Version for EDK 14.2 as of January 9, 2013

Acknowledgement

This module is derived from a Xilinx lab given at the University of Toronto EDK workshop in November 2003. Many thanks to Xilinx for allowing us to use and modify their material.

Goals

- Use Xilinx tools to build and debug a basic MicroBlaze system. This will consist of a MicroBlaze processor, memory, and a UART.
- Understand basic concepts of the Xilinx Embedded Development Kit (EDK), which includes tools such as Xilinx Platform Studio (XPS) and processor IP.
- Explore some concepts used when programming in an embedded processor environment such as where a program is loaded, how it is loaded, what gets added to it (runtimes, *etc.*), and how to interact with it.
- Use some software debugging tools in an embedded processor environment.
- See a little bit of how variables in C are mapped to physical memory. Often useful to know when debugging.
- Get an idea of how to find various useful documentation.

Requirements

You'll need access to:

- The Xilinx EDK 14.2 and ISE 14.2 tools.
- ATLYS board from Digilent.
- About 45MB of disk space for the project files

Documentation

You can find most of the Xilinx documents online. If you are running XPS already, you can use the menu link:

Help⟶EDK Online Documentation

or just search for `EDK Documentation` in the search bar at www.xilinx.com. Be sure to select the correct version of the documentation. You can use the Jump to pull down menu to get to the right place. The *EDK Concepts, Tools, and Technologies* document is a good place to start. It also has more tutorials.

Note: On some Linux installations, the menu entry may not start the browser. It is because you are not using one of the officially supported Linux versions (Red Hat and Suse). It seems that not all the dynamically linked libraries are in the same directories in each Linux version so you will need to figure out what is missing and add some links. Help for Ubuntu 12.04 will be posted and may give hints if you are using a different Linux version.

You can also find some useful documentation under `$XILINX_EDK/doc/usenglish/index.htm` in your installation that you may not find online.

Note

Some of the activity in this module does not require hardware and can be done later, such as examining various files. If time is running short, it is best to leave these steps till later and focus on the steps that actually use the hardware.

The steps in this module assume you are running the tools in the Microprocessor lab on the PCs.

Background

The Base System Builder (BSB), a wizard in XPS, can help you build your first system quickly and easily. XPS uses the Xilinx Integrated Software Environment (ISE) tools to synthesize, place, and route the hardware design. GNU tools are provided in EDK and are used along with XPS to build the software for the embedded system.

Using XPS Base System Builder

1. Create a directory for your modules in your home directory (`W:\` in the ECE labs). **Note: Make sure that the path to your project directory has no spaces.** `W:\ece532\` is probably a good choice. In this directory, you should unzip the `lab1.zip` file available from the UofT EDK page. You should now have a `W:\ece532\lab1\` directory in which you'll create your project for this module.

2. Start XPS by going to Start → Programs → Xilinx ISE Design Suite 14.2 → EDK → Xilinx Platform Studio.

3. Once Xilinx Platform Studio has opened, a window should appear that describes several methods of loading a project. Open File and select New BSB Project.... The Base System Builder is a wizard that helps minimize the effort required to generate a system by building the necessary data files for XPS.

4. The Create New XPS Project Using BSB Wizard dialog box is displayed. Browse to the directory named `lab1` that you unzipped into your project work area and select it in the dialog box. Click Save on the dialog box to select the directory.

5. Select AXI System under the Select an Interconnect type dialog box. If you are using the ATLYS board go to the next step. Otherwise, click OK.

6. This step is for the ATLYS board. You will need to have downloaded the ATLYS board support files from `http://www.digilentinc.com/Data/Products/ATLYS/Atlys_BSB_Support_v_3_6.zip`. Note that in the
   `Digilent_AXI_IPCore_Support_v_1_33` directory there are installation instructions.

   In Set Project Peripheral Repository Search Path browse to find the `lib` directory in the `Atlys_AXI_BSB_Support` directory. Select `lib` and click Choose. Click OK.

7. The Board and System Selection dialog box is displayed. Select Create a System for the Following Development Board under the Board heading. If you are using the ATLYS board, it should already be selected.

8. Under the Select a System heading, choose Single MicroBlaze Processor System. Under the Optimization Strategy heading, choose Area. Click Next.

9. The Processor, Cache, and Peripheral Configuration dialog box is displayed. Check that the clock frequency is 100.00 MHz.

   Under Processor Configuration, change the Local Memory Size to 64KB.

10. Under the Select and Configure Peripherals heading, there are several interfaces and peripherals available. For now, keep only the RS232_Uart_1 interface. The default settings for the UART (9600 baud, 8 data bits, no parity) are OK. The UART peripheral will be used for standard I/O. The standard I/O libraries delivered in the EDK use the UART in polled mode, so do not select the Use Interrupt checkbox. Remove all other devices that have been included by default (e.g. Ethernet, LEDs, DIP switches, pushbuttons, DDR, etc.).

11. Click Finish. This will generate the system project and design files.

At this point, the Base System Builder has generated a user constraint file (`system.ucf`) in the data subdirectory as well as a project file (`system.xmp`), a microprocessor hardware specification file (`system.mhs`), and a microprocessor software specification file. These files are accessible on the left hand side of XPS in the Project tab or by just looking in your `lab1` directory.

Building The Hardware

12. Select the Hardware menu and the Generate Bitstream submenu in XPS to start building the hardware system (*Hint: there's also a button for this on the toolbar*). This will take at least five minutes as the system is synthesized, mapped, placed, and routed for the FPGA. During the build process, a lot of information will be displayed in the bottom window panel of XPS.

    Note: If, under a Windows installation, you get the error

    ```
    ERROR: ConstraintSystem:8 - The file 'system.ucf' could not be opened for reading.
    ```

    follow these steps:

    - Set an environment variable with the name CYGWIN and the value nontsec
    - Launch a shell with Project → Launch EDK shell
    - Change your project file permissions by typing `chmod -R 777 *`

    XPS generates the system HDL file and wrappers for the cores used in your design and then invokes the Xilinx ISE tools to synthesize (`xst.exe`), map (`map.exe`), place, and route (`par.exe`) the design. When this step is complete, a `system.bit` file is created (via `bitgen.exe`) in the `implementation` subdirectory of the XPS project.

    Note: You may get an error regarding the XST synthesis. This relates to the version of cygwin installed on your system. To resolve this, right click on My Computer on your Windows desktop and go to: properties → Advance Tab → Environment Variables. Add a new System Variable called "SHELLOPTS", assign it the value "igncr", and click OK. Reload XPS and try again.

Defining The Software

13. Once the Hardware bitstream generation is complete (i.e. the console displays Done!), select Project on the menu bar and click on Export Hardware Design to SDK... (*Hint: there's also a button for this on the toolbar*).

14. The Export to SDK/Launch SDK dialogue box will appear. Make sure that the Include bitstream and BMM file option is checked. Click Export and Launch SDK.

    SDK is where you shall create all of your software applications that will run on the hardware that you generated using XPS. This is a useful tool that allows programs to be debugged very easily.

15. The Workspace Launcher dialogue box will appear. Set the `workspace` directory to be in the `lab1` directory that you created earlier by entering `yourpath/lab1/workspace`. Make sure that the Use this as the default and do not ask again option is **NOT** checked.

    Click OK.

Creating an Example C Project

16. In the SDK window, select File, New, Project and then click on Xilinx C Project in the New Project window. Click Next.

17. Under Target Software choose standalone and under Select Project Template choose Hello World.

    Click Next.

18. Select Create a new Board Support Package project. This will create all of the board support files (e.g. header files) that are supported by the project and the peripherals that you created using XPS, as well as the C project that will print out Hello World.

    Click Finish to generate the Board Support Project (BSP) and the C project.

19. In SDK under the Project Explorer tab, you should now see the generated BSP (`hello_world_bsp_0`) and the C project (`hello_world_0`).

    You should now see a message in the console that says `Finished building: hello_world_0.elf.elfcheck`. This means that the projects have generated successfully.

    Look through these projects. They contain all the necessary header files and source code that are needed to run the C project on hardware.

20. Return to XPS. In the menu, select Project then select Select Elf File.... A Select Elf File dialogue box will appear. Under Choose Implementation Elf File, select the option that allows you to browse through your directory.

    Under `lab1/workspace`, there should be a directory named `hello_world_0`. Open this directory and then open the directory named `Debug`. There should be a file named `hello_world_0.elf` in this directory. Select this file.

    Do the same for Choose Simulation Elf File and click OK.

    This will cause the `hello_world_0.elf`, which was generated for the software, to be inserted into the hardware bitstream. By doing this, the BRAM memory will be initialized with the program when the hardware is downloaded to the FPGA.

21. Select the Device Configuration menu and then Update Bitstream.

    This step connects the C project that you created using SDK to the hardware that you generated using XPS. The product is a .bit file called `download.bit` to be created in your `lab1` directory under `implementation`. This is the file that will be downloaded onto the FPGA.

    Question: You should also see that there is another file called `system.bit` in the same directory as `download.bit`. What do you think the differences are between these two bit files? (*Hint: the system.bit file was created when you first generated the bitstream*).

Using More GNU Tools

22. Select the Project menu and the Launch Xilinx Shell... submenu. This will start a Bash shell under Xygwin.
    Note: From the `workspace` directory, change to the `hello_world_0/Debug` sub-directory where you should find the `hello_world_0.elf` file. If you are running this on a Linux system, you can just invoke the command in your command shell.

23. In the Bash shell window, type:

```
mb-objdump -d hello_world_0.elf > disassembly.out
```

to disassemble the Executable and Linking Format (ELF) file and save the results in a file named `disassembly.out`. Open this file with your preferred editor to view the disassembly. You might want to look at the MicroBlaze Processor Reference Guide to help you understand the assembly language instructions.

The disassembly file shows the machine code stored at each memory location and the corresponding assembly instruction. What is the address of the function `_start`?

What is the address of the procedure `main`? This corresponds to `main` in the C program.

Why do you think the program is linked to start where it does?

Can you see where the stack pointer (`R1`) is set?

There are a number of activities that are done in the C run time module, which is linked into your program before your main routine. Your program actually starts execution in the C run time module to set things like the stack pointer and to zero the bss segment. Uninitialized variables in C are supposed to be set to 0 before execution of main starts. The C run time modules can be found in the EDK installation directory $XILINX_EDK/sw/lib/microblaze/src. Here you will find a number of run time (`crt`) files. Look at `crt1.s` and `crtinit.s` and compare them to what you find in the `disassembly.out` file you just created.

24. If you have a number of memory banks and you want your code and data to be loaded in different banks, you will have to generate a linker script. Select `hello_world_0` in the SDK Project Explorer tab and click on Generate linker script in the Xilinx Tools menu. Have a look at the Basic and Advanced tabs to see what things you can change.

Downloading the Bitstream to the FPGA

25. In the bottom middle window of the SDK, there is a Terminal tab which can display serial output from the board. Use the Settings button on the right to pick the appropriate COM interface and choose parameters. The correct terminal settings are 9600 baud, 8-N-1, no flow control. This matches the settings from the IO dialog box during the Base System Builder Wizard system creation. Check that you have a USB cable connected from the UART USB port on the board to the PC. Press the Connect button (If you are not sure which COM interface is your USB cable, try out which interface connects and gets you serial output after downloading the bitstream.)

26. Make sure that the USB programming cable is connected from the PROG USB port on the board to the PC and that the board is powered on.

    The micro-USB connectors are more fragile than the regular USB cables so please take care when inserting and removing the cables.

    Select Download Bitstream from the Device Configuration menu in XPS. This will download the hardware and software contained in the bitstream to the FPGA. The ROM monitor software will begin executing after the download completes. This may take a few seconds or a few minutes, depending on the mode in which the parallel or USB port is operating.

    The FPGA Done LED also illuminates when programming is completed. If you see an error/warning in the XPS output window indicating that the Done pin did not go high, try downloading again a couple of times before messing with the cable settings.

    Once the FPGA is programmed, it starts running the `helloworld.c` program. If your terminal is properly connected to the board before you download the bitstream, you should see `Hello World` printed on the terminal window. Each time you hit the Reset button on the board, the program is run again and another `Hello World` should be printed on the terminal.

27. You could also have programmed the FPGA from the SDK and save a few steps. In the SDK window, select the Xilinx Tools menu and then Program FPGA. In the Program FPGA window, the Hardware

Configuration should point to the system.bit file and a system_bd.bmm file, which describes the address map for the processor and the physical BlockRAMs used in the FPGA. For the Software Configuration, be sure that it points to the `hello_world_0.elf` file.

Debugging with XMD

28. In the XPS window, select the Debug menu and select the XMD Debug Options... submenu. We must select which debug method we will be using for our program. Ensure that MicroBlaze_0 is the processor selected. Click OK.

29. Select the Debug menu and the Launch XMD... submenu. This will start Xilinx Microprocessor Debug in a new Xygwin window. This program communicates with the board via the JTAG connection (either the Parallel IV cable or the USB cable). XMD will automatically connect to the hardware debug module, the MicroBlaze Debug Module (MDM), instantiated with the processor. The results should indicate that it connected successfully and that a GDB server was started. GDB (the GNU Debugger) is the software debugger that will be used to debug software for the system.

    Type `help` to get a list of commands and type `help running` to get a list of more detailed execution commands.

    NOTE: If XMD did not launch, open up a terminal and type the command `xmd`. Then type the command `connect mb mdm` to connect to the MicroBlaze processor. If successful, you should see a TCP port number displayed.

30. With XMD, you can now read and write memory addresses, access the processor registers, do low-level debugging at the assembly instruction level and control the execution of the program.

    Type `help` to see a list of command types and `help running` to see the commands you will most likely use for now.

    In the XMD window, read the contents of memory location 0 based on the commands displayed in the help. Try the command `dis 0 12`. What does the `dis` command do? You will find the `dis` command under `help misc`.

    What are the contents of memory location `0`? Is this what you would expect? To help answer this question, go back and look at the `disassembly.out` file you created in Step 23.

    What you have been doing in this step is examining the executable object file (`*.elf`) in a simple way, which gives you an idea of what should be loaded in memory and the address for some of the labels/routines. Using XMD is a very low-level interface for debugging your code, but it is more likely to be telling the truth. You will shortly also use a symbolic debugger, GDB, which adds a layer of abstraction and is a lot more powerful. However, if in doubt, then you can always resort to the XMD interface — if something weird is happening, the simplest interface is likely to be the most reliable.

31. At what address is `main`, the start of the `main` procedure of the C program? Verify that the code in memory at that address corresponds to what you expect. You will need to prepend addresses with `0x` to indicate that they are in hex notation.

Starting a New C Project

32. Let's start a new C project running on the same hardware. You should contrast these steps with what you did starting at Step 16.

    In the SDK window, select File, New, Project and then click on Xilinx C Project in the New Project window. Click Next.

33. Change the suggested project name to `counter`.

    Under Target Software choose standalone and under Select Project Template choose Empty Application. Click Next.

34. Select Target an existing Board Support Package and click on `hello_world_bsp_0`, which was created previously. You are now saying that you want to use a previously generated hardware configuration. Click Finish.

35. Under the Project Explorer tab you should now see the `counter` project. Expand it and right click on the src folder. Select New → File. The New File dialogue box will open. Click the Advanced button and check the Link to file in the file system option. Then, browse to lab1 → code and select `lab1.c`. Click OK and then Finish. In the Console you will see that the new project is then compiled.

    Make sure your terminal window is open and then download (Xilinx Tools→Program FPGA) and watch the program run in your terminal window. Be sure to select the `counter.elf` file under Software Configuration in the Program FPGA window.

Debugging Software

36. This new C program has a loop and a simple procedure so that some debugging features can be demonstrated.

    In the SDK window, select Run → Debug Configurations....

    The Debug Configurations window should open. On the left hand side, double click on Xilinx C/C++ ELF to enable the `counter` project to be debugged.

37. Switch to the Remote Debug tab in the window and make sure that the IP Address displays localhost and that the port number is the same as the port number that was displayed when you launched XMD. Most likely, it will be port 1234.

    Click Debug.

38. At this point, you should see a Confirm Perspective Switch window open. Click Yes. This will now reconfigure the windows for debugging. You can go back to the previous perspective by going to the top right corner and clicking the Open Perspective button beside the button that says Debug.

    If you have not done so already, launch HyperTerminal or Minicom so that you can view the output of the code.

39. You should now be able to debug the C code for the counter project.

    In the lab1.c tab you'll see the source code of the program. If you do not see line numbers, right click in the ribbon at the left edge of the source code window. Select Show Line Numbers.

    Now set some breakpoints. Right click in the ribbon at the left at Line 11 where `Number` is decremented in the procedure. Select Toggle Breakpoint. You should now see a blue dot appear next to Line 11. Set additional breakpoints at Line 22 and Line 26.

    Also add some variables to watch. Right click in the `lab1.c` source code window and select Add Watch Expression. (If the Add Watch Expression does not appear, return to the Run menu and click Debug As → Launch on Hardware and say Yes to terminating the previous launch.) Enter `Counter1` and Ok. You will see `Counter1` appear in the Expressions tab. Also watch `Counter` and `Number`. Note that you can add new expressions to watch directly in the Expressions window. You should see that there is an error in the value for `Number`. More about this later.

    Note, there is also a Variables tab that shows the current local variables and global variables if you add them. We'll ignore it here, but it is probably what you would normally use just to observe the values of variables in your program.

40. Now, you are ready to start execution. In the SDK Run menu, select Debug As → Launch on Hardware, if you did not already do this above. This will start the execution of the program in debug mode and you will see the program paused at Line 17. Note the little blue arrow at Line 17.

You should see `Counter1` with the initial value 20, while `Counter` has some random value because it has not been initialized yet.

At the top of the Debug tab you should see some execution control buttons. The controls are also available in the Run menu. Find the Step Into and Step Over buttons.

Click Step Over once and you'll see execution move to the `xil_printf` at Line 19. Note that `Counter` now has been correctly initialized to 30.

Click Step Over again. This executes the `xil_printf` procedure in one step, i.e., without showing you all the statements executing inside `xil_printf`. Observe on your terminal window that a new line has been emitted.

41. Continue with Step Over until you reach Line 23 where `Counter` will be decremented. Observe that `Counter1` was decremented at Line 22. Now, use Step Into to observe execution of the program inside the `Decrement` function. Observe that `Number` now has a value.

42. Continue to use Step Over until you see output on the terminal and you are back at the start of the loop. Now find the Resume button and use it to allow the program to run until it hits a breakpoint. Do this for an iteration or two of the loop. Observe that the program is executing as expected.

Digging a Little Deeper

At this point, you have tried some basic source-level debugging of a C program running on the MicroBlaze. This section will help you make a connection between what you see at the C language level and what is happening in memory. Understanding the relationship is necessary and important if you want to become an embedded programmer.

First, some observations about the `lab1.c` program:

`Counter1` is a global variable (declared before `main`) so it will have a static memory address. It is an *initialized variable*, meaning that it will be assigned a value when the program is **loaded in memory** by storing the appropriate value in memory at the right location, not when the program is executed.

`Counter` is declared as a local variable inside `main` so it will be allocated space in the stack frame for `main`. `Counter` is also declared with an initial value, but because it is allocated on the stack, it will be initialized when the program starts to execute.

`Number` is a *call-by-value* parameter of the `Decrement` function so it will also be allocated space on the stack each time the function is called.

43. Can you now explain why the value for `Number` sometimes shows an error and other times shows a proper value in the Expressions window?

44. Find the `counter.elf` file in `workspace/counter/Debug`.

    In a Bash shell window, type:

    ```
    mb-nm counter.elf > symbols.out
    ```

    to display the symbols in the C program and their corresponding values. You may want to use the `-n` flag to get the output sorted numerically. On a Unix system, you can just type `man nm` to see how to use the command. `nm` is a standard gnu utility. The `mb-nm` command is just the MicroBlaze version of it.

    The `nm` command is used to dump the symbol table of the object file. Here you will see the address of various symbols in your program such as the start of subroutines, location of global variables, and other internal symbols. This command is often useful for finding the memory location of your symbols, especially if you need to use `XMD` to look at something.

Open `symbols.out` with your preferred editor to view what is inside.

What is the value for `Counter1`? The `D` means that `Counter1` is in the initialized data section of the program. The other value is the address of `Counter1` in memory.

45. Continue executing the program until it hits the breakpoint at Line 22 where `Counter1` is decremented.

46. Close any open `XMD` window you have in a bash shell and open a new one from the SDK Xilinx Tools menu. You will need to do this to make sure the `XMD` window stays in sync with the rest of the debugging windows. Using `XMD` find the value of `Counter1` by reading the memory address for `Counter1` you found by reading the symbol table. You will need to prepend the address with `0x` to tell `XMD` it is a hex number. The value in memory should correspond with what you see in the Expressions window.

47. Using `XMD` change the value of `Counter1`, which is a global variable. With the Expressions window open, Step Over Line 22 so that `Counter1` is decremented. Explain the value of `Counter1` now shown in the Expressions window. What have you just done? Note that you can also change the value of `Counter1` in the Variables window, but the use of `XMD` is to emphasize the relationship of the variable with an address in physical memory.

48. In this step, we'll start to muck around with the variables in the stack. This is not normally recommended, but playing around like this will give you a better understanding of what's going on *under the hood* of the C program. For more information about how the stack is used, look in the MicroBlaze Processor Reference Guide under Stack Convention.

    Step the program until it is at Line 22 where `Counter` is about to be decremented. Now, let's inspect the stack. The stack pointer is `R1`. Find the value of `R1` using XMD. It can also be found using the Register window, but let's stick with `XMD`.

    The value in `R1` will be the address of the top of the stack. Using `XMD` dump the top 10 values of the stack. Can you find where `Counter` is stored in the stack? Hint: You can do this by looking for the current value of `Counter` in the values you dumped. It may appear twice, but pick the one that has the higher address, meaning it is *deeper* in the stack. Change the value at that address using `XMD`. Inspect the stack again to verify your change.

49. Now, step your program until you get into the `Decrement` procedure. What is the value of the stack pointer now? Why has the value changed? Inspect the stack again and observe what has changed. How is the parameter of the function being passed?

50. Do one more step to execute the decrement of `Number`. Inspect the stack again. What has changed? What has happened?

51. Step out of the function call and inspect the stack. What has changed? What has happened?

52. The next few steps illustrate how an initialized global variable works. Reload the FPGA and restart the program using Xilinx Tools → Program FPGA. Observe that you get output on the terminal screen.

    When you hit the Reset button on the board, it causes the program to run again from the beginning. Hit Reset and see what happens. Can you explain why the output is different? Hint: Using `XMD`, check the value of `Counter1`, which determines the exit condition of the loop.

    Note that you will have to reconnect `XMD` with the processor using `connect mb mdm` before you can use any other commands because the reprogramming disrupts the connection.

53. Write some nonzero value into the address for `Counter1`. Hit Reset again. What do you see? Explain.

54. Modify your program by adding the statement:

```
Counter1 = 20;
```

before the start of the loop. Save the change and program the FPGA with this new version. Run the program to see the correct output. Hit Reset and see what happens. Why is this different than what was observed in Step 52?

55. Now, a few brief words about compiler flags. In the SDK you are probably still in the Debug perspective. Change back to the C/C++ perspective by clicking on the Open Perspective button in the top right corner to the left of where it says Debug. In the Project Explorer window right click on the `lab1.c` source file and select Properties. Here you can change the way this source file is compiled.

    Under C/C++ Build→Settings, click on Optimization. You will see that no optimizations are turned on, but you can change it here.

    Click on Debugging and you will likely see that it is at the maximum debugging level. At this level, debug symbols will be put into the `elf` file and it will prevent optimizations so that symbolic debugging can be done from the original C code.

Exploring some files (`system.make`, `system.log`)

You have been working with the GUIs, which hide a lot of the underlying details. This, of course, makes things a lot easier when things work. When things break, you will need to look more deeply. Once you get more comfortable with how things work, you may bypass the GUI completely.

Also, in the long run, especially for large projects, you will need to have some reproducibility when you run the tools to know that the changes that occur are because you fixed some code, rather than because you pushed the buttons in a different order.

In these situations, you will want to investigate how the scripts work and ultimately use them to run your compilation and synthesis. XPS gives you a good start and creates a makefile called `system.make`, which you can find at the top level in your project directory. When you push particular buttons, you'll actually invoke actions that are found in `system.make`. Have a look through the makefile.

Everything that you see in the log window of XPS gets put into a log file. It is called `system.log` and is found at the top level of your project directory. Every time you start up XPS and do things in that project, the output is added to the `system.log` file. You might want to keep an eye on this file as it could grow quite large.

Have a look through the `system.log` file. See if you can find out the utilization of the FPGA and how fast you could actually clock it.

When you are done

Please take care when packing up the kit.

1. Disconnect all cables and place them in the box.
2. Place the board in the anti-static bag to protect the board.

Summary of the structure of the EDK project directory

The following should be used as a reference to aid you in finding information about your MicroBlaze system.

`system.mhs` A higher-level description of the hardware modules in the system.

`system.xmp` The system project file used by XPS. We suggest that you should not edit this file outside of XPS, but if you choose to do so please use extreme caution. When returning to the project, this is the file to open.

`SDK/SDK_Export/system.xml` An XML description of the hardware system that is exported by XPS and then used by SDK to build a board support package (essentially a driver library).

`__xps/` Options used by different tools.

`data/` Contains the user constraint file (`.ucf`) which assigns external pins to ports, sets clock speed, etc.

`etc/` Contains `download.cmd` and `fast_runtime.opt` (not important to general designs).

`hdl/` Generated by XPS. Contains the upper level system file and wrappers for each of the peripherals.

`implementation/` Contains the synthesis files, bit files and initialization files for the BRAMs.

`microblaze_0/` Instance of the MicroBlaze processor. This is found in the `bsp` directory since it is part of the hardware platform.

- `code/` Has the source code run on this particular instance of MicroBlaze (both `.s` and `.elf` files).
- `include/` Contains the drivers, header files, and the `xparameter.h` file. The `xparameter.h` file will be referenced in future labs and is used to program the drivers.
- `lib/` Has the standard libraries `libc.a`, `libm.a` and `libxil.a`.
- `libsrc/` Contains the library source code for the drivers, MicroBlaze, etc.

`synthesis/` Has the output from XST.

`pcores/` User designed peripherals can be added to designs as cores using a specified directory (an example will be provided in future labs).

Look At Next

Module m02: Adding IP and Device Drivers — GPIO and Polling