

# **System-On-Chip Design with the Leon CPU**

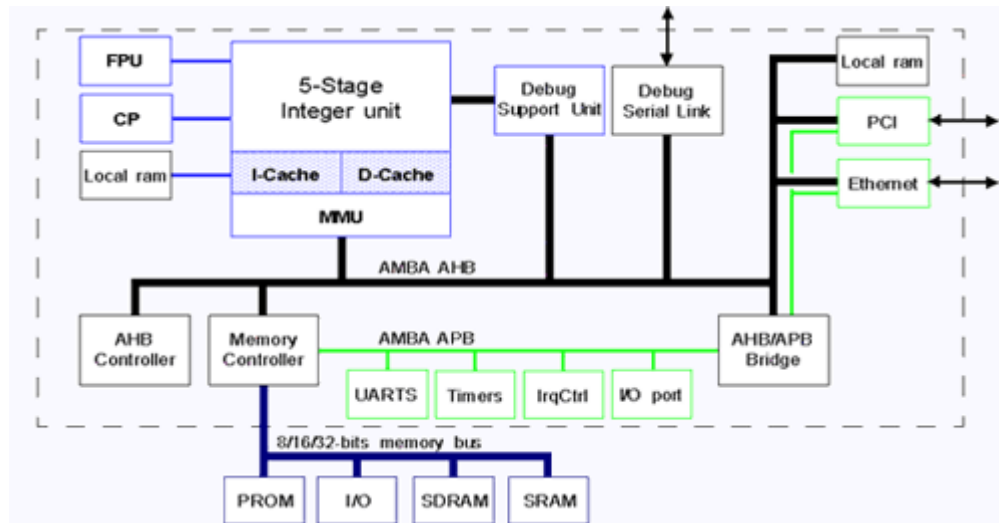
The SOCKS Hardware/Software Environment

## **Introduction**

Digital systems typically contain both, software programmable components, as well as application specific logic. In the past, a general purpose microprocessor and an FPGA would be assembled on a printed circuit board. With the tremendous progress in silicon integration, such a digital system can now be integrated onto a single silicon die. Such a system-on-chip (SOC) approach brings new challenges but also higher flexibility and performance to the field of Hardware/Software CoDesign. The SOCKs environment brings together a high performance CPU, a flexible SOC bus structure, custom HDL logic and a C/C++ compiler.

## The Leon CPU

Leon is a Sparc compliant 32-bit CPU core that is available to the public domain as a synthesizable HDL model. It was developed by Jiri Gaisler of the European Space Agency (ESA). The Leon VHDL code has been made public at [www.gaisler.com](http://www.gaisler.com). It combines a central CPU with a host of peripherals and memory as well as Amba controllers. The general structure is shown below:

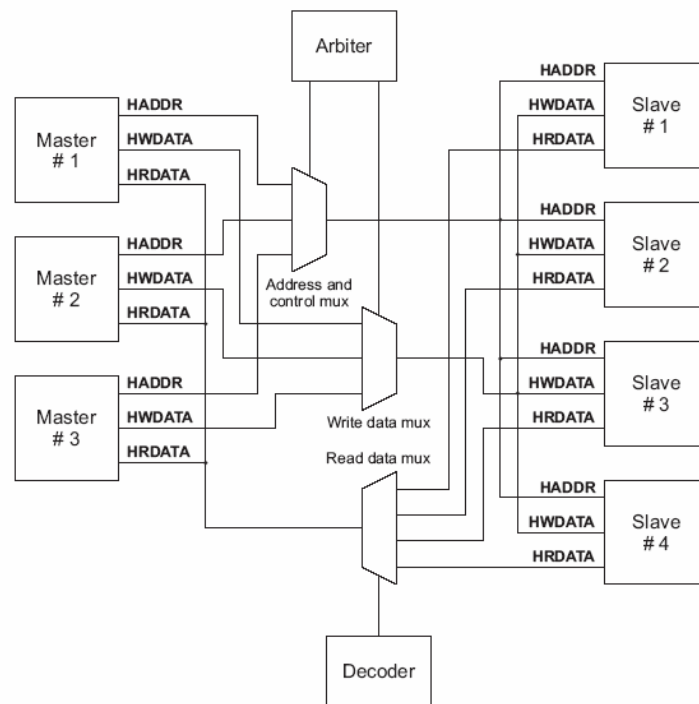


Basic Leon Diagram

The VHDL model is highly flexible and can be configured to meet the application. Among the optional components are: floating-point units, PCI controller, Ethernet controller, variable cache sizes, embedded boot ROMs, multipliers and dividers.

## The AMBA Bus

The Amba bus was developed by ARM to help interconnect the major blocks in SOCs. The biggest difference of this bus to a conventional external microprocessor bus is that it is organized in a star topology, as opposed to a bus topology. (Note that in the diagram above, a bus is shown for simplicity, even though a star topology is used). The problem with a bus is the high load of an interconnect spanning an entire chip and connecting to many bus drivers inside the blocks. In addition, each bus driver has to be able to disconnect from the bus with a tri-state mode. In a star topology, all connections are point-to-point based. This means there is no need for tri-state drivers as each wire has exactly one driver and one receiver. The interconnect load is much lower because each block has to drive only one receiver and a wire that connects to only one block, as opposed to many blocks all over the chip. The general structure of this bus is shown below:



**Basic Amba Bus Diagram**

Simple muxes are used to switch connections between the Masters and the Slaves. A bus Arbiter grants the bus to a particular Master. The address on the address bus is decoded into select signals and connected to the Slaves.

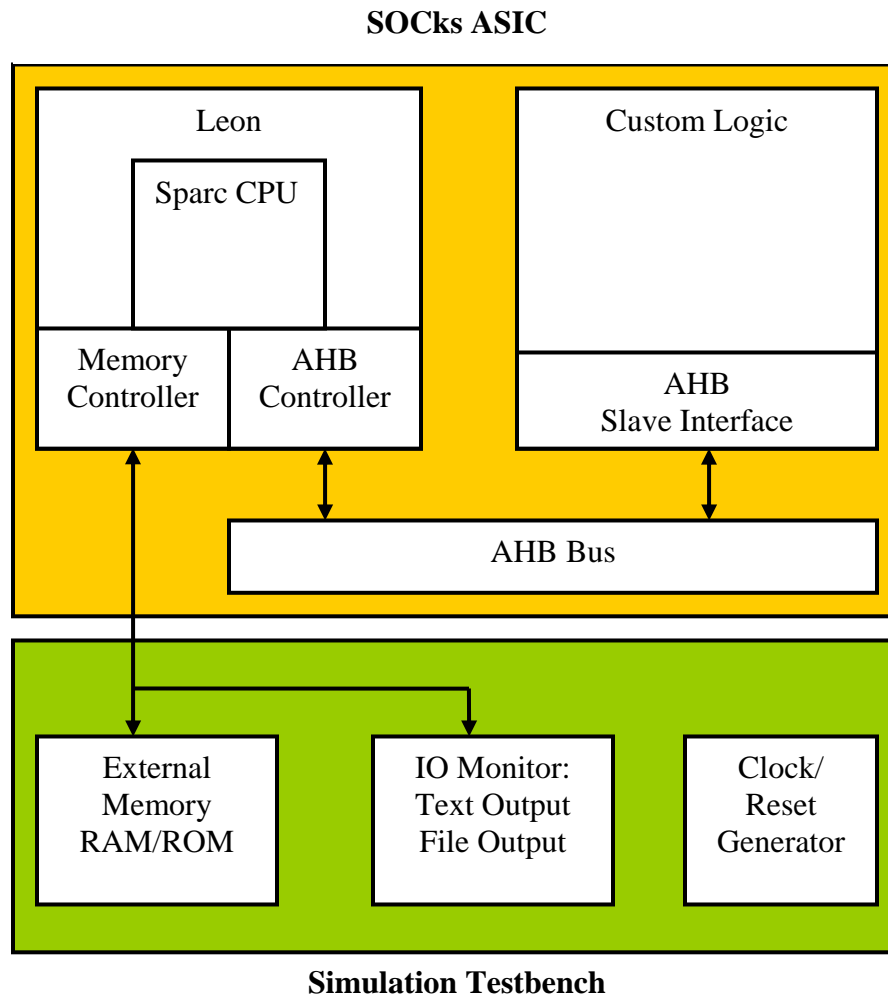
A bus transaction has 2 phases: The Address phase and the Data phase. During the Address phase the Master places the target address on the bus and the addressed Slave has its select signal asserted. At the rising edge, the slave latches the address and either fetches data (read access) or prepares to receive data (write data). At the next rising edge, the data is valid and the transaction is finished. If the slave needs more time to complete the transaction, it inserts wait-states by keeping the HREADY line low.

In addition to this single transfer, the Amba bus also supports burst and split transactions.

A digital system typically combines high bandwidth components (e.g. CPU, RAM, PCI) and low bandwidth components (e.g. UART, Keyboard, Timer). The Amba bus separates both groups onto two busses, in order to prevent excessive wait-states from slow components. The AHB connects high-speed devices and the APB connects slower peripheral devices. An AHB/APB bridge connects the two into a unified address space.

## The SOCKS system

Below is a block diagram of the system environment:



The actual chip contains two major blocks, the Leon and the custom logic block. The AMBA bus controller and arbiter as part of the Leon block. The address range of the custom block is specified in the Leon configuration. The custom logic block can accommodate any digital logic. The communication with the main CPU is entirely through the AMBA bus. Other blocks can be added as well, simply by modifying the Leon Amba bus configuration.

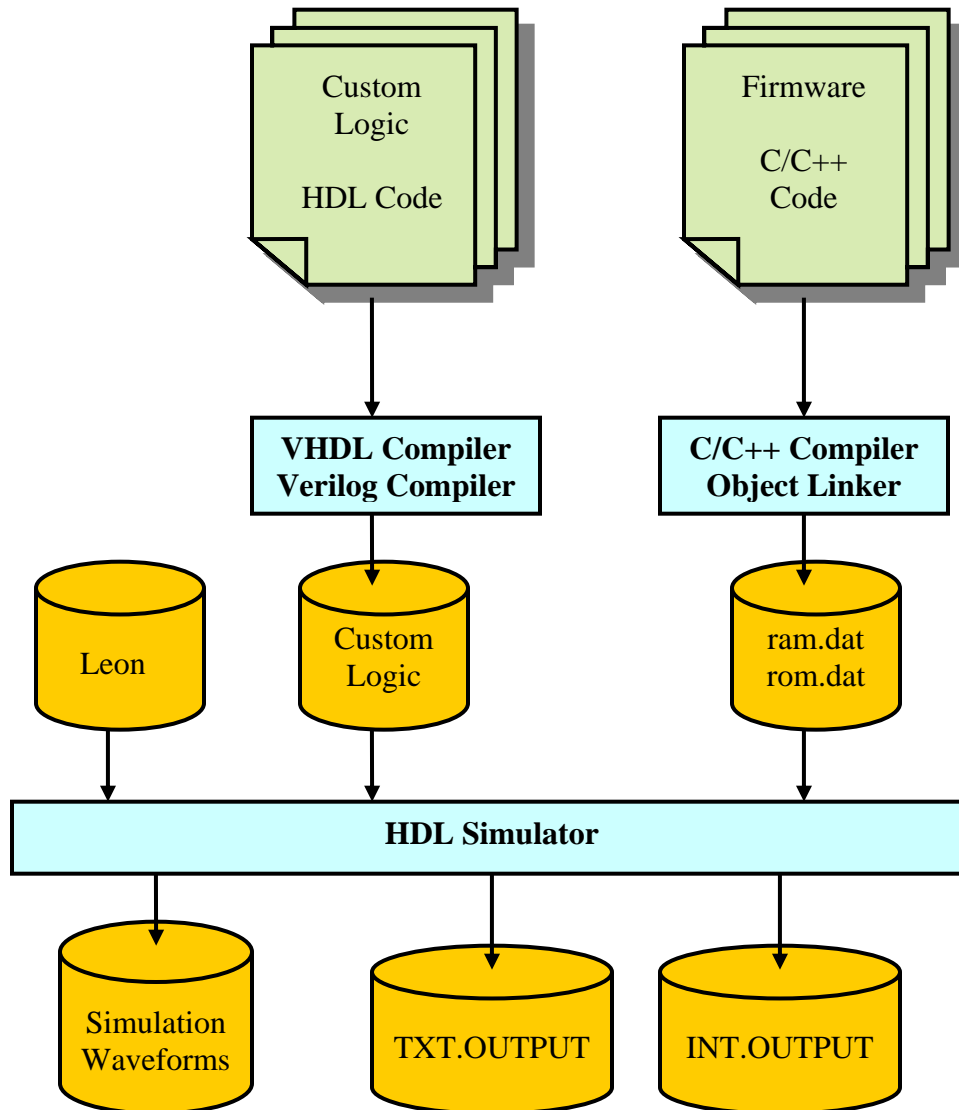
The output from the C/C++ compiler is written into two files: ram.dat and rom.dat. The ROM code is constant for all programs. Both files are read by the VHDL simulator prior to simulation start and loaded into the external memory in the testbench. The IO monitor responds to IO access to certain addresses. The #IOS line indicates if the current access is a memory or IO access. Depending on the IO address, the following happens:

- IOBASE + 0: The LSB is output on the terminal
- IOBASE + 4: The LSB is written into TXT.OUTPUT

- IOBASE + 8: The 32-bit data word is written as an integer to INT.OUTPUT
- IOBASE + 12: The test-bench code ends the simulation

This allows for the programmer to write and compile C/C++ code and execute it through a VHDL simulation. The IO monitor can be used to output debug information and the program results. By allowing the programmer to end the simulation, both short and long programs can be easily accommodated.

## Hardware/Software CoDesign Flow



As can be seen, the mixed VHDL/Verilog Simulator forms the central point of the Hardware/Software CoDesign flow. It reads the compiled HDL code for the Leon and the custom logic block. It also reads in the compiled firmware source code and places it in the program memory. Simulation waveforms can be generated to debug the digital logic. Firmware output is written to the TXT.OUTPUT and INT.OUTPUT files, as well as the simulation terminal.

Each HDL block resides in its own database. The locations of the databases are specified in the "cds.lib" file. Each database corresponds to a directory in the file system. The Leon database is kept in a central location and does not have to be compiled again after changing the custom logic block. The simulation waveforms are written in "shm" format and can be displayed with "simvision". The other tools in the flow are the Cadence NC-Sim simulator, as well as the NC-VHDL and NC-Verilog compilers. The simulation

model is built by NC-ELAB by elaborating over all compiled HDL databases. NC-ELAB automatically handles both VHDL and Verilog code.



## Included Files

Following is the list of all the files included in the SOCKS distribution. Note that the Leon source code is not included, due to its size. Also, once compiled into its database, there is no need to compile it again, even after the custom logic changes. The Leon source code is located at /import/vlsi7/jgrad/socks/socks\_admin/hdl/leon. The file ./hdl/cds.lib is used to tell the NC-Sim tool where to locate the compiled Leon code.

Directory	Description
./exe	Unix scripts to compile and simulate the SOCKs ASIC
./doc	SOCKs Documentation
./firmware	C/C++ Source Code Folder
./firmware/cube	Demo program #1
./firmware/bubblesort	Demo program #2
./sim	HDL Simulation Folder
./testbench	VHDL test-bench code
./testbench/include	Test-bench include files
./testbench/Tcl	TLC Scripts for NC-Sim
./hdl	HDL Folder
./hdl/custom	HDL for the Custom Logic
./hdl/TOP	HDL for the ASIC toplevel
./hdl/rllib	Compiled HDL folder
./hdl/rllib/custom	Compiled Custom logic
./hdl/rllib/top	Compiled Top-level logic

## Included flow scripts

Three shell scripts are included to provide a quick simulation turn-around:

1. socks\_compile            Re-compiles the custom and TOP code
2. socks\_sim                Starts a new HDL simulation
3. socks\_clean              Removes all temporary data

There is no script to compile C/C++ code, since a make file is used for that purpose. To compile for example the C/C++ code for the “cube” demo, the sequence is as follows:

```
cd firmware/cube
make
```

The tool “make” will read the file “Makefile” and automatically re-compile and link all files that have been modified since the last compilation. To force make to recompile all files, do “make clean” and “make” to first delete all compiled data and to then re-compile the C/C++ code. The resulted binary code will be placed into ram.dat and rom.data. It will be read by the simulator from the source code folder.

Following is the sequence of commands to compile the HDL code:

```
cd hdl
../exe/socks_compile
```

The socks\_compile script must be invoked from the hdl folder. It compiles all files that end with “.v” with NC-Verilog into the custom database. It then compiles the toplevel. If the custom logic is implemented in VHDL, the socks\_compile script has to be modified to call NC-VHDL instead of NC-Verilog.

The sequence to simulate for example the “cube” demo is as follows:

```
cd sim
../exe/socks_sim cube
```

The socks\_sim script takes the name of the source-code folder as its first parameter in order to locate the ram.dat and rom.dat files. The general syntax of socks\_sim is:

```
socks_sim source_code_folder [ partial | full ]
```

The second parameter is optional and controls the generation of waveforms:

1. “full”                    Generate waveforms for the entire design (about 10MB)
2. “partial”                Generate waveforms only for the custom logic
3. blank                    Do not generate waveforms

The generation of waveforms slows down the simulation and can use lots of disk space. Typically no waveforms are needed, except to debug the custom logic, which can be done with partial waveforms. Full waveforms are needed to debug the Leon or the toplevel. To display the waveforms, start “simvision&” and open the “socks\_sim.shm” folder.

The socks\_clean script removes all temporary data. It does not delete any code from the firmware or hdl folder. It will delete the contents of the HDL databases in ./hdl/rtl.lib and the contents of the ./sim folder. A simulation typically requires 10MB, and another 15MB if full waveforms are generated. Partial waveforms are small and depend on the size of

the custom logic. To clean all temporary data, run the following command from the root folder:

```
exe/socks_clean
```

Before running any of the NC tools, they need to be added to the path. To do so run once in each new terminal:

```
source /import/cad1/scripts/cadence.ic.cshrc
```

To check that all tools are found use “which ncsim”.

It is not necessary to type the above command at every login. The command can be added to .cshrc, which is automatically processed at each login. To do so run exactly once the following command.

```
echo "source /import/cad1/scripts/cadence.ic.cshrc" >> ~/.cshrc
```

This will add the source command to .cshrc, as can be verified with “cat ~/.cshrc”.

## Developing C/C++ code

The compiler is from the “rtems” package released by ESA. It is based on the GNU gcc compiler. The tools must be installed under “/opt/rtems” and “/opt/rtems/bin” must be added to the path. Since /opt is local to each machine, rtems is currently only available on select machines, e.g. skew. To add rtems to the path, run once and only once the following command:

```
echo "set path = ($path /opt/rtems/bin)" >> ~/.cshrc
```

The file .cshrc is automatically read at each login. The command above added one line to this file, as can be seen by doing “cat ~/.cshrc”. Note that “~” is a shortcut for the user’s home directory. To verify that the compiler can be found use

```
which sparc-rtems-gcc
```

Each program has to be located in a folder under ./firmware. It is easiest to use one of the existing programs as a template:

```
cd ./firmware
cp -r bubblesort new_prog
```

This would create a new folder “new\_prog” as a copy of the “bubblesort” folder.

The central file of each program is “leon\_test.c”. It contains the function “leon\_test( )” which is the starting point of each program. On a conventional operating system it would be known as “main( )”. Another difference between conventional programs and firmware running on an embedded CPU is that a program typically runs in an endless loop. For testing purposes this is not practical. Instead, there is a way for the source code to end the simulation, which is done with the following code:

```
exit_sim();
while (1) { };
```

The first function writes a word to the designated address in the IO address space that causes the test-bench monitor to abort the simulation. The second command is an endless loop that is executed until the simulation ends.

Following is a list of all provided support functions:

- print\_scr(c)                Outputs the character c on the simulation screen
- print\_txt(c)                Writes the character c to TXT.OUTPUT
- print\_int(i)                Writes the integer i to INT.OUTPUT
- exit\_sim(c)                 Exits the simulation, c is ignored

These functions are declared in “misc.c” and are simple shortcuts for writing data to designated addresses in the Leon IO address space. The test-bench monitor is activated by the #IOS signal, which indicates an IO access, instead of a RAM/ROM access. Depending on the address, an action is performed.

This makes it possible to also run the source code on the host CPU instead of the Leon. This makes debugging much easier and the code runs faster. The only difference is that the four functions given above have to be re-defined in a suitable manner, for example with the “printf( )” command.

To access the custom logic memory area, a pointer “custom” is used. It is defined as follows:

```
int * custom = (int *) 0xA0160000
```

This makes it very easy to access registers inside the custom logic. The command

```
custom[i]=a;
```

would trigger a write transaction on the AHB bus and write the value of “a” to address 0xA0160000+4\*i. Since i points to a 32-bit integer it covers a range of 4 bytes. Similarly,

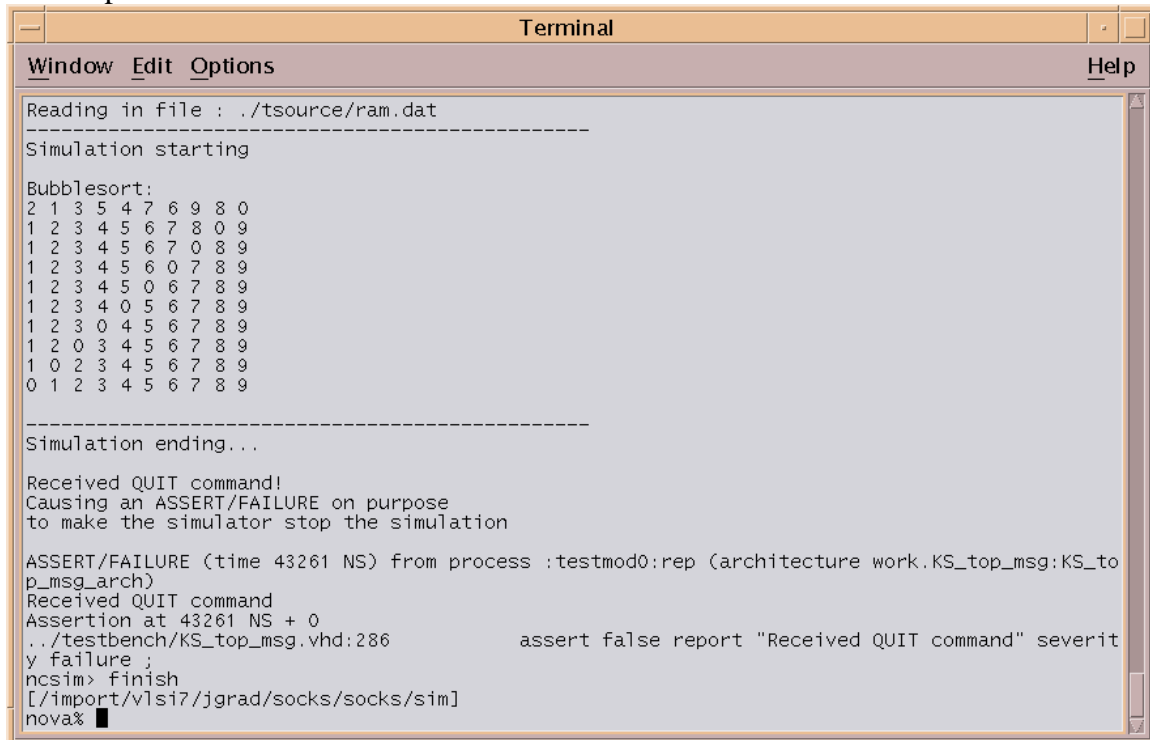
```
b=custom[j];
```

would cause a read transaction and read the word at address 0xA0160000+4\*j into the variable b.

Two demo programs are included: A bubble-sort program and a program to illustrate the cubing logic in the custom logic block. To run the bubble-sort program, do

```
cd ./sim
../exe/socks_sim bubblesort
```

The output will look like this:



```
Terminal
Window Edit Options Help
Reading in file : ../source/ram.dat
-----
Simulation starting
Bubblesort:
2 1 3 5 4 7 6 9 8 0
1 2 3 4 5 6 7 8 0 9
1 2 3 4 5 6 7 0 8 9
1 2 3 4 5 6 0 7 8 9
1 2 3 4 5 0 6 7 8 9
1 2 3 4 0 5 6 7 8 9
1 2 3 0 4 5 6 7 8 9
1 2 0 3 4 5 6 7 8 9
1 0 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
-----
Simulation ending...
Received QUIT command!
Causing an ASSERT/FAILURE on purpose
to make the simulator stop the simulation
ASSERT/FAILURE (time 43261 NS) from process :testmod0:rep (architecture work.KS_top_msg:KS_to
p_msg_arch)
Received QUIT command
Assertion at 43261 NS + 0
../testbench/KS_top_msg.vhd:286          assert false report "Received QUIT command" severit
y failure ;
ncsim> finish
[/import/vlsi7/jgrad/socks/socks/sim]
nova% █
```

The function “print\_scr( )” is used to print the progress on the screen. A set of 10 numbers is sorted by iteratively swapping neighboring numbers. The simulation is ended with “exit\_sim( )”. The way the test-bench ends the simulation is by causing a fake assertion error. Therefore, even though an error is reported, it is just a side effect of forcing the simulation to end. Note that this program only uses the Leon, and not the custom logic. The other program is called “cube” and utilizes the demo logic in the custom logic block. It is described in the next section.

## Developing custom logic

Currently, there are two files in the ./hdl/custom folder:

- `amba_iface.v` Contains the AMBA-bus interface
- `custom_top.v` Contains the “custom\_top” module definition

On the ASIC toplevel in ./hdl/TOP/trial\_Core.vhd the two main modules, “leon” and “custom\_top” are connected to the core module. The pads are defined in `trial_Padring.vhd`. Finally, “trial\_ASIC.vhd” connects the core and the padring to the final HDL toplevel. To avoid having to modify the core module, the top-level of the custom logic block should be called “custom\_top”.

In “custom\_top.v”, there are several signals available. They form the interface to the Amba slave interface in “amba\_iface.v”, where the actual AHB signals are generated.

Signal	Direction	Description
<code>rst_n</code>	Input	Active-low reset
<code>clk</code>	Input	Amba bus clock
<code>adr</code>	Input	Address on the Amba bus
<code>strobe</code>	Input	Indicates a valid address
<code>we</code>	Input	Write-Enable (high=write, low=read)
<code>ack</code>	Output	Indicate end of data phase
<code>data_in</code>	Input	Data from the CPU into the custom logic
<code>data_out</code>	Output	Data from the custom logic to the CPU

A typical transaction would be like this:

1. **strobe**=1 indicates a valid address in the custom-logic memory area
2. **we** indicates if the CPU is writing or requesting data
3. On the rising edge of the clock
  - If **we**=0, read and store the data from `data_in`
  - If **we**=1, put the requested data onto `data_out`

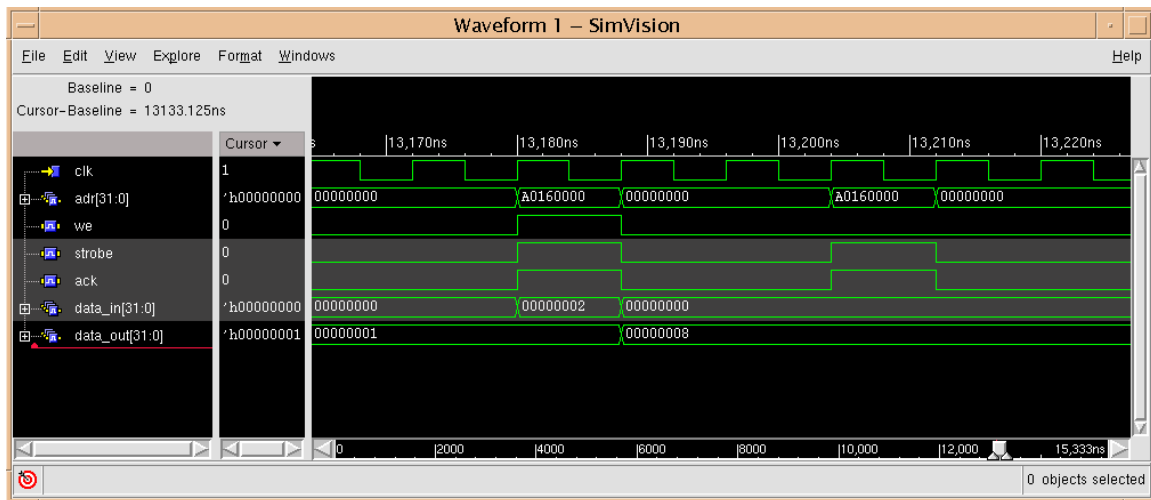
Currently, the custom logic block implements a cubing logic. That is, the output data is the input data to the third power. The address range for the custom logic block starts at `$A016_0000`. In the firmware, 10 numbers are sent to the custom logic block and the 10 resulting numbers are written into `INT.OUTPUT`. The relevant C source code is like this:

```
for(j=0;j<10;j++)
{
    custom[0]=nums[j];
    print_int(custom[0]);
}
```

The pointer `int *custom` has been set to address `$A016_0000`, therefore `custom[0]` addresses the 32-bit integer at `$A016_0000`. Consequently, `custom[1]` would address the 32-bit integer at `$A016_0004`. This is because `custom` is a pointer to an integer and an integer is 4 byte long. The function `print_int( )` writes an integer to the designated address in the testbench monitor that is written to `INT.OUTPUT`.

What this code will do is the following: It will initiate a write transaction on the AHB bus and send the integer in `nums[0]` to the custom logic block. It will then read a 32-bit word from the custom logic block in an AHB read transaction. Finally, the received word is written into the IO address space on the external memory bus. All this is repeated 10 times.

Below is the AHB bus waveform for this transaction, with `nums[j]` equal to “2”.



First, at  $t=13,180$  the write transaction:

1. Strobe=1 indicates that the custom logic block is addressed
2. we=1 indicates that data is written, i.e. data\_in is valid
3. The custom logic block outputs ack=1 because the data will be read at the next rising clock edge (to force the CPU to hold the data longer, ack can be delayed)
4. At the rising clock edge, “2” is read from data\_in
5. Also at the rising edge, the third power of the input data, “8”, is stored in data\_out. This data will be read by the CPU in the following read transaction

After 2 clock cycles, the CPU reads the result:

1. Strobe=1 indicates the valid address
2. we=0 indicates that the CPU is reading data
3. The data is ready at the next clock edge, therefore the custom logic sets ack=1. If more time is needed to provide the data, ack could be delayed.
4. The CPU reads the data on data\_out on the rising edge of the clock cycle.

This is a very simple application. More advanced applications can include:

- Multiple input and output registers by decoding the AHB address
- A FIFO to buffer streaming data
- Control and Status registers so the CPU can read the status of the custom logic and control its operation
- Delaying the ack signal to insert wait states. If a computation requires several clock cycles, “ack” can be delayed until the result is computed.

This avoids wasted clock cycles with the CPU waiting in a loop until data is ready.

- Signals can be added to connect the custom block directly to chip pins. For example to read in a data stream from an A/D converter.



## Summary

Following is a summary of the typical sequence of commands involved in designing on the SOCKS environment. It is assumed that the source code in this case is called "project1".

1. Creating a new source code folder
  - a. `cd ./firmware`
  - b. `cp -r bubblesort project1`
2. Creating and compiling the source code
  - a. `cd project1`
  - b. `[emacs | pico | etc.] leon_test.c`
  - c. `make`
3. Creating and compiling the custom logic HDL
  - a. `cd ../../hdl/custom`
  - b. `[emacs | pico | etc.] custom_top.v`
  - c. `cd ..`
  - d. `../exe/socks_compile`
4. Running the simulation
  - a. `cd ../sim`
  - b. `../exe/socks_sim project1`
5. Running the simulation and creating waveforms for the custom logic
  - a. `../exe/socks_sim project1 partial`
  - b. `simvision&`
6. Running the simulation and creating waveforms for the entire design
  - a. `../exe/socks_sim project1 full`
  - b. `simvision&`
7. Removing all temporary data to save space
  - a. `cd ..`
  - b. `exe/socks_clean`

The design process would be to repeat steps 2 or 3, each followed by step 4, 5 or 6 to verify the changes.

To keep different versions of the custom logic, it is easiest to keep them in different folders in `./hdl` and use the "mv" command to rename the active one into "custom".