Version for EDK 10.1.03 as of January 7, 2009

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.
- Get an idea of how to find various useful documentation.

Requirements

You'll need access to:

- The Xilinx EDK 10.1.01k and ISE 10.1.01k software with IP Update 3.
- Xilinx XUP Virtex-II Pro Development System.
- About 15MB of disk space for the project files (at least 12MB, plus some to spare).

Preparation

You should have a quick look at the following documents. There are links from the UofT EDK page to the Xilinx site for the Xilinx documents and the link for the Training Lecture is there as well (see Tools Documentation, Resources and Access).

You can also find the Xilinx documents in your EDK install directory. Some are online at Xilinx, but others are only available in your installation.

If you are running XPS already, you can use the menu link:

Help⟶EDK Online Documentation

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 it 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 the EDK and are used within XPS to build the software for the embedded system.

Setup

With the Xilinx XUP Virtex-II Pro Development System, your kit should include:

- Xilinx XUP Virtex-II Pro Development System Board
- Documentation and driver CD
- Digilent VDEC1 video decoder board
- USB A-B Cable
- USB to RS-232 adapter cable
- AC adapter

Step-by-step

Setting up the Hardware Connections

For the Xilinx XUP Virtex-II Pro Development System Board

Please be **very** careful when setting up the hardware so as not to break the connectors. **Do not power on the board until a TA has verified your hardware setup.**

If you are using the Xilinx Virtex-II Multimedia Board, consult the last subsubsection.

1. Place the development board on the table such that you can read the Xilinx insignia in the bottom right hand corner.
2. Connect the USB cable to the socket on the right side of the board towards the upper edge. This provides the JTAG connection that is used for downloading FPGA configurations and for debugging.
3. Connect the other end of the USB cable to the PC.
4. At the top left hand corner of the Ultragizmo board there are two grey cables labeled CON and D. Unplug the CON cable and connect it to the serial cable adapter. Plug the other end of the serial cable into the connector located at the bottom right-hand corner of the board towards the lower edge. This will be used for your UART connection.

   Note: you need two connections to your board to do this lab. First, you need a programming connection for configuring the FPGA. This is done using JTAG over either the Parallel Cable IV or the USB connection to the XUPV2P board. Second, you need a communications connection for interacting with the designs you instantiate on the FPGA. For instance, the standard input and standard output of the MicroBlaze in most of your designs will be connected to the serial port on the XUPV2P board. In other words, `printf()` and `scanf()` will run over the serial port. It's for this connection that you need the serial cable connected to the CON port.
5. Plug one end of the power supply into the power bar and the other into the jack located in the top left hand corner (just above the power switch).

6. Check that the DIP switches on the board are configured for JTAG programming. Now's a good time to leaf through the manual for the board (on the XUPV2P CD or online) if you haven't already — look for the section Configuring the FPGA....

7. Get a TA to check your connections. You can then turn on the power switch (ON and OFF are marked). The green power LEDs along the top edge of the board should be illuminated, as should be the green JTAG Config LED on the right side of the board. A red System ACE Error LED will also be flashing since the System ACE controller failed to load a configuration from a Compact Flash (CF) card. When you turn on your board, the PC it's connected to will detect a new USB device and may require that you install a driver for that device.

Using XPS Base System Builder

8. 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 `m01.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.

9. Start XPS by going to Start → Programs → Xilinx ISE Design Suite 10.1 → EDK → Xilinx Platform Studio.

10. Once Xilinx Platform Studio has opened, a window should appear that describes several methods of loading a project. Select Base System Builder Wizard and click OK. 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.

11. 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 Open on the dialog box to select the directory.

    If you are using the XUPV2P board, you'll need to check the Use Repository Paths option to import the board definition files and the drivers for the board's specific peripherals. These files are contained in `lib_rev_1_1.zip`, which is available on the CD that came with the XUPV2P kit and online. Extract the ZIP file into the directory you created for the modules directory (*i.e.*, `W:\ece532\`) and use the `lib/` diretory from that archive as your repository path (*e.g.*, `W:\ece532\lib\`). If you use the wrong path, you'll be presented with an error dialog. Remember to avoid spaces in your paths!

    Click OK on the Create New Project dialog box to start building the project.

12. The Base System Builder — Welcome dialog box is displayed. Select I would like to create a new design and click Next.

13. The Base System Builder — Select Board dialog box is displayed. With the XUPV2P board, select Xilinx as the Board Vendor, XUP Virtex-II Pro Development System as the Board Name, and C as the Board Revision. Click Next on the dialog box.

14. The Base System Builder — Select Processor dialog box is displayed. If you are using the XUPV2P board, you can select a PowerPC or a MicroBlaze processor. Select MicroBlaze for this experiment. Click Next on the dialog box.

15. The Base System Builder — Configure Processor dialog box is displayed. With the XUPV2P board, the Reference Clock Frequency is fixed at 100.00 MHz; set the Processor-Bus Clock Frequency to 25.00 MHz.

    Regardless of which board you're using, select XMD with S/W debug stub as your Debug I/F. This selection uses a ROM monitor debug solution, not a true JTAG debug solution. A ROM monitor debug solution assumes that software can execute on the platform to do debugging. Select 64 KB in the Local Memory panel. All of the data and instruction processor memory will then be implemented using the internal block RAMs of the FPGA. Click Next on the dialog box.

16. The Base System Builder — Configure IO Interfaces dialog box is displayed. There are several Interfaces available for us to select. For now, select only the RS232_Uart_1 interface (for the XUPV2P board). 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. Deselect all other devices (*i.e.*, Dallas 1-wire, Ethernet, System ACE, LEDs, DIP switches, pushbuttons, DDR, PS2, and audio for the XUPV2P board). Click Next on the dialog box.

17. The Base System Builder — Add Internal Peripherals dialog box is displayed. Internal peripherals include timers, interrupt controllers, and other devices that are typically used within the FPGA. Do not add peripherals at this time. Click Next on the dialog box.

18. The Base System Builder — Software Setup dialog box is displayed. Ensure that both STDIN and STDOUT are set to RS232_Uart_1 (for the XUPV2P board). Un-check the Memory test and Peripheral selftest checkboxes. Click Next on the dialog box.

19. The Base System Builder — System Created dialog box is displayed. Here you can view information about the system to be created. Click Generate to cause the system data files to be generated.

20. The Base System Builder — Finish dialog box is displayed. Click Finish on the dialog box to complete the wizard. The system is displayed in XPS and is ready to be built. In the dialog box that appears, select Start using Platform Studio and click OK.

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 (`system.mss`) in your project directory (`lab1`). These files are accessible on the left hand side of XPS in the Project tab.

Building The Hardware

21. 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 pane of XPS.

    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 that 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 select properties. Then select the Advanced Tab, and click 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

22. Double click on Add Software Application Project... in the Applications tab of XPS. For the Project Name, type `mb0_default`. Any name will do here, but this name is nicely descriptive: the project will run on the processor named `microblaze_0` (hence "mb0") and it is the default application for that processor. Click OK. A new project should appear in the Applications tab. In the list of Software Projects in the Applications tab, right click on Default: microblaze_0_xmdstub. Make sure Mark to Initialize BRAMs is checked.

23. Double click on Project: mb0_default. A dialog box will open to allow users to set compiler settings for the project. In the Environment tab, select XmdStub as the Application Mode. In the Debug and Optimization tab, uncheck the Generate Debug Symbols option. Click OK.

24. Expand the mb0_default project in the Applications tab of XPS. Right click on Sources and select Add Existing Files.... Browse to the `code` subdirectory of the project and select the `lab1.c` source file. Click open.

    The source files added are listed in Sources under the mb0_default section of the Applications tab.

Compiling the Drivers and Program

25. Select the Software menu and the Generate Libraries and BSPs submenu (*hint: again, there's a toolbar button for this task, too*). This will cause the drivers and startup code to be compiled into a library that will be used to link with the program.

26. Select the Software menu and the Build All User Applications submenu. This will cause the program source, `lab1.c`, to be compiled and linked. An `executable.elf` file is created in the `mb0_default` directory. This is the file that can be downloaded to the embedded platform.

    You may also see a `xmdstub.elf` file in the `microblaze_0/code` directory. This is the ROM monitor executable.

27. Select the Device Configuration menu and the Update Bitstream submenu. This will cause the xmdstub file, `xmdstub.elf`, that was generated for the software to be inserted into the hardware bitstream. By doing this, the BRAM memory will be initialized with the xmdstub when the hardware is downloaded to the FPGA. Later, you can use the xmdstub to download the executable, `executable.elf`, to the `MicroBlaze CPU` and run it. Please refer to the *Embedded System Tools Reference Manual* EDK10.1 Service Pack 3, Page 162, for more details.

Using More GNU Tools

28. Select the Project menu and the Launch EDK Shell... submenu. This will start a Bash shell under Xygwin. Change to the `mb0_default` directory, which should contain the `executable.elf` file. If you are running this on a Linux system (for instance, one of the ECF machines), you can just invoke the command in your command shell.

29. In the Bash shell window, type:

    ```
    mb-objdump -d executable.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.

    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 function `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 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.

    Disassemble `xmdstub.elf`. What is the address of the function `_start`?

30. Within XPS, where would you specify the program start address for the software application? (*Hint: look in one of the dialogs we accessed via the Applications tab.*) This is useful when you have multiple memory blocks at different addresses in your memory map. This is controlling a flag given to the linker/loader phase of the compiler.

Downloading the Bitstream to the FPGA

31. Go to the `Courseware` directory (folder) and open the `XILINXPORT`. On Linux systems, you can use `minicom` and on other Windows machines, you can use HyperTerminal. The correct terminal settings are 9600 baud, 8-N-1, no flow control. These have been pre-set for you in the lab and they match the settings from the IO dialog box during the Base System Builder Wizard system creation. Double check the serial cable connection between the board and the PC.

32. Ensure that power is on to the board. With the XUPV2P board, make sure that the USB cable is connected to the PC and that the board is powered on.

    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 (red on the XUPV2P board) when programming is completed. If you see an error/warning in the XPS ouput window indicating that the done pin did not go high, try downloading again a couple of times before messing with the cable settings.

Getting Ready to Debug

33. 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, and select Stub as the Connection Type. Click OK.

34. 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 software XMD stub running on the MicroBlaze. If we had chosen to use the hardware debug module, XMD would instead connect automatically to 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.

    At this point XMD is connected to the ROM monitor stub running on the target board. Type `help` to get a list of commands and type `help running` to get a list of more detailed execution commands.

35. In the XMD window, read the contents of memory location 0 based on the commands displayed in the help.

    What are the contents of memory location `0`? Is this what you would expect? To help answer this question, use the disassembler to examine `xmdstub.elf`. You can also try using XMD to disassemble a number of instructions in memory, say twelve, and compare the output with the disassembled output of `xmdstub.elf`.

    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.

36. Recall the start address for the `executable.elf` file that you found previously. In the XMD window, disassemble at memory location `0x800`.

    What is the assembly language instruction contained at location `0x800`? Is this what you expected?

Debugging Software

37. From the Debug menu, select Launch Software Debugger. This is simply a GUI interface that connects to the GDB debugger running on the MicroBlaze that can be used to debug code on the processor. In GDB, select the Run menu and Connect to Target submenu. A Target Selection dialog box is displayed. Select Remote/TCP : XMD as the target. Enter `localhost` as the hostname. Enter `1234` as the port. Ensure that Set Breakpoint at 'main' and Set Breakpoint at 'exit' are checked. Click OK. The processor will be stopped at a breakpoint at the beginning of the program.

    Return to your XMD window. By observing the response of this window, you can see that XMD is a GDB server. It accepts the TCP connection from GDB and facilitates debug between GDB and the target board.

    In your XMD window, check a number of the instructions at `0x800` and above. Here it is probably easier to use the disassemble command rather than the memory read command. What do you see now?

38. At this point, you should see assembly code in the GDB source window. The leftmost pulldown menu beneath the buttons allows you to view the source for code related to this program. Are you able to view the C language source code for `lab1.c`? Why not?

39. Exit GDB so that the code can be rebuilt. GDB holds files open that will prevent the code from being recompiled and linked. Similarly, make sure you are not holding open files or directories that need to be rewritten in your editor or Windows Explorer.

40. In XPS, select the Applications tab and double click Compiler Options. On the Debug and Optimization tab, select No Optimization for the program and select Create symbols for debugging (-g). This will cause debug symbols to be put into the `elf` file and will prevent optimizations so that symbolic debugging can be done from the original C code.

41. Recompile the program, download the program again and restart GDB. You should be able to view the source code for the `lab1.c` program.

42. Using the new `executable.elf` file, run the disassembler again to get a new listing. Save the original one. Using XMD and its disassembler, compare what you see in memory with what you see in the disassembled elf file. Hopefully, they are the same!

    If you have time, or at a later time, see if you can understand the assembly language generated by the compiler for your C program. Note that the optimizer is completely off, so the code is quite inefficient compared to the first version that you built. You may want to refer to the MicroBlaze reference manual to be able to interpret the assembly code.

43. In the EDK shell, try using the `mb-nm` command on your `executable.elf` file. 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.

44. Practice using GDB.

    First, set a breakpoint at line 5, 7, or 8 of the `lab1.c` in the source window. Add the counter of `lab1.c` to a watch window. Note that several breakpoints were set. Can you figure out where these breakpoints are?

    You can find more information on the GDB commands in the Embedded Systems Tools Guide.

45. Use the View Menu to start up other windows and see what they do.

More Trickiness

46. In the window where you are watching the counter, you can change the value of the counter by clicking on it. The counter is a local variable so it will exist as part of the stack frame for `main`. See if you can find out where in the stack the counter is located.

    Start with opening a window for the register values. Note that `r1` is the stack pointer. Dump about 16 locations starting at the stack pointer using XMD. Then change the counter value in the GDB window, and dump the stack again in XMD. Repeat a few times if necessary. Modify the appropriate stack location using XMD by putting in a different number. Step your program a few times and observe what happens.

47. Modify your C program to only do the loop 32 times. Compile and run it without any breakpoints.

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

You have been working with the GUI, which hides 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.

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. Everytime 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.
2. Disconnect all other cables and place them in the bottom of the bin.
3. Put the foam on top of the cables and then the board.
4. Make sure the serial cable is reconnected to the Ultragizmo 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.mss` A higher-level description of the software 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.

`__xps/` Options used by different tools.

`code/` Software source code run on processor.

`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:

- `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