Debug Perspective ribbon, as shown in Figure 1. This will bring up the Disassembler window shown in Figure 2.

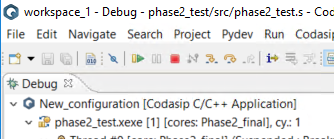


Figure 1

The Disassembler window shows the actual instruction in each memory location, and the green highlight will match the green highlight in the test program source. This makes it easy to see where branch and jump instructions should go, and presents each pseudo instruction as the actual instruction.

1. **Look at the Object Code in Memory**

One problem with Disassembly is that if the hardware code is wrong, the Assembler and the Disassembler will have the same error and the Disassembler may not show the real instruction in memory. It does show the address of each instruction, so you can go to the memory at that address and see the exact instruction generated by the assembler. Note that the memory is in Little Endian order, which must be reversed to see the real value. In order to see the code, open a new Memory Rendering and select standardname2:as\_code and select the address 0. For example, in phase2\_test the data at Instruction Memory location 0 is 0x13051000 little endian. The actual value of the instruction is 0x00100513. This is the assembly instruction:

addi x10, x0, 1

which is the first instruction of the test. Use the RISCV 2.2 Specification instruction sheet to help decode instructions.

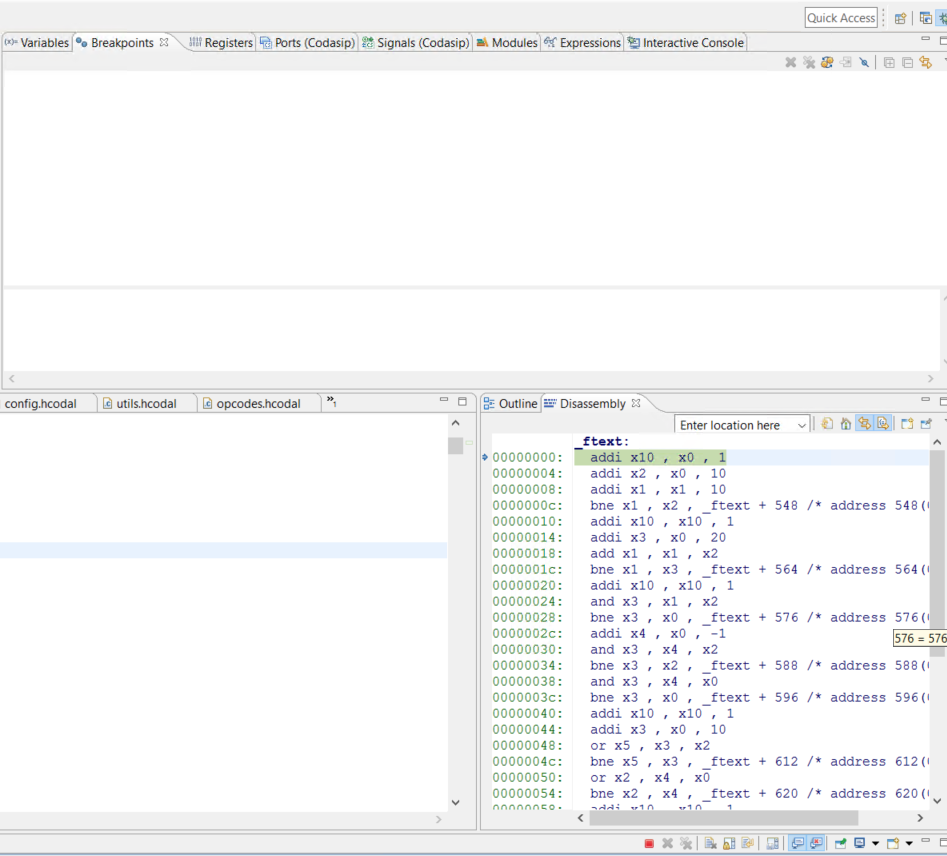


Figure 2

1. **Resetting the Design**

Codasip has the habit of saving all configuration information, which can lead to the case where an accidental error can get it into a bad state. If you see strange behavior (for example, you change the design and build but the result isn’t any different), there are a few easy tricks to recover.

1. Select the project (hardware or software) in the Project Explorer window, right click and select Clean project. You will need to rebuild that project.
2. Select the project (hardware or software) in the Project Explorer window, right click and select Index -> Rebuild. You will need to rebuild the project.
3. Select the project (hardware or software) in the Project Explorer window, right click and select Index -> Freshen all files. You will need to rebuild the project.
4. Select the project (hardware or software) in the Project Explorer window, right click and select Refresh. You will need to rebuild that project.
5. If all else fails, close Codasip and restart it. It will ask you to save files first so nothing will be lost.
6. **Building While Simulating**

A variety of unusual errors which occur during a build (hardware or software) may be caused by attempting to build while a simulation is running. If this happens, go to the Debug Perspective and hit the red square. Figure 3 shows a typical error if you try to build a test when a simulation is running. Figure 4 shows a typical error if you try to build the hardware while a simulation is running.

A screenshot of a social media post

Description automatically generated

Figure 3

A screenshot of a computer

Description automatically generated

Figure 4

You may also accidentally have multiple simulations running, which will be shown in the Debug window as shown in Figure 5. This is almost always a bad thing. To clean this up, select the \*.xexe line in one of the sims and hit the red square. Do that for all the sims. Then hit the double gray X symbol at the upper right – this will clear out everything.

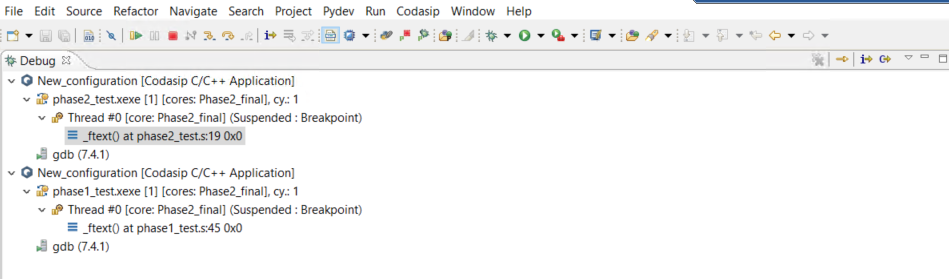


Figure 5