Co-Simulation of Hardware and Software

XAPP 087 July 25, 1997(Version 1.0)

Application Note by Douglas M Grant

Summary

It is possible to implement an entire hardware - software co-design based around the XC6000DS development system. This applications note describes a method to allow simulation of the hardware part of the design using the application code as the test bench. This allows the application to be functionally debugged with the minimum of effort and the maximum of confidence, before proceeding with placement and routing of the hardware part on a XC6200 FPGA.

Xilinx Family

- XC6200

Demonstrates

- ModelTech
- Co-simulation
- VHDL
- Hardware/Software Co-design
- XC6000DS.

Table of Contents

Introduction:

Capturing a design in VHDL is one thing: Writing the VHDL testbench is quite another, taking up to 50% of the total design-time. If the design is to form part of a hardware-software co-design, then the testbench needs to mirror the action of the software part of the co-design. This effectively means that the software part will need to be written twice - Once in VHDL and once in the actual coding language for the application.

The XC6000 Development System allows software to communicate directly with registers and gates in a design that has been mapped onto a XC6200 RPU (Reconfigurable Processing Unit.) It makes much more sense to use this software as the testbench for the hardware design, rather than to create a testbench in VHDL. In this way the complete application, consisting of hardware AND software can be tested.

Where there is high confidence in the correctness of the hardware, it is best to simply place and route the hardware and then run the application code, using a software debugger to step through the working of the design. It is possible, however, that there is a desire to simulate the design in a VHDL simulator. If the application software is available then this should

again be used as the testbench, but this has until recently not been possible.

This applications note describes a method to allow simulation of the hardware part of the design using the application code as the test bench. This allows the application to be functionally debugged with the minimum of effort and the maximum of confidence, before proceeding with placement and routing of the hardware part on a XC6200 FPGA. For the purposes of this note, the 'hardware' is the XC6200 Development System, although it could be user-defined platforms.

The XC6000 Development System

The XC6000 Development system comprises some low-level software and a PCI board containing an XC6216, an XC4013E to implement the PCI protocols and some other functions. It defines the hardware/software interface.

With the XC6000DS software provided, Reads and Writes of up to 32 bits can be made to any 32 registers in a column of the XC6216 (this could be more using wildcarding). The control registers on the XC6216 may also be written and read with this software.

The value on any gate, multiplexer or register output may be read from the circuit, which to the host processor looks like an SRAM. The hardware can be clocked by the software.

Model Technology V-System Simulator

The VHDL (and Verilog) simulator from Model Technology (ModelTech) has the unique feature of allowing access to the simulation data models, via specific access routines. These functions and procedures form the so-called VHDL foreign language interface (FLI). By calling these functions from within the C/C++ code written as part of a co-design, it is possible to probe and to drive signals within the hardware part of the design. For example, the function *mti_GetSignalValue()* returns the value of a hardware signal from a simulation.

It is also possible to schedule drivers on signals at some point in the future and in fact to schedule some special function calls.

Although originally intended to speed up simulation by providing pre-compiled simulation models for parts of a design, it has been found possible to use the same functionality to implement part of the communications protocol which is required between the application code and the simulator. Code has been written to do just this.

Implementing the interface

There are two separate blocks of code required to implement the interface between some application software and the ModelTech simulator. One handles calls from the application code to read from, write to or to clock some signal in the hardware. This will be referred to as the ***mti6200*** code. The other piece of code handles requests generated by the mti6200 code and calls the ModelTech FLI routines to implement the desired behavior. Data may also be passed back to the application code. This latter code will be referred to as the ***mti_rxtx*** code.

MTI6200

The following routines are available to the designer:

1) mti_initialize
2) mti_read
3) mti_write
4) mti_clock
5) mti_end

The function *mti_initialize* should be called somewhere at the start of the application code, with the name of the hardware design as an argument. the will start up the simulator and set up the required data structures and interrupt handlers.

The procedure *mti_read* may be called with a signal or port name and a slice range as arguments. This will return the value of that signal from the simulator. For a single bit signal the slice ranges have no meaning.

The function *mti_write* has several possible arguments. The signal name must be given, as must the value to drive onto it. Optional arguments include a slice range, a flag to signify that this write should happen in parallel with the next write, and a delay time. If the delay time is non-zero then this will be the time in nanoseconds (ns) which the simulator allows signals to settle before allowing any more

reads/writes to the circuit. Otherwise a default delay of 1000 ns is used.

As expected, the function *mti_clock* will result in a signal in the design being clocked. The arguments to this routine are the name of the signal, the number of clocks to generate, and the period of those clock pulses. A clock pulse begins with a low value on the signal for half the period, followed by a high value for the remaining half. Finally the signal is set low again.
To complete the mti6200 code, the function *mti_end* may be called to halt the interaction between the simulator and the application code.

Names of signals and ports in the design should have the following format: "/lev1/lev2/.../signame", where lev1, lev2, etc. are the names of levels of hierarchy in the design. The top-level design is "/".

The mti6200 code uses a class called ***Pci6200***. It includes the routines described above, as well as dummy routines for all the functions available within the ***pci6200*** code. This code is available for use in an implementation based around the XC6000DS development system and uses a class of the same name - ***Pci6200***. This allows the designer to simply substitute the *#include "Pci6200.h"* for *#include "Mti6200.h"* in the application code. Any calls to the pci6200 routines will effectively be ignored if the latter *#include* is used.

MTI_RXTX

This piece of code must be precompiled as a shared object file and may then be referred to in the VHDL hardware description as a foreign architecture. This may then be instanced in the top-level design.

The ModelTech manuals specify exactly how to do this, but the following description should suffice:

1) Add the following lines to the VHDL description of the circuit:

```
entity cosim is
end;
architecture sw of cosim is
   attribute foreign : string;
   attribute foreign of sw : architecture is
"mti_rx_tx_init mti_rx_tx.so";
begin
end;
```

2) In the top-level design's architecture header add:

```
component cosim
end component;
```

3) In the top-level architecture body add:

```
software : cosim;
```

This results in the shared object code, *mti_rx_tx.so*, begin "instanced" and the routine *mti_rx_tx_init* being called during the initialization phase at the beginning of simulation.
The mti_rxtx code has the following functions defined:

1) mti_rx_tx_init
2) mti_rx
3) mti_tx

The function, *mti_rx_tx_init* is an initialization function which sets up the required data elements for the co-simulation to work. In fact this is only ever called once, so the Restore and Restart parts of this code, as described in the ModelTech FLI manual, are not actually required.

The handles to the other two functions are created here, and the function *mti_rx* is scheduled for 1 tick in the future.

The *mti_rx* function extracts the details of the requested behavior from a file written by the mti6200 code, and places this in an internal data model, before scheduling the function *mti_tx* to begin in 1 tick.

The *mti_tx* function then implements that behavior, for instance reading the value of some signal from the hardware under simulation and placing the result in a file which will be read from by the mti6200 code. This function also schedules *mti_rx* to run again after some amount of time, which may be longer than a single tick if data has been driven into the circuit, to allow the circuit to settle.

Sychronization

This co-simulation method depends on the fact that the application code and the simulation are run "in parallel" as two separate (forked) processes under UNIX. It is not desirable that one piece of code gets "ahead" of the other. For instance, if the application requests a write to some signal in the hardware, it is necessary to wait for the simulator to "catch up", propagating this change through the circuit, before

handling any more requests from the application code.

To this end the UNIX interrupt signals SIGUSR1 and SIGUSR2 are used by both pieces of code to synchronize their actions. Before performing any request, the application code must wait until the mti_rxtx code is ready to receive the request. In turn, before the request may be handled by the mti_rxtx code, the application must signal that the details of that request is ready and waiting in a specific file.

All these details are fortunately hidden from the user.

Method

Co-simulation consists of the following steps:

1) Write VHDL describing the hardware part of co-design. This will be altered slightly from simulation, as opposed to synthesis, by adding the VHDL code described in a previous section.

2) Write application code. This may be in C or C++, and may include all the pci6200 function calls normally used in a co-design, for instance *initialize()*, *reset()* and *set_bus_width()*. These calls will effectively be ignored during simulation. Wherever the functions *set_column()* and *get_column()* are used, the additional mti6200 functions should be added, which will implement the same functionality as the former calls. The call to *mti_iniitialize()* should appear near the start of the application code, at least before any calls to *mti_read()*, *mti_write()* or *mti_clock()*.

A call to *mti_end()* should appear once all processing is finished, which will cause the communications between the application code and the simulator to terminate.

3) Compile the application code, using, for instance:

```
g++ Mti6200.cpp appcode.cpp -o appname
```

4) Compile the mti_rxtx code, using, for instance:

```
gcc -c -I /tools/mtech/modeltech4.6c
mti_rx_tx.c
ld -o mti_rx_tx.so mti_rx_tx.o
```

5) Compile the design, using:

```
vcom design.vhd
```

6) Run the application code. When this code reaches the call to *mti_initialize()* the simulator will start up, loading the design named in that call. It is then possible to select signals for tracing during simulation.

7) Run the simulation. A time-to-simulate should be given at this point, which should be long enough for the simulation to be completed. Alternatively, shorter runs may be done, one after the other. The “Break” and “Continue” commands may be given at any time during simulation.

By running the simulation, the application code proceeds, synchronizing with the simulation, until the call to *mti_end()* is encountered.

8) The application code should then end, returning control to the user. The simulator remains open, however, to allow the signal traces to be inspected, before an explicit “Quit” command is performed by the user.

9) If the user wishes to stop the co-simulation half-way through, then the “Break” command should be used on the simulator, followed by “Quit”. The application code may then be safely stopped with a “^C”.

Example

The following, simple example shows the use of this technique to test a circuit which comprises two multipliers and an adder, implementing the algorithm, `c = ax + by`, where `a`, `x`, `b` and `y` are 11-bit values.
On the hardware side, the signals we are interested in are *m1ina, m1inb, m2ina, m2inb* and *addout* and *cout*. In the final hardware implementation, the former signals are produced from columns of RPFD flip-flop parts, fed via the PC interface. The latter are passed to a column of buffers, which the host PC will read from.

So, any writes to the multiplier inputs will eventually be coded as calls to *set_column()*. Since there is not yet any layout for the circuit these have no meaning, and are augmented with calls to *mti_write()* for simulation.

Correspondingly, the calls to *get_column()* are augmented with calls to *mti_read()*.

The application code follows, with the additions for simulation highlighted.

All the multiplier input registers are aligned in a single column in the planned layout - MULTIN, with one multiplier's inputs on the lower half of the array, one at Y=0 and one at Y=16. the other multiplier's inputs are planned to be on the top half of the array, one at Y=32 and one at Y=48.

We will therefore be able to write both of a multiplier's inputs with a single, 32-bit write to the XC6000DS board.

```
#include <stdio.h>
// For use after place, route and makebits:
// #include "Pci6200.h"
// For use with Mdoeltech co-simulation:
#include "Mti6200.h"

#define MULTIN 0
#define ADDOUT 63
#define all32Bits 0x00000000;
#define noBits 0xffffffff;

main()
{
    long res;
    Pci6200* board = new Pci6200();
    board->initialize();
    board->clock_on();
    board->reset();
    board->load_cal_file("multadd.cal");

    board->mti_initialize("multadd");

    board->set_map(all32Bits, noBits);
    board->set_column(MULTIN, 3+(4 << 16));
    board->set_map(noBits, all32Bits);
    board->set_column(MULTIN, 6+(8 << 16));

    // These two will be written concurently
    board->mti_write("m1ina", 3, 0, 10, 1);
    board->mti_write("m1inb", 4, 0, 10, 0);

    // As will these two
    board->mti_write("m2ina", 6, 0, 10, 1);
    board->mti_write("m2inb", 8, 0, 10, 0);

    // Now retrieve the result
    board>set_map(all32Bits, noBits);
    res = board->get_column(ADDOUT);

    res = board->mti_read("addout", 0, 21);
    res += board->mti_read("cout",-1,0)<<22;

    printf("3*4 + 6*8 = %d\n", res);

    // Repeat with different data...

    board->set_map(all32Bits, noBits);
    board->set_column(MULTIN, 9+(2 << 16));
    board->set_map(noBits, all32Bits);
    board->set_column(MULTIN, 5+(3 << 16));

    board->mti_write("m1ina", 9, 0, 10, 1);
    board->mti_write("m1inb", 2, 0, 10, 0);
    board->mti_write("m2ina", 5, 0, 10, 1);
    board->mti_write("m2inb", 3, 0, 10, 0);

    board>set_map(all32Bits, noBits);
    res = board->get_column(ADDOUT);

    res = board->mti_read("addout", 0, 21);
    res += board->mti_read("cout",-1,0)<<22;

    printf("9*2 + 5*3 = %d\n", res);

    board->mti_end();

}
```

Pros and Cons

The main reason for implementing this co-simulation methodology is to reduce the number of place and route re-design stages between initial hardware definition and having a working circuit. The alternative is to simply place and route the design, producing a CAL file, and to run the application around the XC6000DS. The circuit may be debugged at this stage using software breakpoints and the PCITest software. However, if a hardware bug is discovered, this can entail a possibly lengthy re-place and re-route stage after re-design.

The circuit can be proven ***before*** placement and routing using the method described here. This will run the application code MUCH slower than is possible with the XC6000DS, but does not require any placement or routing to have occurred beforehand. Where the design will be processing a lot of information, for instance in some video applications, a suitably small amount of information should be used as stimuli during co-simulation to avoid the simulation time becoming prohibitive.

Reprise

This applications note has described a method to implement co-simulation of application code and a hardware design, before the design has been placed and routed. An attempt has been made to minimize the number of alterations that need to be made to both software and hardware, between co-simulation and actual implementation on the XC6000DS.

Only the Model Technology V-System simulator allows the necessary links between software and hardware to be made.

Limitations And Restrictions

Warning: THIS IS AN UNTESTED DESIGN.

Xilinx, Inc. does not make any representation or warranty regarding this design or any item based on this design. Xilinx disclaims all express and implied warranties, including but not limited to the implied fitness of this design for a particular purpose and freedom from infringement. Without limiting the generality of the foregoing, Xilinx does not make any warranty of any kind that any item developed based on this design, or any portion of it, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country. It is the responsibility of the user to seek licenses for such intellectual property right where applicable. Xilinx shall not be liable for any damages arising out of or in connection with the use of the design including liability for lost profit, business interruption, or any other damages whatsoever.