University of Toronto
ECE532 Digital Hardware

# Module m05: Adding a User-Designed Peripheral

Version for EDK6.3 Jan/13/05

As of EDK6.3, there is now a wizard, the *Create/Import Peripheral Wizard* that can be used to add user-designed peripherals into an EDK repository or an XPS project. If you wish to add a peripheral that will be connected to the OPB or PLB bus, then this tool should make it much easier. At this point, a key limitation is that it only works for cores written in VHDL. See the *Embedded System Tools Reference Manual* for more information on this wizard.

This module was created before the *Create/Import Peripheral Wizard* was available. The value of doing this module is to get an understanding of some of the underlying directory structure of an XPS project and to see how to handle a peripheral that might not exactly fit the model of a typical peripheral. In this case, the core to be added also has a connection directly to the processor, not just to the bus.

## Goals

- Add a user designed peripheral to a basic MicroBlaze system. You will be provided with a core, called *snoopy* that is written using VHDL.
- Demonstrate the required structure necessary for interfacing user-designed cores to the Xilinx cores in an XPS project.
- Learn about how you would make your own core to attach to the OPB.

## Requirements

Module 1 Building a base system. Preferably, do Modules 1-4.

## Preparation

1. Review the handout outlining the EDK project structure provided in Module 1. We will be focusing on the pcores subdirectory for this module.
2. Look at Module m09, "Integrating a Verilog design into a MicroBlaze System". Much of the content is also applicable to creating VHDL cores. Pay particular attention to the section about "Adding the design to a MicroBlaze system".
3. If you are unfamiliar with the profiling of code, read the manual page for "gprof", which is available on the ugsparcs with the "man" command.

## Background

To this point, you have only been adding cores from the existing library. If you cannot find a core with the required functionality, you will have to add your own.

This lab adds a simple core that can be used to profile the code running on a MicroBlaze. This has similar functionality to the "gprof" utility for profiling that is available in the GNU tools. However, it is much more accurate. You may find it useful for your project.

## Step-by-step

1. Copy your working lab1 project into a new directory.

2. Add a pcores subdirectory to your new project directory.

   Using an ssh window, login to your ugsparc account. The files you will need for this lab can be found in `m05.zip`. Copy: 1) system.c to the code directory of your project; 2) the snoopy_v1_00_a directory into the pcores directory; and 3) the sst script and example_results.txt to the root of the project directory; 4) the OPBInterfaceModule directories to a directory where you can refer to them later. Note: You may find the "cp -r" command useful. As well, the "mv" command can also be used to "move" directories.

3. Take time to look through the directory structure of the snoopy core. The naming structure is essential for XPS to be able to detect a user's peripheral. All user cores must be located in the pcores subdirectory or in a globally specified path to a peripheral repository. User cores can be defined using either VHDL or Verilog, but any one core cannot have a mix of HDLs.

   Along with the hdl files used to implement the core, the user must also include data files: a .pao file (Peripheral Analyze Order) and an .mpd file (Microprocessor Peripheral Description). The .pao file lists the order in which files in your design should be synthesized to resolve component architectures. The .mpd file describes the external interface of the core to a system. For more information on these files and their structure, go to the Embedded System Tools Guide.

   Notice that the version numbers in the core name and the version numbers in the data file names differ. The version in the core name is the core's version. The version in the data file names is the version of the syntax used to write the data file.

   The easiest method for including user IP into an EDK project, is to follow an example. When you develop your own cores for your project, you can use snoopy as a guide. The cores provided by Xilinx in the `INSTALL_DIR\hw\XilinxProcessorIPLib\pcores` directory may also be used as a reference.

   You are also encouraged to look around through the directories in the EDK installation because there is a lot of source code available that might help you or guide you with your own designs.

4. The snoopy core is a snooping profiler that is able to profile software running on a softcore processor in real time. The counters calculate the exact number of clock cycles spent executing contiguous address ranges. The user specifies the number of counters and the lower and upper bounds for each counter before synthesis. This information can be used by embedded system designers to determine which, if any, sections of the software should be moved to hardware to achieve the required design specifications.

5. Open the Add/Edit cores menu. The snoopy core should appear in the list of peripherals you can add to your design. Add the snoopy core and connect it to the slave opb bus. The core requires 0x100 bytes and a 0x100 byte alignment. Since the tools resolve connections to the opb and lmb buses based on the address of a peripheral, we suggest address range 0xffffff00 to 0xffffffff to guarantee the peripheral resides on the opb.

6. For the snoopy core to work, it must be interfaced with the system clock, and the PC_EX and valid_instr ports on the Microblaze core. Connect the clock from the snoopy core (OPB_Clk) to the system clock. Don't forget to check the net name and the scope under the Ports tab. Unfortunately, the PC_EX and valid_instr ports on the snoopy core are not visible through the Add/Edit core interface. Therefore, they will have to be added manually to the hardware description.

7. Close the XPS GUI. Go to your project directory and open the system.mhs file. You are going to edit this file by hand. It is important to remember that this file contains the project description used by XPS to generate your MicroBlaze system. Therefore, it is safest to only edit the mhs file when XPS is not running.

8. You are going to add two lines of code to the MicroBlaze and snoopy module descriptions. Each module description begins with a BEGIN <module_type>. Look for the MicroBlaze core and the snoopy core and add the following two lines just before both END statements:

   ```
   PORT PC_EX = PC_EX
   PORT valid_instr = valid_instr
   ```

9. Save and close the mhs file. Restart XPS and open your current project. Go to the Add/Edit cores menu. If you look under the ports tab, you should now see that the valid_instr and PC_EX ports of the processor have been connected to snoopy.

10. The core lets you set a reset address for clearing the counters, the number of counters you want to use (maximum of 16) and the upper and lower bounds for the instruction addresses. Choose an address that is within the address range assigned to the core. (Hints: if you used the suggested address range for the core, the default reset address, 0xffffffe4, will be fine. Remember this is found under the Parameters tab.)

11. Remove the old system source file and add the new source file you copied into the code directory, system.c. Compile the source program without optimizations and including debug flags to see the generated executable. If you get an error, go to the compiler options menu and verify that the executable is being placed in the appropriate directory. Disassemble your executable into a file.

12. Open the disassembled file to determine the address ranges you will profile. You will be selecting contiguous address ranges to profile based on function calls. Edit the parameters for the snoopy core to profile the following functions:

| **Counter** | **Function** |
|---|---|
| 0 | _start |
| 1 | exit |
| 2 | _crtinit |
| 3 | main |
| 4 | _exception_handler |
| 5 | _interrupt_handler |
| 6 | _program_clean |
| 7 | _program_init |
| 8 | print |
| 9 | putnum |
| 10 | outbyte |
| 11 | XUartLite_SendByte |
| 12 | XUartLite_RecvByte |
| 13 | All Functions (complete program) |

The lowerbound should be the starting address of the function and the upperbound should be the address of the last instruction in the function. Change the default value for the NUM_COUNTERS parameter to 14

13. Now you can generate the bitstream to download onto the FPGA.

14. After the bitstream has been downloaded onto the FPGA, start up the XMD window. Connect to the XMD stub and then download the executable onto the board using the XMD download command dow. Remember that when you start up the XMD window you will be in the project root directory and the executable.elf file is in the `mb0_default` subdirectory.

15. The sst script you copied to your project directory can be used to read the counters. To reset them, use the memory write command in the XMD window: mwr reset_address <value>, where the reset_address is one of the core parameters. The counters are designed to reset independent of the written value. Type "source sst" and a file res.out will be generated. Open the file res.out to view the values stored in each of the 64-bit counters. The most significant portion is stored in 0x00 and the least significant portion in the 0x04 address. As you have just reset the counters, all the values should be zero.

16. Open the GDB Debugger and connect to the target. The assembly listing of the main routine should be visible, with the program counter set to 0x500. Set a breakpoint at the very last assembly instruction in main, and execute the program using the "Continue" button. Remember, do not click the "Run" button as GDB will try to re-download the code before execution. This feature does not work due to a bug in the tools.

After you have run your application, you can check the new counter values again using the sst script. Open res.out to see what the values for each counter are. Are they all non-zero values? Why or why not? (Hint: look at the disassembled code to understand the values.) Each counter is 64-bits wide, and the numbers in res.out are given in big-endian format.

17. Now copy res.out to res.bak. Reset the counter values as previously instructed and rerun the program. Run the sst script again and look at the values of the counters. Are they all the same as the values in your res.bak file. Why or why not? You should compare your results to those found in the file example_results.txt to verify the counter values.

18. You can also use the counters to determine how many clock cycles are required to execute a single instruction by setting both the upper and lower bounds to the same address. If the instruction is executed only once, the number of clock cycles should be the same as what is specified in the MicroBlaze processor manual. Familiarize yourself with the available XMD commands or change the core parameters to better understand how your application runs on the MicroBlaze processor.

19. Look at the OPBInterfaceModule files that you can use as templates when creating your own core for the OPB. Your Verilog or VHDL code will go in the subdirectory of the "hdl" directory. You will need at least an .mpd and .pao file in the "data" directory. If your core makes use of components that are in netlist format instead of HDL, you will need a .bbd file.

## Look at Next

Module 6: Using ISE
Module 8: Using ZBT Memory
Module 10: Using FSLs
Module 12: Using ChipScope
Module 13: MBlaze MP3 Decoder