University of Toronto
ECE532 Digital Hardware
Lab 1: Building a MicroBlaze System in XPS

Version 1.4 For EDK 6.2i 8/15/2004

Acknowledgement

This lab 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 your basic program (runtimes, etc.), how to interact with it.
- Use some software debugging tools in an embedded processor environment.

Requirements

Access to EDK 6.2i and the Xilinx Multimedia development board.

Preparation

You should have a quick look at the following documents. Links to them are available on the course web site:

EST Tools Guide, especially the sections on the GNU Compiler Tools and GNU Debugger.

sources.redhat.com/insight describes the GUI interface to GDB

MicroBlaze Processor Reference Guide The assembly language and instructions are described here.

Training Lecture These are the slides used at the Xilinx workshop. Flipping through them may be useful.

Note

Some of the activity in this lab 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.

Background

Building a system manually for the 1st time can be tedious, but Base System Builder (BSB), a wizard in XPS, can help to build your 1st 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 an embedded MicroBlaze system.

Setup

In your kit, you should have the following hardware:

- Xilinx Virtex-II Multimedia Board
- Xilinx Parallel Cable 4
- Straight-thru serial cable (or use a null modem adapter)
- Power cable

Step-by-step

Setting up the Hardware connections for the Multimedia 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.**

1. Place the Multimedia board on the table such that you can read the Xilinx insignia in the bottom right hand corner.
2. Attach the flying leads cable to the Parallel Cable IV pod such that the red (JTAG/SERIAL) lead is furthest away from you and the pod is faced so you can read the labels. The leads provide a JTAG connection that is used for downloading FPGA configurations and debugging.
3. Connect the cable from the pod to the parallel cable from the computer and the smaller lead to the adaptor dangling from the back of the computer. The smaller lead provides power to the pod. The status light on the pod should now be lit orange.

4. At the top left hand corner of the gizmo board there are two grey cables labeled " CON" and " D". Unplug the " CON" cable and connect it to the serial cable adaptor. Plug the other end of the serial cable into the connector located at the top left hand side of the board just below the power switch. This will be used for your UART connection (a CONPORT).

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. Get a TA to check your connections. You can then turn on the power switch (ON and OFF are marked). The LEDs on the board should now be on and many are likely flashing.

Using XPS Base System Builder

7. Create a project directory for your labs in your home directory (W:\in the Windows directory system). Make sure that the path has no spaces!!! In this directory, copy the lab1 directory from X:\Courseware\Xilinx 6.2i\User_Area

8. Start XPS by going to the Courseware folder and selecting the Xilinx 6.2i directory. Select the Xilinx Embedded Development Kit 6.2 folder and double click on the Xilinx Platform Studio icon.

9. Select the File menu, select New Project , select Base System Builder. A Create New Project Using Base System Builder Wizard dialog box is displayed.

10. Browse to the directory named lab1 you copied into your project work area and select it in the dialog box. Click Open on the dialog box to select the directory.

11. Click OK on the Create New Project dialog box to start building the project. A Base System Builder - Select Board dialog box is displayed. This may take a few minutes.

12. Select Xilinx as the Board Vendor.

13. Select the Virtex-II Multimedia FF896 Development Board as the Board Name.

14. Select revision 1 for the Board Revision.

15. Click Next on the dialog box. A Base System Builder - Select Processor dialog box is displayed.

16. The MicroBlaze processor should be selected by default because there is not a PowerPC in this specific FPGA.

17. Click Next on the dialog box. A Base System Builder - Configure Processor dialog box is displayed.

18. Select XMD with S/W debug stub and 64K of Local Data and Instruction Memory in the Processor Configuration. 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. All of the processor memory uses the internal block RAMs of the FPGA.

19. Click Next on the dialog box. A Base System Builder - Configure IO Interfaces dialog box is displayed. The checkbox indicates that the project will be built with the RS-232 interface as specified. The default settings of the Uart 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.

20. Click Next on the dialog box. A 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.

21. Click Next on the dialog box. A Base System Builder - Software Configuration dialog box is displayed. Un-check the Generate Sample Application and Linker Script checkbox.

22. Click Next on the dialog box. A 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. Base System Builder is a wizard that helps ease the effort to generate a system by building the data files for XPS.

23. A Base System Builder - Finished! 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.

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).

Building The Hardware

24. Select the Tools menu and the Generate Bitstream submenu in XPS to start building the hardware system. This may take about 10 minutes as the system is compiled, 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, map,

place, and route the design. When this step is complete, a system.bit file is created in the /implementation subdirectory of the XPS project.

Defining The Software

25. Double click the microblaze_0 processor in the System tab of XPS. To use standard I/O (printf, etc.), the peripheral to be used for standard I/O must also be setup for the processor. In the Library/OS Parameters tab, select the STDIN and STDOUT peripherals to be RS232. Click OK.

26. Double click on Software Projects in the Applications tab of XPS. For the Project Name, type mb0_default. 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. Click Mark to Initialize BRAMs.

27. Double click on the project mb0_default. A dialog box will open to alow users to set compiler settings for the project. In the Environment tab, select XMDStub as the Mode.

28. Expand the mb0_default project in the Applications tab of XPS. Right click on Sources and select Add File. Browse to the /code subdirectory of the project and select the lab1.c source file. Hit the open button on the dialog box.

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

    While it was necessary to set driver levels in EDK 6.1i, it is not necessary to do so in EDK 6.2i. Driver levels and the device driver architecture are documented in the EDK documentation and are a subject that will be covered in future labs.

Compiling the Drivers and Program

30. Select the Tools menu and the Generate Libraries and BSPs submenu. This will cause the drivers and startup code to be compiled into a library that will be used to link with the program.

31. Select the Tools 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.

32. Select the Tools menu and the Update Bitstream submenu. This will cause the xmdstub.elf file that was generated for the software to be inserted into

the hardware bitstream (in BRAM) such that both software and hardware may be downloaded to the FPGA.

Using More GNU Tools

33. Select the Tools menu and the Xygwin shell submenu. This will start a Bash shell under Xygwin. Change to the microblaze_0/code directory, which should contain the executable.elf file.

34. In the Bash shell window, type "mb-objdump -d executable.elf ⟩ disassembly.out" to disassemble the 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 at 0x400 instead of at 0x0?

    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?

35. Within XPS, where would you specify the program start address for the software application? (hint: look in the Applications tab) This is useful when you have memory in various places in your memory map. This is controlling a flag given to the linker/loader phase of the compiler.

Downloading the Bitstream to the FPGA

36. Go to the Courseware directory (folder) and open the XILINXPORT. The terminal settings have been preset to 9600, 8, N, 1. These serial port settings match the settings from the I/O dialog box during the Base System Builder Wizard system creation. Double check the serial cable connection between the board and the PC.

37. Ensure that power is on to the board and the parallel 4 cable is connected to the PC. The Status light on the parallel 4 pod should be green. Select the Tools menu in XPS and Download submenu. 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 will take a few minutes.

The FPGA Done LED also turns on when programming is completed. If you see an error/warning in the XPS ouput window that the done pine could not be driven, try downloading again.

Getting Ready to Debug

38. Select the Tools menu and the XMD submenu. This will start Xilinx Microprocessor Debug in a new bash window. This program communicates with the board over the parallel 4 cable.

39. In the XMD window, type "help" to display the commands which can be used within this window. Type "mbconnect stub" to connect to the Microblaze Debug Module (MDM) and the MicroBlaze processor that is executing in the FPGA. The results should indicate that it did connect successfully and that a GDB server was started. GDB (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.

    If you get a message stating that your processor is in reset, check the position of the User Input switches. One of these switches is tied to the processor reset. Which one? What position must it be in for you to be able to use the processor? (Hint: The UCF file contains pin mappings and the MultiMedia Board schematics will tell you the pin names for the switches)

40. In the XMD window, read the contents of memory location 0 based on the commands displayed in the XMD 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 12, 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 J 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 (something weird is happening, go to the simplest interface), then you can always resort to the XMD interface.

41. Recall the start address for the executable.elf file that you found previously. In the XMD window, disassemble at memory location 0x400.

What is the assembly language instruction contained at location 0x400? Is this what you expected? What is going on?

Debugging Software

42. Due to a bug in EDK 6.2 that causes MicroBlaze to hang, the software executable must be downloaded using XMD and not GDB. In XMD run

    dow mb0_default/executable.elf

    to download the executable.

43. 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. Click OK. The processor will be stopped at a breakpoint at the beginning of the program.

    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 0x400 and above. Here it is probably easier to use the disassemble command rather than the memory read command. What do you see now?

44. 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?

45. 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.

46. In XPS, select the Applications tab and double click Compiler Options. On the 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 no optimization so that debugging can be done.

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

48. 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.

49. In the xygwin 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.

50. Practice using GDB, keeping in mind that you must avoid having GDB download the program (it does this when you use Run → Run or the small running man icon to run the program). 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.

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

More Trickiness

52. 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.

53. 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 and, ultimately use them to run your compiles 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. Do not remove the flying leads for the JTAG connection from the board.
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 Gizmo 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 2: Adding Drivers and IP - GPIO.