μCore/μForth is a hardware/software co-design environment

You must have docker installed, which may either run under Linux, MacOS, or Windows10.

docker pull microcore/gforth_062 will pull a gforth_0.6.2 system into your host (https://hub.docker.com/r/microcore/gforth_062). It is the basis of μForth, the assembler/cross compiler, OS, and debugger for μCore. Please note that the docker image requires an RS232 serial interface.

μCore is written in VHDL and the code is available on github.com. During development, ModelSim has been used for simulation and Synplify for synthesis.

git clone https://github.com/microCore-VHDL/microCore will initially pull the basic μCore kernel without application specific additions. Documentation on μCore is available in the **documents** directory.

git clone https://github.com/microCore-VHDL/LFXP2_8_protoboard will initially pull a customized version for a simple prototyping board around the LFXP2-8E FPGA from Lattice. This is meant as a model on how to use μCore in applications. Documentation on the prototyping board is available in the **documents** directory.

Both projects have the same subdirectory structure:

`<project>`	
`documents`	Documentation
`<technology>`	Only present for lfxp2_8_protoboard.
`modelSim`	Simulator config file and wave display scripts for various tests
`work`	'work' library for the simulator's compiled μCore code
`software`	μForth cross compiler and basic sim/load files
`test`	Additional load/sim files. Copy the ones to use into 'software'.
`synplify`	Synthesizer onfiguration files for synplify
`vhdl`	VHDL files of μCore

Please note that this structure is different from what the FPGA vendor specific design environments would like to see. Unfortunately, they all expect something different. Therefore, I try to abandon the use of the environments, just using the simulator, the synthesizer, and the FPGA-specific P&R suite as stand alone software. If this is not possible, the vendor's design suite is installed under <**technology**>.

In the following discussion, a Linux system is assumed as host OS.

To get started, μCore's instruction set test **./software/coretest.fs** will be executed assuming ModelSim on Linux and the `/dev/ttyUSB0` serial port. When using a different OS or serial interface, both **./software/umbilical.fs** and the bin/bash file **./gforth_062.sh** have to be modified accordingly.

1. At first, **./software/sim_core.fs** must be cross compiled. To this end, open a terminal in the microCore directory. Start the docker image with **`./gforth_062.sh`**. This should display gforth's signon message if all goes well. If not, make sure that you did specify a valid serial port in the .sh-script.
2. **`include sim_core.fs<cr>`** will produce an object code file for initializing μCore's program memory at the start of simulation. A couple of information lines will be displayed and the last line should read **`sim_core.fs written to program.mem  ok`**.

3. `cd` into **./modelSim** and start modelSim by double-clicking **microCore.mpf.** This will open the project display. Don't forget to "Suppress Warnings: [ ] From IEEE Numeric Std Packages (Pull-down 'Simulate' -> 'Runtime Options...') or else your Transcript screen will be flooded with irrelevant warning messages during simulation.
4. Create a **work** library by clicking 'File' -> 'New' -> 'Library'. Select "a new library and a logical mapping to it" and enter **work** for both 'Library Name:' and 'Library Physical Name:'.
5. 'Compile' -> 'Compile All' will compile the μCore design. It should produce one warning and no errors. (Perhaps your ModelSim version needs different initialization files in the simulator/work directory.)
6. Double click 'simulation' in the project tab. This will, amongst others, open the wave window.
7. In the wave window, click on the 'file' pull down menue and select 'load', which will open the **modelSim** directory. Double click **core.do**. This will select a set of basic μCore signals for the wave window.
8. Simulate for 320 usec and take a look at the right end of the wave display. If all went well, signal **bitout** will be high and signal **r.tos** will toggle between $FFFFFFFE and 0.

   ```
   # initializing ../software/program.mem
   # starting core test ...
   # ** Note: core test ok
   #    Time: 309300 ns  Iteration: 4  Instance: /bench
   ```

   If the last two message lines will not show, you either did not simulate long enough (320 us), or there was an error. Then **bitout** will be low and a number different from 0 in **r.tos** indicates the error, which you can search for in **coretest.fs**.