ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Peng Li
TA: Andrew Targhetta
(Lab exercise created by A Targhetta and P Gratz)

Laboratory Exercise #4

Linux boot-up on XUP board via Compact Flash

Objective

The purpose of lab this week is to get Linux up and running on the XUP boards. There are many advantages to running an Operating System (OS) in an embedded processor environment, and Linux provides a nice open-source OS platform for us to build upon. This week, you will use XPS to build a MicroBlaze based soft-IP microprocessor system suitable for running Linux, and you will also use open source tools to compile the Linux kernel based on the specification of your custom microprocessor system. You will then combine the bit stream for programming the FPGA hardware with the compiled object code containing the Linux OS into a SystemAce file, which we will use to boot our Linux system from Compact Flash.

System Overview

The microprocessor system you will build in this lab is depicted in Figure 1. As in last lab, this system has a MicroBlaze processor, a debug peripheral, a UART, and a custom multiplication peripheral. Unlike last lab, however, this system has an XPS Timer, a System Advanced Configuration Environment (SysACE) peripheral, an interrupt controller, and a DDR2 SDRAM (Double Data Rate v2 Synchronous Dynamic Random Access Memory) controller. Additionally, the MiroBlaze itself has a Memory Management Unit (MMU) and instruction and data cache. These additional peripherals along with the MMU within the MicroBlaze are required to run Linux. The timer is necessary for certain OS system calls. The SysACE controller provides Compact Flash read/write access. The DDR2 SDRAM controller provides the system with 256MB of

RAM where the Linux kernel can reside. The interrupt controller enables interrupt handling necessary for interaction with I/O. The MMU within the MicroBlaze enables virtual memory, which is required to run a mainstream Linux kernel. The MicroBlaze cache is added to improve performance, as DDR SDRAM accesses have high latencies. Please note that the system will run without the cache but will run considerably slower.

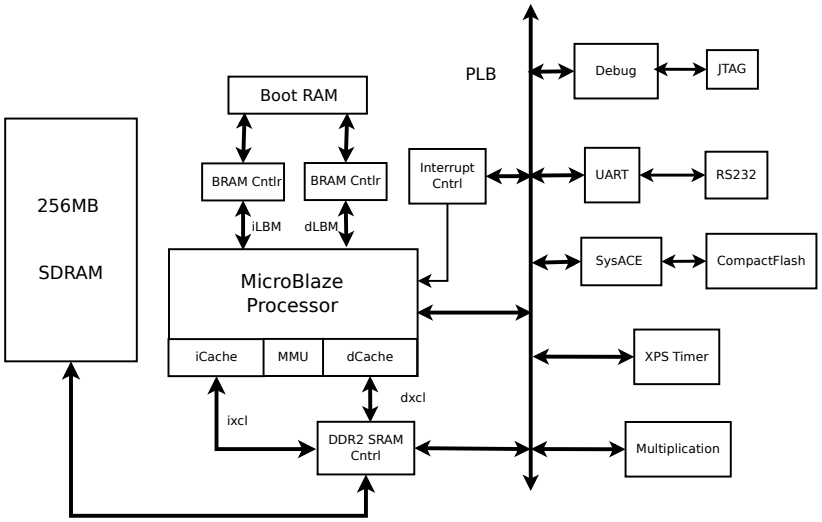

Figure 1: MicroBlaze System Diagram

Procedure

1. To begin, we must create a base MicroBlaze system which includes all the peripherals shown in Figure 1 except the multiplication peripheral.

   (a) Open XPS and start the Base System Builder as outlined in Lab 2. Please note that we will be using the BSB to add more peripherals to our base system than we did last week.

(b) In the 'Configure MicroBlaze Processor' window, set the 'Processor-Bus clock frequency' to 75 MHz.

(c) In the same window, select 'Enable' under 'Cache setup'. See Figure 2. This will add cache to our MicroBlaze system.

Caches are small, high-speed memories within the microprocessor. The running Linux kernel along with user applications are stored in the 256MB RAM module; however access to DDR2 SDRAM is slow relative to the MicroBlaze. To alleviate the speed mismatch, portions of program code and data are "cached" inside these small memories.

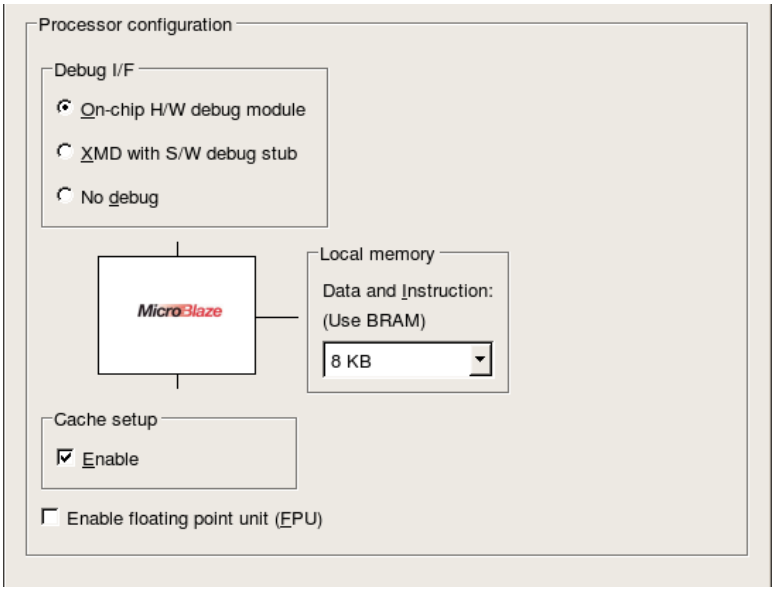

Figure 2: Enable cache

(d) On the first page of the 'Configure IO Interfaces' portion of the BSB, select the 'RS232_UART_1' peripheral. Like last lab, select the 'XPS UARTLITE' variant. Unlike last lab, set the baud rate to 115200 and ensure the 'Use interrupt' option is selected (Figure 3).

We must increase the data rate of the RS232 port; otherwise, the XUP Linux system will appear sluggish at terminal end. By selecting the 'Use interrupt' option on any of the peripherals, we cause XPS to include an interrupt controller in our MicroBlaze system. Interrupts allow the microprocessor to respond quickly to external events. The interrupt controller allows the software to control exactly which events our software responds to and at what priority.

(e) In the remaining IO interface pages, unselect all peripherals except the 'DDR2_SDRAM' and

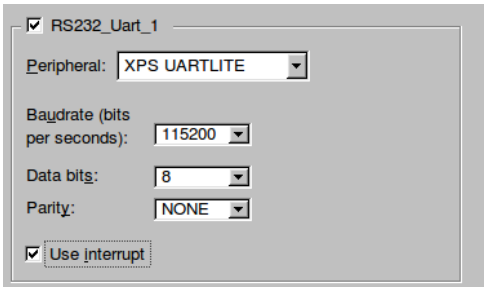

Figure 3: Configure RS232 UART

'SysACE_CompactFlash'. For the Compact Flash peripheral, ensure the 'Use interrupt' option is selected (Figure 4).

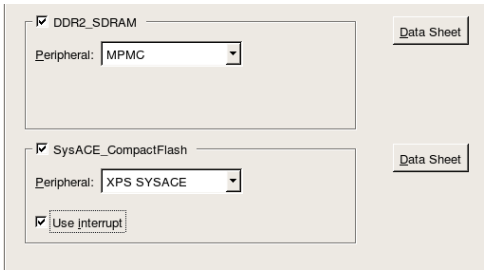

Figure 4: DDR2_SRDRAM and SysACE_CompactFlash peripherals

(f) In the 'Cache Setup' window, specify the 'ICache' and 'DCache' options for DDR2_SDRAM. See Figure 5 As with the local processor memory, there are separate caches for instructions and data (i.e. Harvard Architecture). For performance reasons, we want to cache both data and instruction accesses for programs residing within the DDR2 SDRAM. Notice that the local memory, which is used for the boot loader, is not listed as a cache-able memory. This is because both local memory and cache are constructed from BRAMs. Therefore, they operate at the same speed.

(g) Finish the BSB setup window as outlined in Lab 2.

(h) Click on the 'IP Catalog' tab in the XPS window and expand the 'DMA and Timer' category. Right click on the 'XPS Timer/Counter' peripheral to add a timer to your microprocessor system as shown in Figure 6.

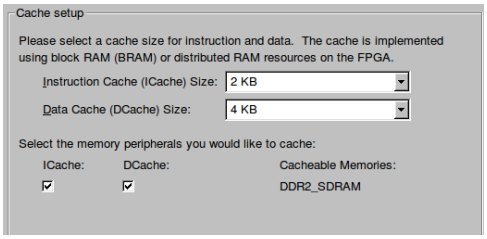

Figure 5: Cache Setup

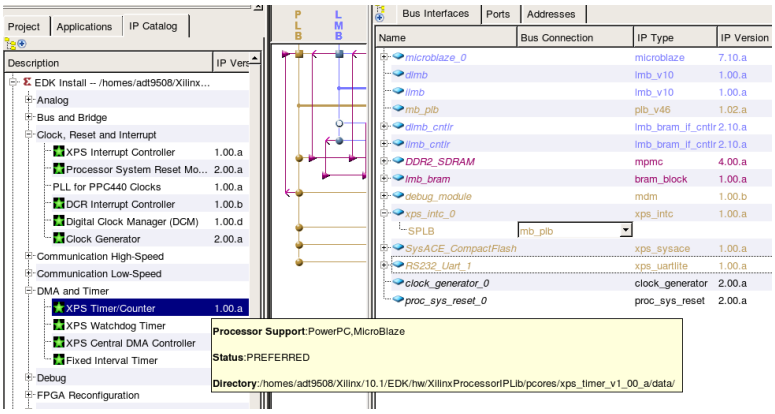

Figure 6: XPS Timer/Counter

(i) The timer module will now show up in the System Assembly view. Click on the 'Bus Interfaces' tab and connect the XPS timer module to the PLB as done in previous labs.

(j) Next, click on the 'Ports' tab and expand the 'xps_intc_0' module. This is the interrupt controller discussed previously. We must connect the timer, uart, and sysAce interrupt signals to the 'Intr' port in order to enable our peripheral interrupts. Double click on the 'Net' field for the 'Intr' port of the interrupt controller module. This will bring up the window depicted in Figure 7. Add the 'xps_timer_0_Interrupt', the 'SysACE_CompactFlash_SysACE_IRQ', and the 'RS232_Uart_1_Interrupt' signals to the list of connected interrupts and hit 'OK'.

(k) Verify that all three interrupt signals have been connected to the interrupt controller by opening the "system.mhs" file and locating the 'xps_intc' entry. Do this now and examine the file's contents.

(l) In the 'Addresses' window, set the address range for the timer to 64K and regenerate the peripheral addresses.

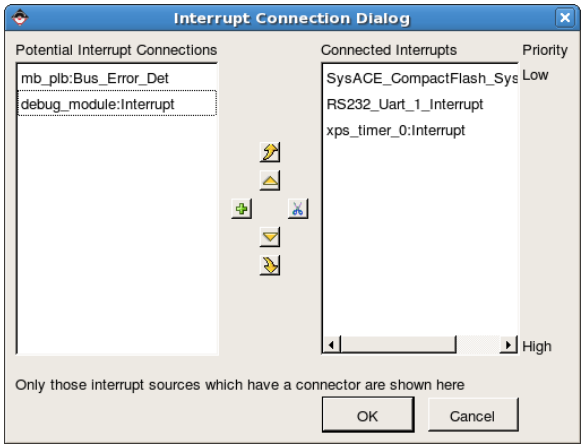

Figure 7: Connect Timer Interrupt

2. At this point, we have a base system that is almost ready to run Linux. We must now enable virtual memory within the MicroBlaze processor to fully support the kernel. Virtual memory allows programs to run independent of one another such that each is operating in its own 'virtual' RAM. As a result, each process is protected from other processes in that one process cannot overwrite another process' memory space. Additionally, if secondary memory is available, processes, which require more main memory than the system can support, are still able to run as pages can be swapped between RAM and disk. For many programmers, the most important artifact of virtual memory is that we can write our C programs as though no other programs are intended to run on our target machine.

   (a) In the system assembly view under the 'Bus Interfaces' tab, right click on the 'microblaze_0' instance and select 'Configure IP'. In the configuration window, click on the 'MMU' tab. Set the MMU settings to the following:

   ```
   Memory Management: VIRTUAL
   Data Shadow Translation Look-Aside Buffer Size: 4
   Instruction Shadow Translation Look-Aside Buffer Size: 2
   Enable Access to Memory Management Special Registers: FULL
   Number of Memory Protection Zones: 2
   ```

   The above settings configure the MMU to meet the minimum requirements for the Linux kernel.

   (b) While in the MicroBlaze configuration window, browse through the various configuration tabs and see if you can recognize some familiar terms. Then, select the 'Instructions' tab and enable the barrel shifter. Although not completely necessary, a barrel shifter will speed up many of the bit level operations found within commonly used subroutines for floating point emulation and cryptography.

(c) Hit 'OK' to complete the configuration of the MicroBlaze processor.

3. We are almost finished building the system depicted in Figure 1. To complete it, we must import the multiplication peripheral that we created last lab. We will be using this same system for the next lab. Thus, we add it now. Since the system we will synthesize has increased greatly in size, the goal is to reduce the number of system builds.

   (a) Copy the 'pcores' directory from lab 3 into your lab 4 directory. Try getting used to using the terminal. Run the following command in the terminal, one directory up from you lab 4 directory:
   >cp -r lab3/pcores/ lab4/pcores/

   Note that the '-r' stands for recursive copy and must be used when copying a directory.

   (b) Select **Hardware→Create or Import Peripheral** from the main menu of the XPS window.

   (c) Import the 'multiply' peripheral exactly as you did last lab.

   (d) Once the import process is complete, add the user peripheral, 'multiply', to your current microprocessor system. Do not forget to connect the multiplication peripheral to the bus in the 'Bus Interfaces' window and adjust the system addresses appropriately. Please consult lab 3 if this process is unclear.

   (e) While in the 'Addresses' window, ensure the local memory is set to 8 KB. If you recall, the previous labs have asked you to expand the local memory to 128 KB. We do not need the extra memory in this lab as we have 256 MB of DDR2 SDRAM, and in fact, our MicroBlaze may be unable to run at 75 MHz if we increase the local memory. This is due to the fact that we are now including cache made from BRAMs, and more BRAM utilization within the MicroBlaze means that we must start using BRAMs further away from the core logic of the MicroBlaze.

4. Now we have completed the hardware description of our microprocessor system. It is time to begin the steps necessary for compiling the software that will run on our system. In order to successfully boot the Linux kernel, we must create a device tree using the Xilinx device tree generator. The device tree generator allows the Linux kernel to configure itself upon boot-up based on the hardware in your system.

   (a) Copy the device tree generator directory, '/homes/faculty/shared/ECEN449/bsp', into your lab 4 directory.

   (b) Restart XPS so that the device tree generator will load into XPS.

   (c) Select **Software→Software Platform Settings** from the XPS main menu.

   (d) Within the 'Software Platform Settings' window, change the 'OS' field to 'device-tree'. See Figure 8.

(e) Now select 'OS and Libraries' on the left side of the 'Software Platform Settings' window. Within the 'Current Value' field under 'device-tree'→'console device' type in 'RS232_Uart_1'. Also under 'device-tree'→'bootargs', change 'console=ttyS0' to 'console=ttyUL0'. This tells Linux to use the ultralite RS232 peripheral for the console. Click 'OK' to save your settings and exit out of the current window.

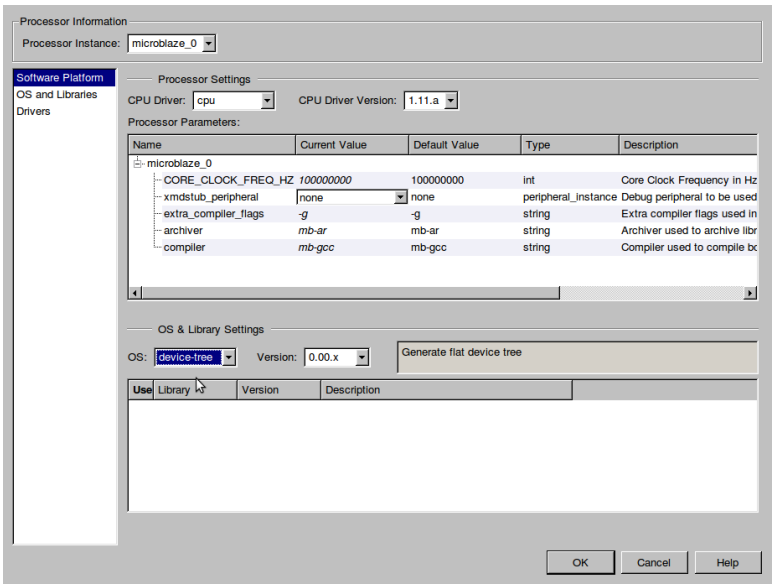

Figure 8: Software Platform Settings

(f) Before starting the build process, click on the 'Applications' tab and ensure only the 'microblaze_0_bootloop' is marked to initialize in the BRAMs. Pay close attention to what is going on here as there will be a question about this at the end of lab.

(g) Now generate the necessary software libraries by selecting **Software→Generate Libraries and BSPs** from the main menu in XPS.

(h) Finally, select **Device Configuration→Update Bitstream** from the XPS main menu. This will automatically synthesize and implement the entire hardware system, compile the boot-up loop, initialize the BRAMs, and generate the device tree.

(i) Once complete, the device tree file, 'xilinx.dts', will be placed in the 'microblaze_0/libsrc/device-tree_v0_00_x/' directory. Navigate to this directory and ensure the device tree file is present.

5. It is now time to compile the Linux kernel!!!

(a) First, we must setup our environment such that the compile process utilizes the microblaze GNU cross-compiler and associated libraries. These tools are stored in the '/homes/faculty/shared/ECEN449' directory so we must point to them. Open a terminal window and type the following command:

>source /homes/faculty/shared/ECEN449/compile_settings.csh

(b) Open the "compile_settings.csh" file and note its contents. The term "cross-compile" refers to the process of compiling source code for one architecture on another. For example, we are using an x86 machine to compile source code for a MicroBlaze.

(c) Untar the '/homes/faculty/shared/ECEN449/linux-2.6.35.7.tar.gz' file by executing the following command from within your lab 4 directory:

>tar -xvjf /homes/faculty/shared/ECEN449/linux-2.6.35.7.tar.bz2

The uncompressed directory, 'linux-2.6.35.7', contains the Linux kernel source files.

(d) During the compile process, the tools will look for the device-tree created in step 4. In the 'linux-2.6.35.7/arch/microblaze/boot/dts/' directory replace the 'system.dts' file with your 'xilinx.dts' file.

(e) The Linux system you will build needs a root file system to operate. For this lab, we will use a prebuilt RAM file system. Copy the '/homes/faculty/shared/ECEN449/initramfs_minimal.cpio.gz' file into your 'linux-2.6.35.7' directory.

(f) Before configuring the kernel for compilation, we want to start with a base configuration so we only have to make minor modifications. Enter the following commands in the terminal window from within the 'linux-2.6.35.7' directory:

>make ARCH=microblaze xilinx_mmu_defconfig
>make ARCH=microblaze menuconfig

(g) A window similar to that depicted in Figure 9 will appear. Navigate to 'Platform options' and hit enter. Scroll down to the 'Physical address where Linux is' field and change the default value to the start address of the 256 MB DDR2 SDRAM. Note that this value can be found in the 'Addresses' window within XPS. After boot-up, the system will operate using virtual memory. However, initially the boot code needs to know where to place the uncompressed Linux kernel, and that is exactly what we are specifying here.

(h) During the compile process, certain compiler flags must be set based on the available MicroBlaze instructions. If you recall, we added a barrel shift to our microprocessor. Thus, you must change the 'USE_BARREL' field to '1'. Also, change the 'Core version number' to '7.10.d' to reflect the version of MicroBlaze we are using.

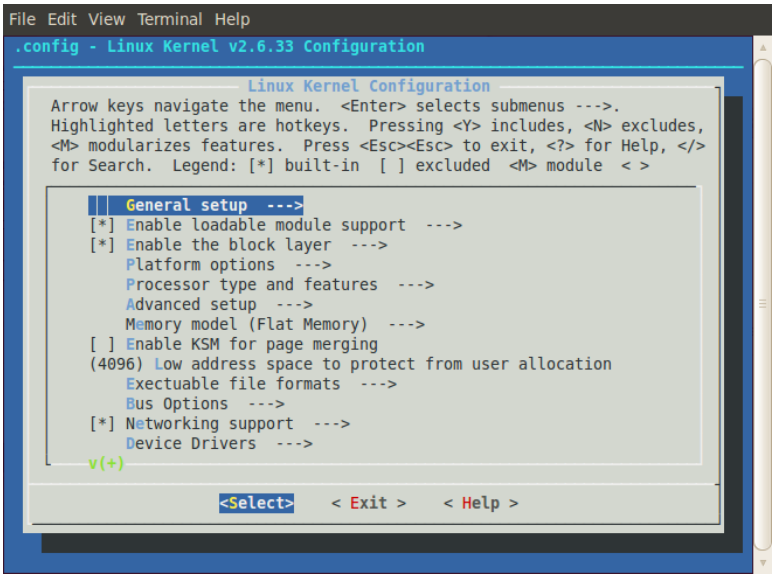

Figure 9: The Linux Kernel Configuration Tool "menuconfig"

(i) In our hardware system, we did not include an Ethernet MAC therefore networking support is unnecessary and will be removed from our custom kernel. To do this, exit our of the 'Platform options' part of the kernel configuration window and scroll down to 'Networking support' from the main menu. Hit the space bar to unselect this option.

(j) The kernel configuration is almost complete. Have the TA check out the configuration before exiting the tool.

(k) The next step is to actually compile the kernel. Do this by entering the following command in the current terminal:

>make ARCH=microblaze simpleImage.xilinx

(l) Once complete, a 'simpleImage.xilinx' file will appear in the 'linux-2.6.35.7/arch/microblaze/boot/' directory. This file is an Executable and Linkable Format (elf) file, which contains the object code for uncompressing the kernel along with the compressed Linux kernel object code. The RAM file system is also included in this elf file. Ensure this file exists after compiling the kernel and copy it to the lab 4 directory.

6. Now that we have the software and hardware compiled, we must combine them into a single file that we can program the FPGA with. Xilinx provides an easy way to boot a system via the Compact Flash so the next few steps will outline this process.

(a) Copy the '/homes/faculty/shared/ECEN449/xupGenace.opt' file into your lab 4 directory. We will use the Xilinx Microprocessor Debugger (XMD) along with a Tcl script and this .opt file to generate a SysAce file. The Tcl script provides XMD with the necessary commands to create the SysAce file, while the .opt file provides XMD with the necessary device information.

(b) In the XPS window, select **Project→Launch EDK Shell** from the main menu.

(c) Run the following command from the EDK shell:

>xmd -tcl genace.tcl -opt xupGenace.opt

This will create a file called 'linux_system.ace' in your project directory. Ensure this file exists once the Tcl script completes.

(d) Copy the 'linux_system.ace' file to the Compact Flash (CF) card provided in lab. Be sure to properly unmount the CF card from the CentOS machine once the copy is complete.

(e) Open kermit as outlined in lab 2 (except with a baud rate of 115200) and CAREFULLY insert the Compact Flash card into the CF slot on the XUP board. If the process was successful, you should see Linux booting on the XUP board via the kermit console. Demonstrate this to the TA.

Deliverables

1. [6 points.] Demo Linux booting on the XUP board to the TA.

Submit a lab report with the following items:

2. [8 points.] Correct format including an Introduction, Procedure, Results, and Conclusion. Be sure to summarize the process required to build the hardware and compile the Linux kernel.

   Warning: Missing information will result in missing points.

3. [2 points.] The output of the terminal (kermit) showing the Linux boot.

4. [4 points.] Answers to the following questions:

   (a) Compared to lab 3, the lab 4 microprocessor system shown in Figure 1 has 256 MB of SDRAM. However, our system still includes a small amount of local memory. What is the function of the local memory? Does this "local memory" exist on a standard motherboard? If so where?

   (b) After your Linux system boots, navigate through the various directories. Determine which of these directories are writable. (Note that the man page for 'ls' may be helpful).

   Test the permissions by typing 'touch <filename>' in each of the directories. If the file, <filename>, is created, that directory is writable. Suppose you are able to create a file in one of these directories. What happens to this file when you restart the XUP board? Why?

(c) If you add a peripheral to you system after compiling the kernel, which of the above steps would you have to repeat? Why?

(d) Suppose during the kernel configuration process you left the field 'USE_BARREL' set to 0. Would the system still function? Why or why not?