

# Interfacing HDLs with conventional programming languages

FRANCISCO MORA AND ANDRES TORRUBIA, POLYTECHNIC UNIVERSITY OF VALENCIA

One of the typical problems of developing computer-system hardware is that you cannot program or debug its associated software unless a physical implementation of the hardware exists. Furthermore, if the device you've designed is complex or the specifications are not completely clear, hardware engi-

neers may make some arbitrary decisions about these specifications with little or no input from their software counterparts.

With the arrival of HDLs, hardware engineers can design devices using structured languages, such as VHDL or Verilog, targeted for both simulation and synthesis. These HDLs let you program applications that interface with the hardware's HDL model rather than with the hardware itself.

Figure 1 depicts a simple interaction between a software application and some hardware attached to a real system. You can reasonably assume that the software limits and specifies the memory resources that the hardware uses. This assumption is true in many modern multitasking environments in which the operating system manages all memory and hardware to avoid conflicts between applications.

For now, consider that the application communicates only with the hardware by means of a set of input and output functions and through the system memory. How would the hardware-to-software interface work under this scheme? First, the application, written in some computer programming language, such as C, C++, or Pascal, would call a memory-management function to retrieve memory for further operations. The hardware may at some time access this memory. If the system had logical or virtual-memory capabilities, the application would call a function to find out the physical address of the previously allocated memory. The application would then initialize the hardware to a known state by means of a set of input and output functions. The application could then start issuing commands to the hardware. Some of these commands could tell the hardware where to look in physical memory for the next operations it needs to perform.

Assume that the hardware can later gain access to physical memory. The operating system usually provides supplied functions, the input and output and memory-management functions that the application calls. The calling order is not fixed but illustrates the interface in a real environment.

For designing a hardware-simulation environment, you

Programming languages that interface with HDL models facilitate hardware/software codesign. This interface technique—along with a practical example using a VHDL behavioral model of a memory-copier device in a C-based application—helps you with this codesign.

should be able to move both HDL code and application code between the simulation and real environments with few modifications. Furthermore, the application should see no difference in HDL code between both environments. Along with these features, a hardware-simulation environment

must achieve transparent emulation of the input and output functions the application uses as well as memory sharing between the application and the HDL hardware model. The most difficult task is memory sharing between the application code and the HDL model, because you are using a standard HDL simulator without extended memory-sharing features.

Figure 2 illustrates the interface between the parts of the proposed hardware-simulation environment. The blocks in Figure 2 are defined as follows:

- APP (application) is an application written in a conventional programming language. It should work in the simulation environment as well as in the real environment ideally without modifications.
- SF (supplied function) is generally I/O and memory-management. The SF and APP are probably written in the same language, and the simulation environment rather than the operating system can provide them.
- HDL-C (HDL code) is a hardware model, written in an HDL,

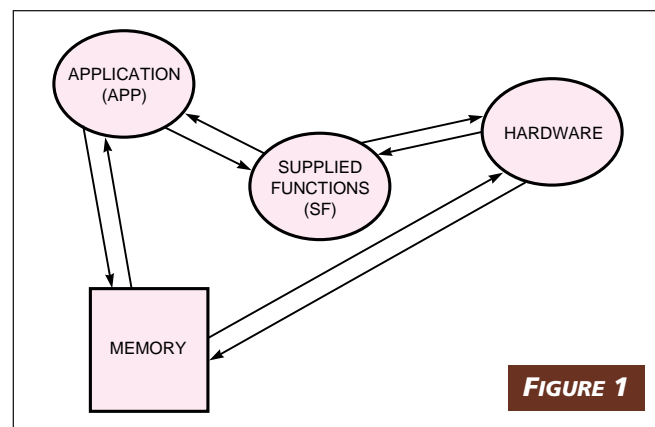


FIGURE 1

A simple interaction occurs between a software application (APP) and some hardware attached to a real system.

## INTERFACING HDLs

of the hardware being simulated. The model should work in the simulation environment as well as in the real environment ideally without modifications.

- HDL-SIM (HDL simulator) is an executable program that interprets HDL code so that you can see how it behaves under certain stimuli. Assume that HDL-SIM is unmodifiable.
- AH-GATEWAY (application HDL-C gateway) is a gateway that provides basic communication between the APP and the HDL-SIM. AH-GATEWAY allows the APP to communicate with the HDL-SIM on an I/O basis.
- A-MEM-GATEWAY (application memory gateway) is a gateway that exports shared memory previously allocated with a memory-management SF so that HDL-C can later use it. A-MEM-GATEWAY also imports memory from the HDL-C so that shared memory is coherent between the APP and the HDL-C. A-MEM-GATEWAY and H-MEM-EMU are tightly coupled.
- H-MEM-EMU (HDL-C memory emulator) is an emulator that provides the HDL-C access to memory exported by A-MEM-GATEWAY. It also exports this memory when the HDL-C has modified it so that the memory is coherent between the APP and the HDL-C.

You must make minor modifications to make the APP work in a hardware-simulation environment. The APP has to perform several initializations on start-up and must release resources upon quitting, using AH-GATEWAY and A-MEM-GATEWAY. SF differs from real-environment functions. Typically, SF in a simulation environment calls functions supplied by A-MEM-GATEWAY and AH-GATEWAY. Conditional compilation of the APP can easily accomplish these calls.

For the HDL-C to work in a hardware-simulation environment, the environment should provide only special shared-memory functions. You need shared-memory functions because HDL-SIM executes HDL-C, and APP controls HDL-SIM by means of AH-GATEWAY. HDL-C basically needs peek and poke functions, as well as some procedure to refresh shared-memory contents upon input to and output from HDL-C. Assume that the only way to communicate large amounts of external data to and from the HDL-C is through files, so H-MEM-EMU is heavily based on these files.

The actual hardware/software interface scheme in a simulation environment works as follows:

- The APP performs all necessary initializations on AH-GATEWAY and A-MEM-GATEWAY.
- The APP retrieves shared memory by calling an SF from A-MEM-GATEWAY. A-MEM-GATEWAY keeps track of this memory for later use.
- The APP performs I/O commands from and to the HDL-C using a set of functions that AH-GATEWAY supplies. AH-GATEWAY then informs A-MEM-EMU that an I/O operation

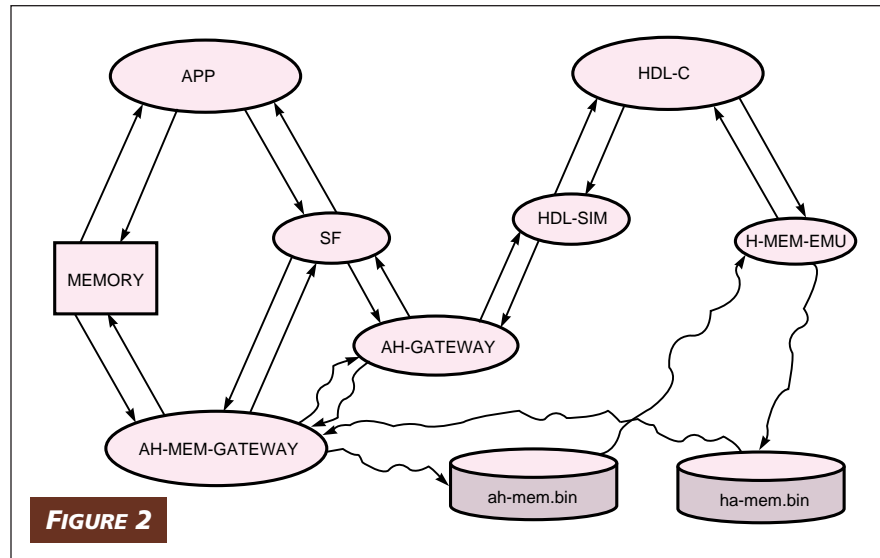


FIGURE 2

Interaction occurs between the parts of the proposed hardware-simulation environment.

is about to be executed, so A-MEM-EMU flushes all shared memory to file ah-mem.bin. AH-GATEWAY instructs HDL-SIM to perform all necessary input and output cycles upon request. Users can customize these cycles. The simulation environment then gives control to HDL-SIM.

- HDL-SIM applies stimuli to HDL-C so it executes the desired input or output cycle. In turn, HDL-C calls H-MEM-EMU so HDL-C gets the shared memory from file ah-mem.bin. HDL-C may call a set of functions provided by H-MEM-EMU to read or write this memory. When HDL-C finishes the specified cycle, H-MEM-EMU flushes the shared memory to file ha-mem.bin.
- AH-GATEWAY gains control and instructs A-MEM-GATEWAY to synchronize shared-memory contents with those of the HDL-C. A-MEM-GATEWAY reads the file ha-mem.bin and updates shared memory from the APP's perspective. If the simulation does an input operation, AH-GATEWAY reads the resulting response from HDL-SIM.

Under this asynchronous environment, you should start all commands with an I/O cycle.

The previously described interface scheme has been successfully implemented in a Microsoft Win 32 environment using VHDL as the HDL and C as the programming language. The implementation uses Microsoft (Redmond, WA) Visual C++ Version 5.0 for C programming and Model Technology's (Beaverton, OR) VSystem for VHDL compilation and simulation. The simulated sample hardware is a blitter, or memory-copier device. The hardware model is a behavioral description of the blitter.

The VHDL blitter model (HDL-C) comprises an 8-bit input-address of fset port to the blitter's 32-bit registers, a 32-bit I/O data port for reading and writing to and from the blitter's registers, a 1-bit input line decoded-read (dread) port, and a 1-bit input line decoded-write (dwrite) port. To read from a blitter register, the simulation environment must apply a rising edge to the dread line. Once address lines are stable, the environment out-

## INTERFACING HDLS

puts the contents of the address register to data. Writing to a blitter register is accomplished by applying a rising edge to `dwrite`, once address and data lines are stable, with the address and the contents of the register to be written.

The blitter has several 32-bit I/O-mapped registers (Table 1). The operating procedure is simple. To copy `REG_BYTE_CNT` bytes from address

`REG_SRC_ADDR` to the address specified by `REG_DEST_ADDR`, the system has to write value `COMMAND_COPY_MEM` (0x00000001) to register `REG_COMMAND`.

By polling `REG_STATUS` until bit `STAT_BLITTER_BUSY` (0x00000001) is 0, you can determine whether the blitter has finished the previous command. During simulation, bit `STAT_BLITTER_BUSY` is always 0 because HDL-C and APP executions do not overlap. Also note that both `REG_DEST_ADDR` and `REG_SRC_ADDR` point to addresses in memory space and not to registers in I/O (port) space.

The sample APP, using HDL-C, is a simple program that performs several tests of the blitter: initializing, reading from registers, writing to registers, and copying memory by using the `COMMAND_COPY_MEM` blitter command. Listing 1 is a section of the application source code.

`AH-GATEWAY` and `A-MEM-GATEWAY` are both implemented in C and linked with the sample application `BLITTEST.EXE`. `AH-GATEWAY` starts `HDL-SIM` (`VSIM.EXE`) and sends it keystrokes for communication. For example, when `AH-GATEWAY` wants to retrieve some data from `HDL-SIM` for an input command, it forces `HDL-SIM` to write a list file containing current-signal status. `AH-GATEWAY` parses this file and fetches the requested value, returning it to APP.

`A-MEM-GATEWAY` implementation is somewhat more complicated. `A-MEM-GATEWAY` maintains a list of shared-memory items allocated by the simulation system. When `AH-GATEWAY` performs an input or output operation, `A-MEM-GATEWAY` saves this list to file `ah-mem.bin`. Upon the list's return, `A-MEM-GATEWAY` loads the file `ha-mem.bin`, which `H-MEM-EMU` generates, and updates memory contents with those of the file. The format of both files `ah-mem.bin` and `ha-mem.bin` is such that they contain a list of memory elements allocated with `A-MEM-GATEWAY` and its associated contents.

The behavioral description of the blitter is in VHDL with one main process sensitive to `dread` and `dwrite` changes. When the simulation sees a rising edge on `dread`, the system checks the address for readable registers (currently, only `REG_STATUS`), and if a match exists, the system outputs the contents of the

**TABLE 1—BLITTER WITH SEVERAL 32-BIT I/O-MAPPED REGISTERS**

Address	Name	Access
0x00	REG_COMMAND	Write
0x01	REG_STATUS	Read/write
0x02	REG_SRC_ADDR	Write
0x03	REG_DEST_ADDR	Write
0x04	REG_BYTE_CNT	Write

specified register to data. If a match does not exist, the system updates status register `REG_STATUS` to reflect that status in `STATUS_INVALID_REGISTER`.

If the system detects a rising edge on `dwrite`, it checks the address for writable registers. If a match exists, the next operation depends on the type of register being written. If the register is of

type `REG_SRC_ADDR`, `REG_DEST_ADDR`, or `REG_BYTE_CNT`, the system writes the value data to the selected register. If it is type `REG_COMMAND`, the system compares value data with allowed commands (currently, `COMMAND_RESET` and `COMMAND_COPY_MEM`) and executes the selected command.

`COMMAND_RESET` just resets all internal registers, including status. `COMMAND_COPY_MEM` copies `REG_BYTE_CNT` bytes from address `REG_SRC_ADDR` to address `REG_DEST_ADDR`. `COMMAND_COPY_MEM` makes heavy use of `H-MEM-EMU`. `H-MEM-EMU` is written in VHDL in the form of a VHDL package that is instantiated later in HDL-C. `H-MEM-EMU` provides functions to load and save shared memory from files and to access that memory on a per-byte basis. Because effective memory modification within the HDL-C occurs only during command writes (specifically, on `COMMAND_COPY_MEM`), the system loads and saves shared memory only during output cycles.

You can easily port this simulation environment to other operating systems and HDL simulators. (All source code is available electronically from [www.upv.es/die/espanyol/grp\\_micr.htm](http://www.upv.es/die/espanyol/grp_micr.htm).) Changing the operating system on which the environment runs requires modifying several internal `AH-GATEWAY` functions. Using another `HDL-SIM` implies changes in `AH-GATEWAY`. Moving to different programming and HDL

```
printf(" * About to test blitter copy (COMMAND_COPY_MEM): \n");
printf(" Memory contents follow:\n Source: %s\n Destination: %s\n", pSource, pDest);

// Fill src, dest & byte count register with associated data
WritePortDword( REG_SRC_ADDR, pSource);
WritePortDword( REG_DEST_ADDR, pDest);
WritePortDword( REG_BYTE_CNT, sizeof(testString));

printf(" * Performing blitter copy (COMMAND_COPY_MEM)\n");
printf(" Source (REG_SRC_ADDR): 0x%08lx\n", pSource);
printf(" Destination (REG_DEST_ADDR): 0x%08lx\n", pDest);
printf(" Bytes (REG_BYTE_CNT): 0x%08lx\n", sizeof(testString));

// Perform the copy
WritePortDword( REG_COMMAND, COMMAND_COPY_MEM);

// At this point pDest should contain the same data as pSource
printf(" * Polling completed\n");
printf(" Memory contents follow:\n Source: %s\n Destination: %s\n\n", pSource, pDest);
```

## INTERFACING HDLS

languages means new implementations of AH-GATEWAY, A-MEM-GATEWAY, and H-MEM-EMU.

This interface provides a number of benefits. You can use the same application to test hardware during early system development and then later with real prototypes. You can also write applications with only a behavioral model of their associated hardware. You can also program and debug device drivers by combining this approach with the technique described in Reference 1.

This simulation environment also has some shortcomings. Because of the overhead produced by the abusive file activity involved in each transaction, a performance penalty can take place if several transactions occur. You can minimize this problem by using a small RAM disk for these files or by using some faster means of sharing memory with the HDL-SIM. Another problem is that you can start commands only with I/O-mapped registers. This restriction results from the fact that shared-memory modifications are synchronized with HDL-C upon I/O cycles, because APP and HDL-SIM run asynchronously.

### References

1. Quicksall, E, and K Gibson, Simulation and device driver development, Dr. Dobbs Journal, January 1997, pg 52.
2. Goering, Richard, VHDL shared variables create dissent spec divides EDA, Electronic Engineering Times, Jan 20, 1997.
3. Ashenden, PJ, The Designer's Guide to VHDL, Morgan

Kaufman, San Mateo, CA, 1996.

4. Gupta, RK, and SY Liao, Using a programming language for digital system design, IEEE Design and Test of Computers, April to June, 1997, pg 72.

### Authors biographies

Francisco Mora is an assistant professor in the Electronic Engineering Department at the Polytechnic Institute of Valencia (Spain), where he received his PhD and where he has worked for six years. In his spare time, Mora enjoys traveling, skiing, and jogging.

Andres Torrubia is an assistant professor in the Electronic Engineering Department at the Polytechnic Institute of Valencia (Spain).

### VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

<b>High Interest</b>	<b>Medium Interest</b>	<b>Low Interest</b>
<b>590</b>	<b>591</b>	<b>592</b>