Using ModelSim Foreign Language Interface for c – VHDL Co-Simulation and for Simulator Control on Linux x86 Platform

Andre Pool - fli@andrepool.com - Version 1.6 - created November 2012, last update March 2014

Introduction

Writing testbenches in VHDL can be very cumbersome. This can be solved by using a programming language with more features that does not need to bother about hardware implementation restrictions. This project demonstrates how *plain* c or can be used for testing. Besides generating Stimuli and Analyze results, optional features, like a control interface and simulation accelerators, have been added to this testbench environment.

Target

This project has been created to show hardware designers how easy it is to use c for testing their Design Under Test (DUT). Besides writing tests in c is much easier, also the simulation time can be reduced significantly. The project is very simple, so the reader can focus on the flow, however a basic knowledge of VHDL, c, Makefiles and ModelSim is required.

Challenge

Using c to generate stimuli and evaluate results is quite easy, but there are two problems when using software and VHDL (RTL) together:

- interfacing between the different worlds
- time awareness in software

Solution

Fortunately Model Technology (now Mentor Graphics) recognized the need for interfacing these different worlds and defined the *proprietary* Foreign Language Interface (FLI) [1].
The second issue can be solved by using the VHDL level to generate the clock and use this clock to trigger the c functions through this FLI interface.

Short description FLI

In a very basic use case one could define an Entity declaration with an Architecture in VHDL. Normally the Architecture would contain the functionality but in this case it only contains a link to a *foreign* c file. This *foreign* c file contains the desired functionality including a sensitivity list. Besides this *foreign* architectures also a number of simulator controls can accessed through the FLI interface, including the mti_ForceSignal and TCL commands, but unfortunately **not** the *run* command.

Simulation Data Flow

Data is generated by the Stimuli Generator, this data is used for the input signals on the DUT (from which the behavior has to be verified). The output signals on the DUT are used by the Data Analyzer validating the DUT behavior. If needed a feedback can be used to adjust the input signals to the DUT depending on the output signals from the DUT.

Foreign Architecture (optional)

In cases where parts of the simulation environment are consuming a significant amount of the simulation time, and if these parts are only required to support the verification process, these parts could be moved from VHDL to c. Then these parts become Foreign Architecture (FA) blocks. The blocks make use of the FLI to interface with the simulator.
An example of such FA block could be a processor that configures and monitors some registers.
The FA block is still sequential, meaning the simulator waits for the FA block and the FA block waits for the simulator. Be careful with *concurrency* between the FA blocks and other blocks.

Separate Process Threads (optional)

The next step in optimization is looking for (FA) blocks, or modify (FA) blocks, that can be more or less run independent from each other and use some kind of handshaking to synchronize. In the simplest form if your FA block is only sensitive to the clock and your design uses only one clock edge, then the simulator can send the information on the active edge of the clock from the simulator to the separate process thread (SPT) and continue with simulation without waiting for the SPT to finish. At the non active clock edge the simulator requires the results from the SPT thread, only when the (probably much faster) SPT was not ready, the simulator has to wait, otherwise it can just continue. This can be very efficient on a multi (processor) core system because a number of such SPTs can run in parallel with each other and in parallel with the simulator (checkout with: top and or pstree -a <process_id>).

Controlling the Environment (optional)

Once the simulator environment is up and running it is often required that you need to check the status and or change variables in the running environment. For this a server interface has been added to the simulation environment. The server runs also as an SPT. A simple API has been defined which covers most used commands.

Control Application (optional)

The control application is a separate process (on the same system) that will connect to the server of the simulation environment. The control application does a number of changes and checks on the running simulation environment (Simulator, Stimuli Generator, Data Analyzer, FA and SPT). Furthermore it also excepts tcl commands as arguments that will be forwarded to the Simulator. The control application has been implemented in as well c as python.

Diagram

The next figure shows how the blocks are related to each other

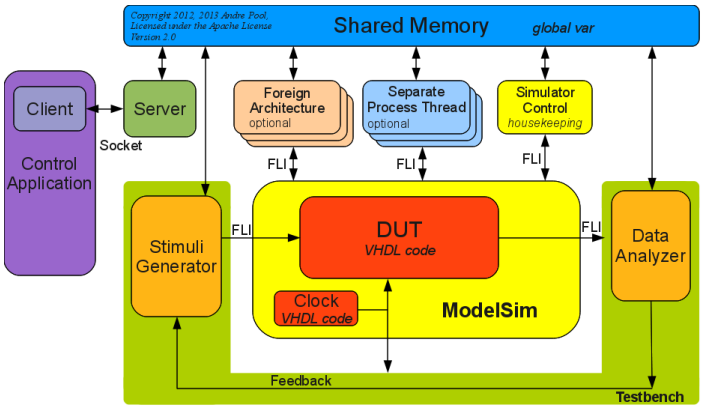

Figure 1. c – VHDL Co-Simulation diagram using FLI.

Get Going

Step 1 get the code:

```
git clone git://github.com/andrepool/fli.git

(or https://github.com/andrepool/fli/archive/master.zip)
```

Step 2 check if the simulator environment settings are correct

```
MTI_HOME (e.g. export MTI_HOME=/opt/modeltech/v10.2)
PATH (e.g. export PATH=/opt/modeltech/v10.2/bin:$PATH)
```

Step 3 start the simulation environment

```
cd fli/rtl
make clean
make
```

The simulator workspace with the wave window should pop up.

Using ModelSim Foreign Language Interface for c – VHDL Co-Simulation and for Simulator Control on Linux x86 Platform

Andre Pool - fli@andrepool.com - Version 1.6 - created November 2012, last update March 2014

```
andre@zaphod:~/proj/fli/rtl$ make
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o testbench.o -c
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o counter.o -c c
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o sqrt_int.o -c
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o global_var.o -
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o housekeeping.c
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o server.o -c se
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o sock_functions
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../socket -I$MTI_HOME/include -o server_functic
gcc -shared -Wl,-Bsymbolic -Wl,-export-dynamic -std=c99 -m32 -o c_environment.so testbench.o counter.o sqrt_int.o global_var.o housekeep
vcom -2002 -O5 -pedanticerrors -quiet testbench.vhd
vcom -2002 -O5 -pedanticerrors -quiet counter.vhd
vcom -2002 -O5 -pedanticerrors -quiet sqrt_int.vhd
vcom -2002 -O5 -pedanticerrors -quiet comparator.vhd
vcom -2002 -O5 -pedanticerrors -quiet ram.vhd
vcom -2002 -O5 -pedanticerrors -quiet dut.vhd
vcom -2002 -O5 -pedanticerrors -quiet top.vhd
Reading /mtitcl/vsim/pref.tcl
```

Figure 2. make rtl log output.

Step 4 control application connect to simulation environment and perform actions

open another terminal

```
cd fli/socket
make
```

This will demonstrate the interaction between the control application and simulation environment. One of things you should notice that cursor in the wave window is controlled from the "external" control application.

```
andre@zaphod:~/proj/fli/socket$ make
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../src -I$MTI_HOME/include -o control_application
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -I. -I../src -I$MTI_HOME/include -o sock_functions.o -c
gcc -g -O2 -Wall -ansi -fms-extensions -std=c99 -pedantic -m32 -freg-struct-return -o control_application control_application.o sock_funct
INFO    server API version 10
INFO    buffer size 0,524 Mbytes
INFO    mti version: 'ModelSim For Questa  Version 10.2 2013.02'
INFO    current count value  260 current sqrt value  13
INFO    get sum 12 from server, after sum + 1 and inc, sum gets 14
INFO    mti cmd 'date' returns 'Mon Sep  9 18:50:55 CEST 2013'
INFO    mti cmd 'pwd' returns '/home/andre/git/proj/fli/rtl'
INFO    File            Library    Hierarchy  Instance   Entity          Architecture
INFO     top.vhd         work          *        top        top             sim
INFO     testbench.vhd   work          *        t0         testbench       c_model
INFO     dut.vhd         work          *        d0         dut             rtl
INFO     counter.vhd     work          *        c0         counter         c_model
INFO     sqrt_int.vhd    work          *        s0         sqrt_int        c_model
INFO     comparator.vhd  work          *        p0         comparator      rtl
INFO     ram.vhd         work          *        m0         ram             rtl
INFO    iteration           0, current time          78,934,140 ns
INFO    iteration           1, current time          79,028,585 ns
INFO    iteration           2, current time          79,067,485 ns
INFO    iteration           3, current time          79,103,340 ns
INFO    iteration           4, current time          79,130,345 ns
INFO    iteration           5, current time          79,167,645 ns
INFO    iteration           6, current time          79,207,825 ns
INFO    iteration           7, current time          79,237,245 ns
INFO    iteration           8, current time          79,284,445 ns
INFO    iteration           9, current time          79,333,335 ns
INFO    client all done connecting 10 times to the server
INFO    done with testing, send one last message to server
andre@zaphod:~/proj/fli/socket$ make quit
INFO    control application request to quit simulator
INFO    response 'simulator has stopped'
andre@zaphod:~/proj/fli/socket$
```

Figure 3. make socket / control application log output.

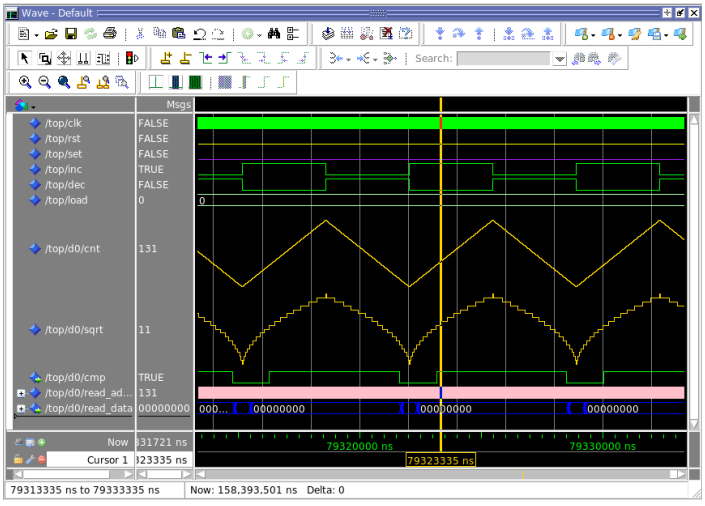

Figure 4. ModelSim Simulator Wave window.

Step 5 a series of tcl commands

./control_application -p "hello world"

```
./control_application -m "force /top/d0/cnt 8"
./control_application -w
./control_application -m "noforce /top/d0/cnt"
./control_application -w
```

This will print a message in the transcript window and forces a signal by sending tcl commands to the simulator and show this change in the wave window.

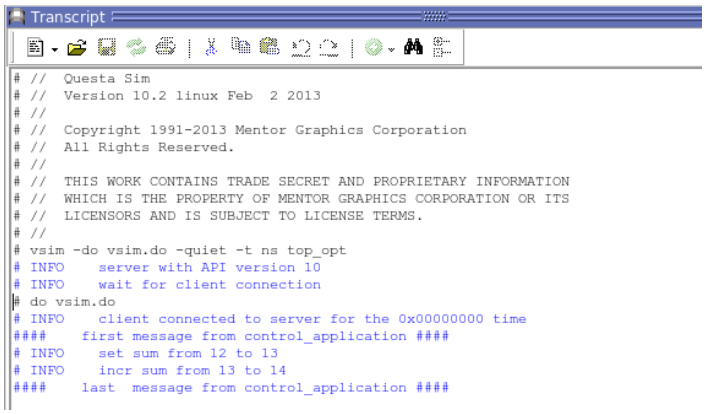

Figure 5. ModelSim Simulator transcript window.

Step 6 shutdown the simulation environment

```
make quit
```

Because simulator keeps on running, it is not possible to type a command in the transcript window, however you can stop the simulator by sending a break (e.g. make break), type the commands in the transcript window followed by "run -all" and you can continue using the control application through the socket interface.

Project directory structure

```
fli/docs
     fli_c_vhdl_cosimulation.pdf
     fli_c_vhdl_cosimulation.odt (openoffice)
     fli_c_vhdl_cosimulation_diagram.odg (openoffice)
     fli_c_vhdl_process_flow.odg (process relations)
     todo.txt (known issues / features)

fli/rtl
     Makefile (build and start simulation environment)
     top.vhd (including clock generator)
     testbench.vhd (stimuli / analyzer)
     dut.vhd (counter->sqrt->comparator)
     counter.vhd (vhdl or c model)
     sqtr_int.vhd (square root entity pointing to c model)
     comparator.vhd (comparator in vhdl)
     vsim.do (simulator startup script with wave)
     vsimc.do (command line simulator script, no wave)

fli/src
     Makefile (build shared object for the simulator)
     global_var.c (shared memory used by all functions)
     testbench.c (stimuli and analyzer)
     "Foreign Architectures"
        sqrt_int.c (c fuction square root)
        counter.c (optional c model for counter.vhd)
        housekeeping.c (c function to control simulator)
     "Separate Process Threads"
        server.c (environment control interface)

fli/socket
     Makefile (build control application and test server)
     sock_config.h (socket configuration file)
     sock_functions.c (generic AF-UNIX socket functions)
     client_functions.c (client socket functions)
     server_functions.c (server socket_functions)
     control_application.c (externally control environment)
     only for testing client server interface
        test_server.c (client server test)
        test_server_functions.c (client server test)
```

Using ModelSim Foreign Language Interface for c – VHDL Co-Simulation and for Simulator Control on Linux x86 Platform

Andre Pool - fli@andrepool.com - Version 1.6 - created November 2012, last update March 2014

Flow

This paragraph shows how a c-file is linked with the Simulator

c-file example (counter.c)

```
#include "mti.h"
#include "global_var.h"
#include <stdio.h>

// create one struct that contains all vhdl signals that
// need to be passed to the function
typedef struct
{
   mtiSignalIdT clk;
   mtiSignalIdT rst;
   mtiSignalIdT set;
   mtiSignalIdT inc;
   mtiSignalIdT dec;
   mtiSignalIdT load;
   mtiDriverIdT cnt;
} counter_t;

// the process function that will be called on each event
// (in this case only clk)
static void counter( void *param )
{
   // connect function argument to counter struct
   counter_t * ip = (counter_t *) param;

   // get current values from the vhdl world
   _Bool clk = (_Bool) mti_GetSignalValue ( ip->clk );
   _Bool rst = (_Bool) mti_GetSignalValue ( ip->rst );
   _Bool set = (_Bool) mti_GetSignalValue ( ip->set );
   _Bool inc = (_Bool) mti_GetSignalValue ( ip->inc );
   _Bool dec = (_Bool) mti_GetSignalValue ( ip->dec );
   mtiInt32T load = mti_GetSignalValue (ip->load );

   // implement the counter functionality
   static mtiInt32T cnt = 7; // initial value
   if( clk )
   {
      if( rst )
      {
         cnt = 0;
      }
      else if( set )
      {
         cnt = load;
      }
      else if( inc )
      {
         cnt++;
      }
      else if( dec )
      {
         cnt--;
      }
   }

   // send value cnt back to vhdl world with 1 ns delay
   mti_ScheduleDriver( ip->cnt, cnt, 1, MTI_INERTIAL );
}

// c initialization function
void counter_init(
   mtiRegionIdT region, // location in the design
   char *parameters, // from vhdl world (not used)
   mtiInterfaceListT *generics, // from vhdl world (not used)
   mtiInterfaceListT *ports // linked list of ports
)
{
   // create a struct to store a link for each vhdl signal
   counter_t *ip = (counter_t *)mti_Malloc(sizeof(counter_t));

   // map input signals (from vhdl world) to struct
   ip->clk = mti_FindPort( ports, "clk" );
   ip->rst = mti_FindPort( ports, "rst" );
   ip->set = mti_FindPort( ports, "set" );
   ip->inc = mti_FindPort( ports, "inc" );
   ip->dec = mti_FindPort( ports, "dec" );
   ip->load = mti_FindPort( ports, "load" );

   // map "cnt" output signal (to vhdl world) to struct
   ip->cnt = mti_CreateDriver( mti_FindPort( ports, "cnt" ) );

   // create "counter" process with a link to all vhdl signals
   // where the links to the vhdl signals are in the struct
   mtiProcessIdT process_id = mti_CreateProcess( "counter_p",
counter, ip );

   // trigger "counter" process when event on vhdl signal clk
   mti_Sensitize( process_id, ip->clk, MTI_EVENT);
}
```

Compile (-I points to directory where mti.h file is located)

```
gcc -I$MTI_HOME/include -o counter.o -c counter.c
```

Link create one shared object from all .o files

```
gcc -shared -o c_environment.so xxx.o counter.o yyy.o ...
```

Connect shared object in VHDL world

```
entity counter is
   port(
      clk : in boolean;
      rst : in boolean;
      set : in boolean;
      inc : in boolean;
      dec : in boolean;
      load : in integer;
      cnt : out integer := 0
   );
end;

architecture c_model of counter is
   attribute foreign : string;
   attribute foreign of c_model :
      architecture is "counter_init ../src/c_environment.so";
      -- counter_init is called in c_environment.so
begin
   -- architecture function in c model
end;
```

For more information about compiling and linking see chapter Compiling and Linking FLI C Applications in [1] and for more examples check the ModelSim examples in [3].

Client Server Interface

A very simple protocol is used to communicate between client and server. The server only responds when it receives a packet from the client, and the client only expects a packet from the server when it has send a packet itself.

Structure of a packet (same for client and server)

```
cmd_resp (command/response one of the list, see below)
size (total packet size in bytes)
addr (32 bit, start address)
payload (to transfer the data c8,u8,u16,u32,u64)
```

A few commands from command response list:
(see sock_config.h for full list)

```
API_GET → version check between server and client
DISCONNECT → client disconnects from server
MTI_BREAK → send break command to simulator
MTI_CMD → run tcl command in simulator
MTI_QUIT → quit simulator (and environment)
OKAY → server response to client correct
PAYLOAD_READ → read data block from the environment
PAYLOAD_WRITE → write data block to the environment
TIME_GET_NOW → current time in the simulator
TRANSCRIPT_PRINT → print in simulator transcript window
```

Questions and Answers

Q. What about Apache License Version 2.0?

A. You can do pretty much anything with this project, and in contradiction to GPL, you do not need to publish other files linked with this project.

Q. Does this project also work with QuestaSim?

A. Yes it does.

Q. Why does c simulate way faster than VHDL?

A. Mainly because it does not need to check for all VHDL/RTL

restrictions and each bit can only be true or false. And quite often you can reduce overhead by mapping your signals to the native 32 or 64 bit system type.

Q. Which parts I should do in VHDL and which in c?
A. Try to move as much as possible parts that do not to be verified to c, but keep in mind that it is even more important that you keep a *clean* and *logical* design structure, which can be understood by somebody else. Also be careful with concurrency.
Q. Can I use this for hardware in the loop / hardware software co-simulation.
Y. Yes you can use an FPGA board for hardware in the loop testing, but you need to create a data mapping interface in the FPGA, connecting to the part you want to accelerate, an interface to the x86, e.g. PCI express, USB or Ethernet and a driver from which you can use the created data mapping. With this driver you should be able to access the hardware through an FA or SPT.
Q. Can I have a DUT written in Verilog rather than VHDL?
A. Yes you can instantiate the Verilog code inside the VHDL top level. However in that case you need a mixed simulation license. Another approach would be to use PLI instead of FLI.
Q. What about Programming Language Interface (PLI)?
A. PLI is more or less the same as FLI, but it is intended for Verilog, I choose for FLI because most of my projects are VHDL. An advantage of PLI is that it is a standard, so it can be used with different simulators (FLI is Model Technology proprietary).
Q. And what about Direct Programming Interface (DPI)?
A. DPI is a sort of simplified PLI, however DPI does not provide direct access to the internals of a simulation data structure [2].
Q. Why are AF UNIX (file handle) sockets used for the client server connection and not AF INET (Internet) sockets?
A. This saves the port management overhead when running multiple simulators (with all their own server port) on one workstation, furthermore it is more efficient because there is no TCP/IP overhead with AF UNIX sockets.
Q. Can multiple clients connect to the server?
A. No only one client (control application) can connect to the server, however if you want to you can modify or duplicate the server.
Q. I want to create a control application in a different language then c, e.g. Python, Perl or TCL, is that possible?
A. A Python and TCL control application have already been included, the Python application directly connects to the server (through AF UNIX sockets) while the TCL application indirectly connects to the simulator through the c control application.
Q. What about code coverage?
A. Sorry that is beyond the scope of this project.
Q. On what platform does this project work?
A. Only Linux x86 (32 bit) and Linux x86_64, because the project makes use of pthreads, sockets and some compiler options for the simulator, it will not work on other platforms.
Q. Does this work with ModelSim Altera Starter Edition, ModelSim PE, or ModelSim DE?
A. No, FLI is only supported on ModelSim SE and QuestaSim.
Q. So why did you not use boost or Qt to make the project platform independent?
A. That would make things more complicated to understand and add more dependencies.
Q. Why did you use c instead of c++?
A. Model Technology FLI functions are also in c and I wanted to keep the project transparent, you are free to add c++ functions, checkout chapter “Compiling and linking FLI c++ applications” in [1].
Q. I have Linux distribution x, Linux kernel version y and ModelSim version z, does this project work on my system?
A. Both the ModelSim FLI and Linux functions used in this project are around for a while and as far as I know the interfaces of the functions did not change for a long time, so I would be surprised if it did not work on your system.
Q. Do I need multiple ModelSim licenses when I use these Separate Process Threads to speedup my simulation?
A. No, that is one of the main advantages, you can speedup simulation, but still require only one ModelSim license.
Q. Why is little endian used for client server communication?
A. To save conversion overhead by keeping the default endianness of x86 and x86-64.

References

[1] ModelSim Foreign Language Interface Reference $MTI_HOME/docs/pdfdocs/*_fli.pdf
[2] http://sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf
[3] $MTI_HOME/examples/vhdl/foreign
[4] Model Sim / Questa Sim User's Manual $MTI_HOME/docs/pdfdocs/*_sim_user.pdf