XAPP1078 (v1.0) February 14, 2013

Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors

Author: John McDougall

Summary

The Zynq™-7000 All Programmable SoC contains two ARM® Cortex™-A9 processors that can be configured to concurrently run independent software stacks or executables. This application note describes a method of starting up both processors, each running its own operating system and application, and allowing each processor to communicate with the other through shared memory.

Included Systems

The design is created and built using the Xilinx Platform Studio (XPS) 14.3. The design also includes software built using the Xilinx Software Development Kit (SDK). A complete set of project files is provided with this application note to allow the designer to examine and rebuild the design or use the files as a template for starting a new design.

Pre-built and pre-implemented files targeting the Zynq-7000 ZC702 demonstration platform are also provided in case the designer wants to skip the steps of reproducing hardware, software, or boot file targets.

Introduction

The Zynq-7000 AP SoC provides two Cortex-A9 processors that share common memory and peripherals. Asymmetric multiprocessing (AMP) is a mechanism that allows both processors to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources. The reference design includes the hardware and software necessary to build a reference design that runs both Cortex-A9 processors in an AMP configuration. CPU0 runs Linux and CPU1 runs a bare-metal application. Care has been taken to prevent the CPUs from conflicting on shared hardware resources. This document also describes how to create a bootable solution and how to debug both CPUs.

Design Overview

In this reference design, each of the two Cortex-A9 processors is configured to run its own software. CPU0 is configured to run Linux and CPU1 is configured to run a bare-metal application.

In this AMP example, the Linux operating system running on CPU0 is the master of the system and is responsible for:

- System initialization
- Controlling CPU1’s startup
- Communicating with CPU1
- Interacting with the user

The bare-metal application running on CPU1 is responsible for:

- Managing a “heart beat” that can be monitored by Linux on CPU0
- Communicating with Linux on CPU0
- Servicing interrupts from a core in the programmable logic (PL)
- Communicating interrupt events to Linux running on CPU0

© Copyright 2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. ARM and Cortex are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.

The Zynq SoC processing system (PS) includes resources that are both private to each CPU and shared by both CPUs. In running the design in an AMP configuration, care must be taken to prevent both CPUs from contending for these shared resources. Refer to *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 1] for further information on shared and private resources.

Examples of some of the private resources are:

- L1 cache
- Private peripheral interrupts (PPIs)
- Memory management unit (MMU)
- Private timers

Examples of some of the shared resources are:

- Interrupt control distributor (ICD)
- DDR memory
- On-chip memory (OCM)
- Global timer
- Snoop control unit (SCU) and L2 cache

In this example, CPU0 is treated as the master and controls the shared resources. If CPU1 were to require control of a shared resource, it would have to communicate the request to CPU0 and let CPU0 control the resource. To keep the complexity of this reference design to a minimum, the bare-metal application running on CPU1 has been modified to limit access to the shared resources.

OCM is used by both processors to communicate to each other. When compared to DDR memory, OCM provides very high performance and low latency access from both processors. Deterministic access is further assured by disabling cache access to the OCM from both processors.

Actions taken by this design to prevent problems with the shared resources include:

1. DDR memory: Linux has only been made aware of memory at `0x00000000` to `0x2FFFFFFF`. CPU1 uses memory from `0x30000000` to `0x3FFFFFFF` for its bare-metal application.
2. L2 Cache: CPU1 does not use L2 cache.
3. ICD: Interrupts from the core in PL are routed to the PPI controller for CPU1. By using the PPI, CPU1 has the freedom to service interrupts without requiring access to the ICD.
4. Timer: CPU1 uses the private timer for the heart beat.
5. OCM: Accesses to OCM are handled very carefully by each CPU to prevent contention. In the case of the heart beat, only CPU1 writes the location and CPU0 reads the location. When data is sent from CPU1 to CPU0, only CPU1 writes the data value, then CPU1 sends a flag by setting a different address location in OCM. CPU0 in turn detects the flag, reads the data, and then clears the flag. Only CPU1 can set the flag, and only CPU0 can clear the flag.

For demonstration purposes only, a custom embedded core included with this example design is used to provide a simple interrupt source. An output from the ChipScope™ analyzer Virtual Input/Output (VIO) core is connected to this core, enabling the user to generate interrupts towards the PS at their leisure. Using the Chipscope VIO core provides more control over when an interrupt occurs and therefore makes it easier to measure the latency of interrupts. In a real-world design, however, this core would not exist and instead, the interrupt would be sourced by a truly functional piece of logic in the PL such as a direct memory access (DMA) engine.

Hardware

The PL contains a custom, embedded core connected to a synchronous output of a ChipScope analyzer VIO core (Figure 1). The VIO core provides a mechanism for a user to interact with hardware from ChipScope analyzer.

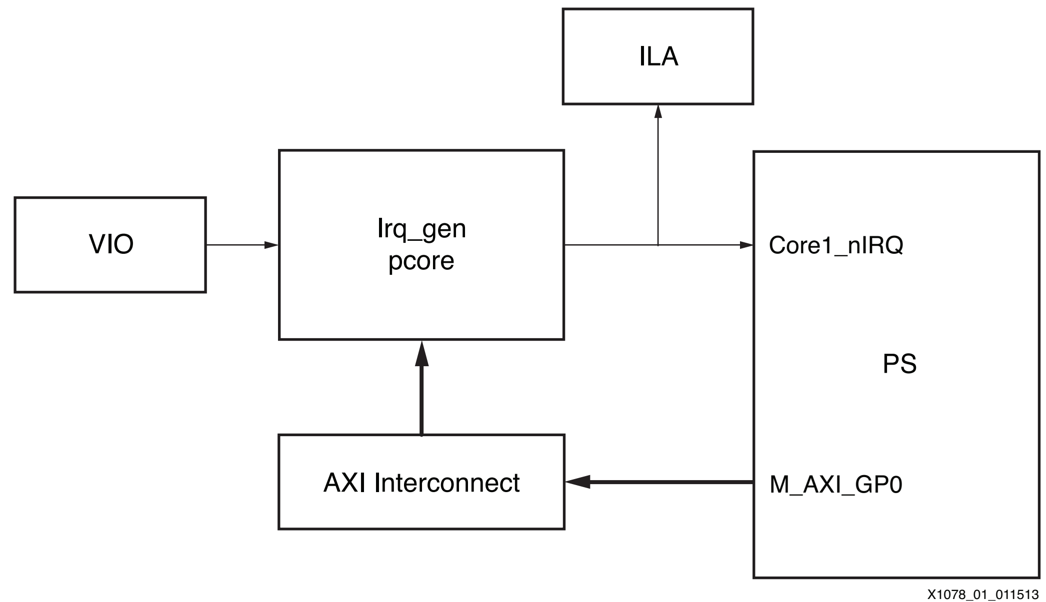

Figure 1: **PL Block Diagram**

In this design, when the VIO generates a pulse, the custom core forwards an interrupt to the PS Core1_nIRQ pin. The core is also connected to the PS master general purpose port (M_AXI_GP0) through an AXI Interconnect, allowing both CPU0 and CPU1 access to the control register within the core. CPU1 accesses the control register to clear the interrupt request (IRQ) during the interrupt service routine. CPU0 can optionally use the control register to create an interrupt towards CPU1. The Core1_nIRQ pin connects directly to CPU1's PPI block so there is no need to modify the configuration of the shared ICD. A ChipScope analyzer AXI monitor core is also included and allows the user to measure the latency of the IRQ being serviced.

Address Map

In the PL, there is a single irq_gen embedded core that contains a single control register. The register is located at BASE + 0 (`0x78600000`). Table 1 contains a description of the IRQ_GEN control register.

Table 1: **IRQ_GEN Control Register**

Bit	Access	Description
[31:1]	R/W	Unused. Value written can be read.
[0]	R/W	IRQ Asserted • 0: IRQ is not asserted towards the PS. • 1: IRQ is asserted towards the PS. If the VIO_IRQ_TICK pin is asserted (by the VIO), this bit is set. Also, the CPU can set this bit. Only the CPU can write this bit to clear it.

Software

The software can be broken down into three sections:

- First stage boot loader (FSBL)
- Linux operating system and applications for CPU0
- Bare-metal operating system and application for CPU1

FSBL

The FSBL always runs on CPU0. It is the first software application that is run after power-on reset of the PS. The FSBL is responsible for programming the PL and both application executable and linkable format (ELF) files to DDR memory. After loading the applications to DDR memory, the FSBL starts executing the first application that was loaded.

However, the version of FSBL included in the ISE® Design Suite 14.3 does not support multiple data or ELF files. The FSBL first looks for a bit file. If a bit file is found, the FSBL writes it to the PL. Next, whether or not a bit file is found, the FSBL loads one application ELF into memory and executes it. This operating sequence does not support such an AMP configuration, so the FSBL must be modified.

Within this AMP example's project files, the FSBL has been modified to continue searching for files and loading them into memory until it detects a file that has a load address of `0xFFFFFFF0`. Upon detection, the FSBL downloads this last file and jumps to the executable address of the first non-bit or non-boot file found (which is the application for CPU0). For details regarding how CPU1 starts up, refer to *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 1].

Linux

The easiest way to use Linux in an AMP configuration is to configure Linux as symmetric multiprocessing (SMP) but restrict the number of available CPUs to 1. Such an approach ensures that Linux configures the ICD and SCU correctly for a multiple CPU environment.

To create the Linux kernel, U-Boot, device tree, and the root file system ramdisk, refer to the wiki pages at http://wiki.xilinx.com. All generated files are available as part of the project files accompanying this application note.

To instruct Linux to use only one CPU for SMP, the bootargs in the device tree is modified to add **`maxcpus=1`**. By default, the Linux `.config` is already setup to use SMP on the ZC702 demonstration board.

The device tree is also modified to reduce the amount of memory available to Linux to provide untouched memory space for CPU1's application.

Linux Applications

Two Linux applications that run on CPU0 are provided to interact with CPU1 that is running the bare-metal application. The first application, rwmem, provides a simple memory read and write access from Linux to OCM. This rwmem application is used to peek and poke addresses in OCM. As specific address locations are changed, CPU1 detects the changes and interacts in a specific way. The second application, softUart, provides a UART-style communication between Linux running on CPU0 and bare-metal running on CPU1 through predefined memory locations in OCM.

After the PS powers up and the internal boot ROM completes execution, CPU1 will have been redirected to a small piece of code in OCM at `0xFFFFFE00`. This piece of code is a continuous loop that waits for an event, checks address location `0xFFFFFFF0` for a non-zero value and then continues the loop. If `0xFFFFFFF0` contains a non-zero value, CPU1 will jump to the fetched address.

CPU0 (running Linux) starts CPU1 (running bare-metal) by writing the value of 0x30000000 to address 0xFFFFFFF0 using the included rwmem application. Normally, CPU0 would need to run a set event (SEV) command to wake up CPU1. Because Linux, running on CPU0, is constantly servicing interrupts (another source of events), an SEV command is not necessary. When CPU1 wakes up, it reads the value 0x30000000 from address 0xFFFFFFF0 (written using the rwmem command) and then jumps to address 0x30000000. Note that the FSBL placed CPU1's ELF at 0x30000000.

The softUart application, which is also included in the design files, is run as a background task in Linux. When running, softUart continuously monitors shared OCM memory at locations 0xFFFF9000 (COMM_TX_FLAG_OFFSET) and 0xFFFF9004 (COMM_TX_DATA_OFFSET). Whenever a 1 is present at COMM_TX_FLAG_OFFSET, softUart reads the value found at COMM_TX_DATA_OFFSET and temporarily stores the value in a string array. When a value of 0x0A (\n) is received, the string array is displayed on STDOUT. Every time softUart reads a value from COMM_TX_DATA_OFFSET, it clears the COMM_TX_FLAG_OFFSET content. This clear signals to CPU1 that another character can be sent towards the softUart application running on Linux.

Bare-Metal Application Code

The reference design has CPU1 running bare-metal application code. Linux, running on CPU0, is responsible for initializing shared resources and starting up CPU1.

The bare-metal board support package (BSP) named standalone_v3_07_a that is part of the EDK 14.3 install includes support for the preprocessor define constant USE_AMP. This constant prevents the BSP from re-initializing the PS SCU that has previously been initialized by CPU0. One caveat of using the USE_AMP constant is that the MMU mapping is adjusted to create an alias of memory where the physical memory located at address 0x20000000 is virtually mapped to 0x00000000. This remapping is done in the BSP file boot.s. The re-mapping is not necessary for this design. A modified version of the BSP is included in the reference design to remove the re-mapping when USE_AMP is set.

Within this AMP reference design, no Zynq UARTs are used by the bare-metal application. Instead, the application running on CPU1 contains its own outbyte() function that is used to communicate via OCM to a software UART running in a Linux application on CPU0. By adding the outbyte() function, all stdout functionality of the standalone BSP is intact, allowing functions such as xil_printf() to be used.

To prevent shared resource conflicts, the bare-metal application running on CPU1 must be careful not to access resources such as the SCU. Linux disables cache access to the OCM. However, the default standalone BSP would attempt to enable cache for OCM and therefore conflict with Linux. When used in an AMP configuration, the function XIL_SetTlbAttributes() is used in the CPU1's main() application function to disable cache on OCM. The XIL_SetTlbAttributes() function has been modified in the included source code such that it only flushes L1 cache and leaves L2 cache untouched to prevent access to the SCU where L2 cache is controlled.

If the bare-metal code running on CPU1 requires control of L2 cache, a communications channel must be created allowing the bare-metal code to request Linux to make the necessary changes to the SCU. This action is beyond the scope or requirement for this example design so care has been taken to prevent SCU access directly from the bare-metal code.

CPU1 Application

CPU1's application is located in memory starting at address 0x30000000. The linker script is used to set the starting address.

CPU1's application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched so OCM exists at addresses 0x00000000–0x0002FFFF and addresses

`0xFFFF0000–0xFFFFFFFF`. Only the high 64 KB of OCM is used by the design so cache is disabled on addresses `0xFFFF0000–0xFFFFFFFF`.

2. Initializes the PPI interrupt controller and interrupt subsystem.
3. Increments an OCM location (COMM_VAL). This OCM location is referred to as the Heartbeat.
4. Sleeps for one second.
5. CPU1's main() function repeats step 3 and step 4 continuously.
6. As interrupts are detected, an interrupt service routine in the background clears the interrupt status of the embedded core and prints a string. The output from the print statement is redirected to use the OCM COMM_TX_FLAG_OFFSET and COMM_TX_DATA_OFFSET locations. In turn, Linux consumes the OCM data and prints the string to the Linux console.

Reference Design

The reference design files can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=202155

The reference design matrix is shown in Table 2.

Table 2: **Reference Design Matrix**

Parameter	Description
General	
Developer name	John McDougall
Target devices (stepping level, ES, production, speed grades)	XC7Z020-CLG484-1
Source code provided	Yes
Source code format	VHDL and Verilog
Design uses code/IP from existing Xilinx application note/reference designs, CORE Generator software, or third party	No
Simulation	
Functional simulation performed	No
Timing simulation performed	No
Test bench used for functional and timing simulations	No
Test bench format	N/A
Simulator software/version used	N/A
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/version used	XST 14.3
Implementation software tools/versions used	EDK 14.3
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	ZC702 board

The reference design contains these files:

- XPS project
- SDK source files for Linux and CPU1 applications
- Generated files including:
    - Bit file
    - All files for the SD card
    - Application ELF files for Linux and CPU1
- `BOOT.BIN` build scripts
- Modified bare-metal BSP
- Modified sw_apps FSBL
- Modified `devicetree.dts` and `devicetree.dtb`

Table 3 and Table 4 show the device utilization details.

Table 3: **Device Utilization (1)**

<table>
<tr><th>Parameters</th><th colspan="2">Specification/Details</th></tr>
<tr><td rowspan="4">Device utilization without testbench</td><td>Slices</td><td>246</td></tr>
<tr><td>GCLK buffers</td><td>2</td></tr>
<tr><td>PS7</td><td>1</td></tr>
<tr><td>RAMB36</td><td>1</td></tr>
<tr><td colspan="2">HDL language support</td><td>Verilog/VHDL</td></tr>
</table>

Table 4: **Device Utilization (2)**

Device	Speed Grade	Package	Pre-Map (Synthesis Constraint)	Post-Route	Slices
XC7Z020	-1	CLG484	240 MHz	160 MHz	246 (1%)

Implementation Details

This section discusses the implementation of the reference design.

The design files should be extracted to a directory called `design`. After the files have been extracted, a new directory called `design\work` should be created. Files should be copied as shown:

`design\src\bootgen` to `design\work\bootgen`

`design\src\edk_system` to `design\work\edk_system`

All generated files have been included and are located in the directory `design\generated_files`.

Generating Hardware

This section describes the creation of the hardware design. The user can skip this section and go to Generating Applications. The pre-compiled design is available at `design\generated_files\fpga\download.bit`.

To implement the embedded design and export it to SDK:

1. Start XPS and open the embedded project at `design\work\edk_system\system.xmp`.

2. Select **device_configuration > update_bitstream**. After it completes, the downloadable FPGA bit file is available at `design\work\edk_system\implementation\download.bit`.

   A precompiled version of the bit file is also available at `design\generated_files\fpga\download.bit`.

3. Export the hardware project to SDK by selecting **project > export_hardware_design_to_SDK**. Check the **Export & Launch SDK** button. At this point, XPS exports the embedded system configuration via a `system.xml` file that is used by SDK to understand what peripherals are present in the design and what the base addresses are. The file is automatically exported to `design\work\edk_system\SDK\SDK_Export\hw`. SDK opens a dialog box asking where the workspace is located. Browse to and select the directory `design\work\edk_system\SDK`, click **OK**. Before clicking **OK** a second time, add to the end of the selection `\Workspace` as shown in Figure 2. SDK automatically creates the `Workspace` subdirectory.

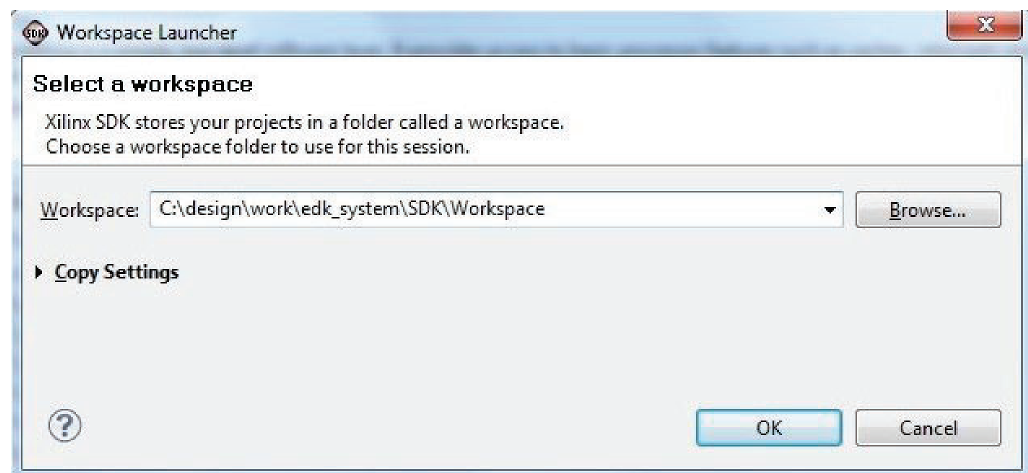

Figure 2: **Select Workspace Directory**

Generating Applications

Configuring SDK

The standalone BSP files (used by the bare-metal application) and modified FSBL files have been included in this design. To give SDK knowledge of these files, SDK needs to be configured to have knowledge of the new repository.

1. Start SDK and open the workspace at `design\work\edk_system\SDK\Workspace`. This step is not necessary if XPS was used with **Export & Launch SDK**.
2. Point SDK to the included repository that contains the modified standalone BSP and modified FSBL (Figure 3):
   - Select **Xilinx_tools > repositories**.
   - Select **New** for local repositories.
   - Browse to and select the directory `design\src\sdk_repo`.
   - Select **OK**.

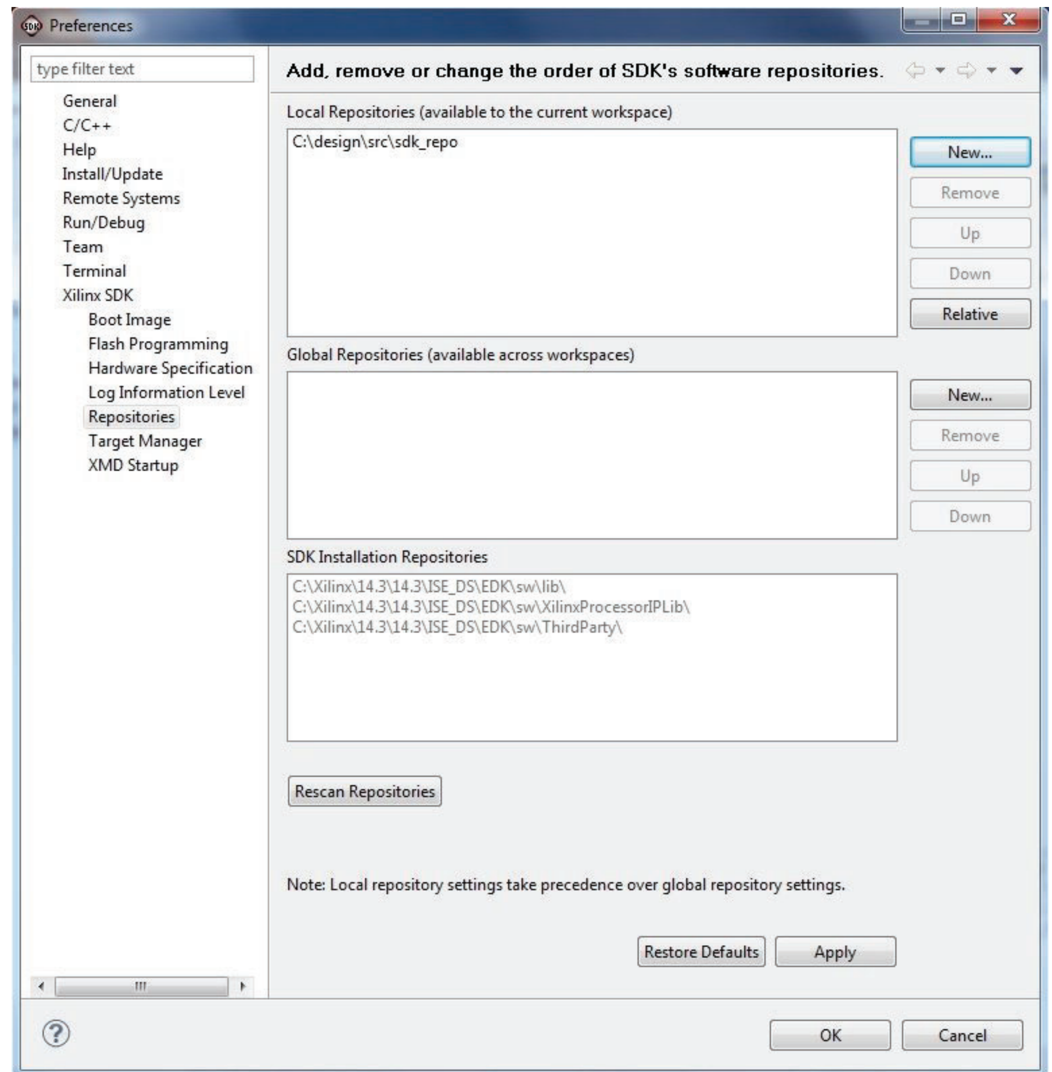

Figure 3: **Select Repository**

Creating Custom FSBL Application

1. Ensure that the Configure SDK section has already been run.
2. Create the FSBL using the new template from the repository (Figure 4):
   a. Select **File > New > Application_Project**.
   b. Set the project name to **amp_fsbl**.

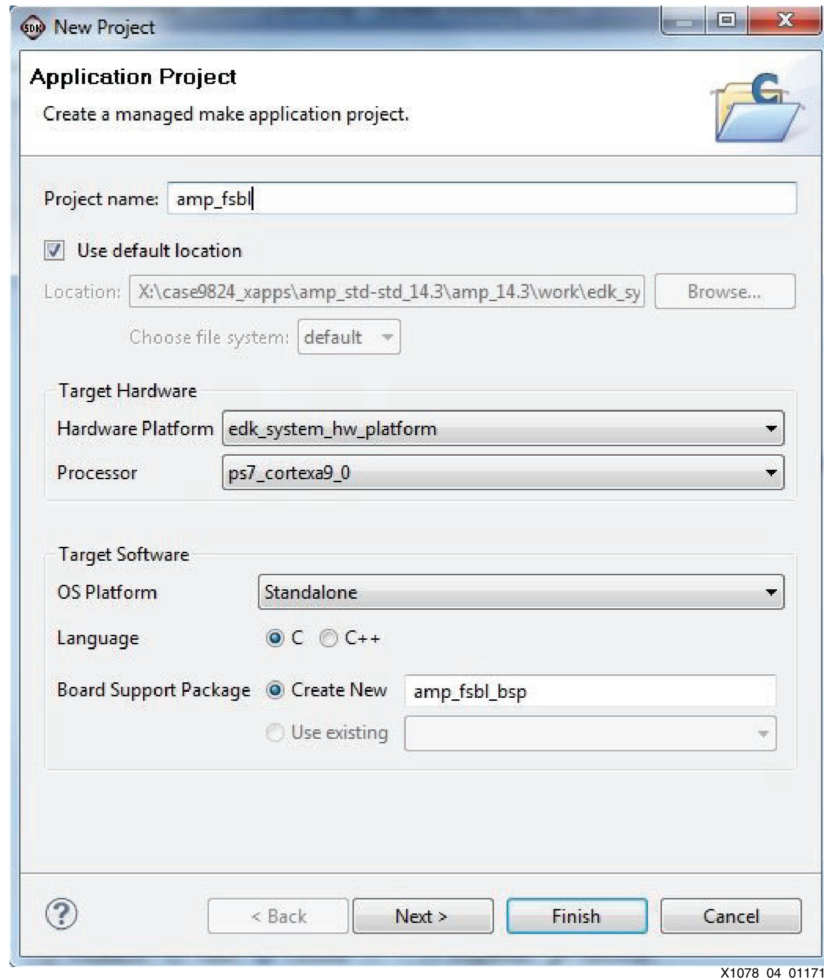

Figure 4: **Create FSBL**

   c. Click **Next**.

d. Select the available template Zynq FSBL for AMP. If this template is not available in the list, verify that the repository is set up correctly (Figure 5).

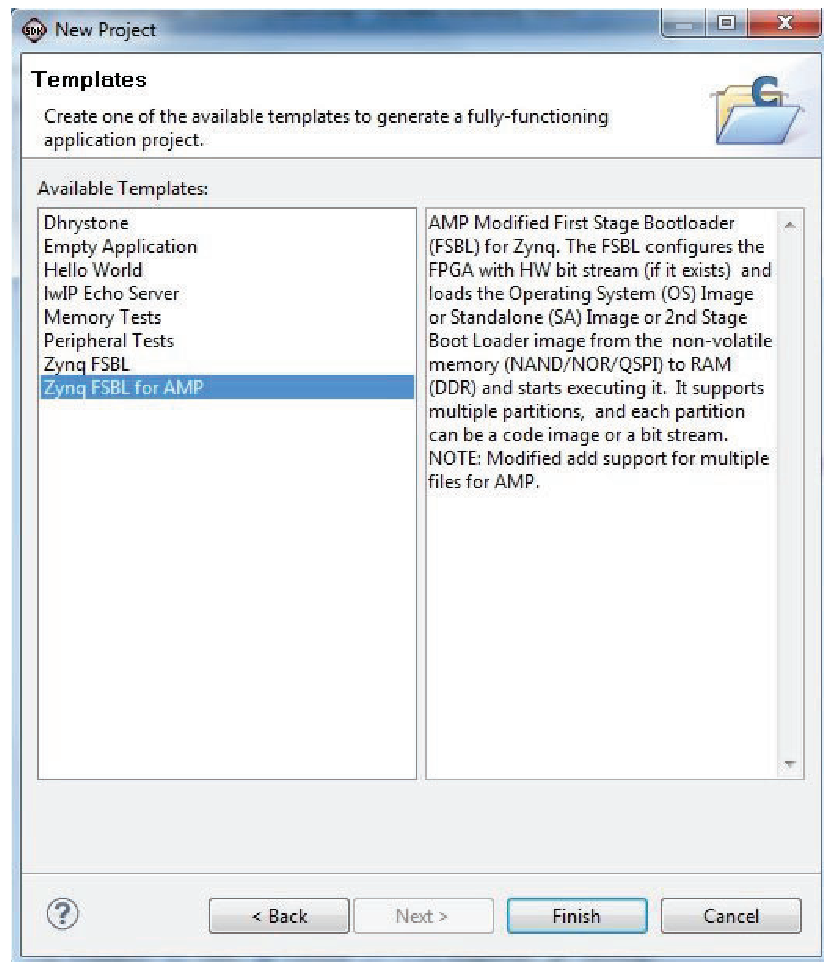

Figure 5: **Select Custom FSBL Template**

e. Click **Finish**.

3. After SDK completes compiling the new amp_fsbl_bsp BSP and the amp_fsbl application, the FSBL ELF is available at `design\work\edk_system\SDK\Workspace\amp_fsbl\Debug\amp_fsbl.elf`.

   A pre-compiled version is also available at: `design\generated_files\SDK_apps\amp_fsbl.elf`.

Creating Bare-Metal Application For CPU1

The instructions in this section create the application ELF that runs on CPU1 after the FSBL loads the applications to DDR memory. This step is slightly different than creating the application for CPU0 because CPU1 uses the customized BSP. This design prevents CPU1 from accessing shared resources such as the ICD or SCU.

The application has already been compiled and is available at `design\generated_files\SDK_apps\app_cpu1.elf`.

1. Within SDK, create the BSP using the customized standalone BSP from the repository that was included with the design (Figure 6).

   a. Select **File > New > board_support_package**.

   b. Change the project name to **app_cpu1_bsp**.

c. Change the CPU to **ps7_cortexa9_1**.

d. Change the board support package OS to **standalone_amp**.

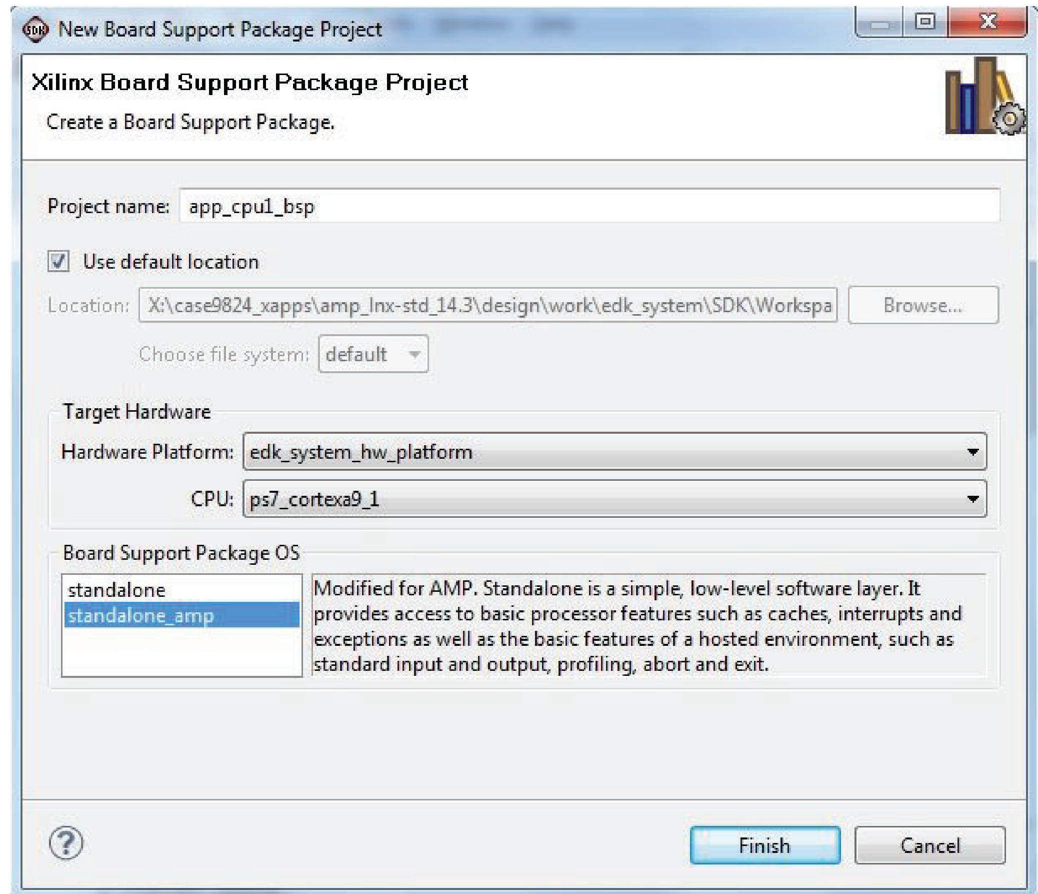

Figure 6: **CPU1 BSP**

e. Click **Finish**.

f. In the board support package settings, select **overview > standalone_amp** and change both stdin and stdout to **none** (Figure 7).

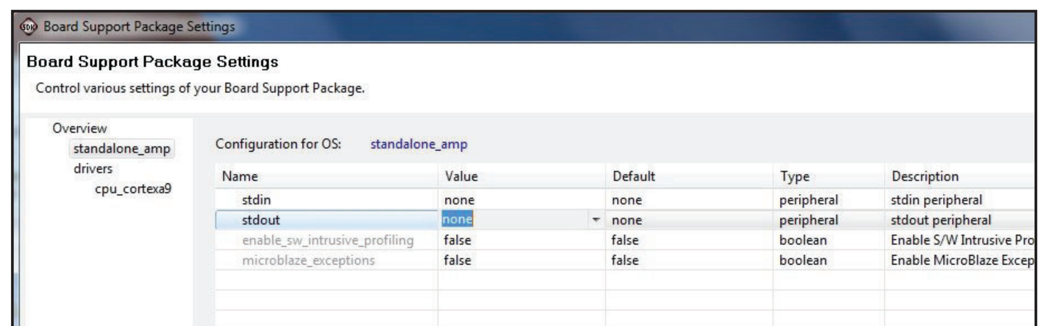

Figure 7: **CPU1 Set No STDIN or STDOUT**

g. Select **Overview > drivers > cpu_cortexa9** and add **`-DUSE_AMP=1`** to extra_compiler_flags (Figure 8).

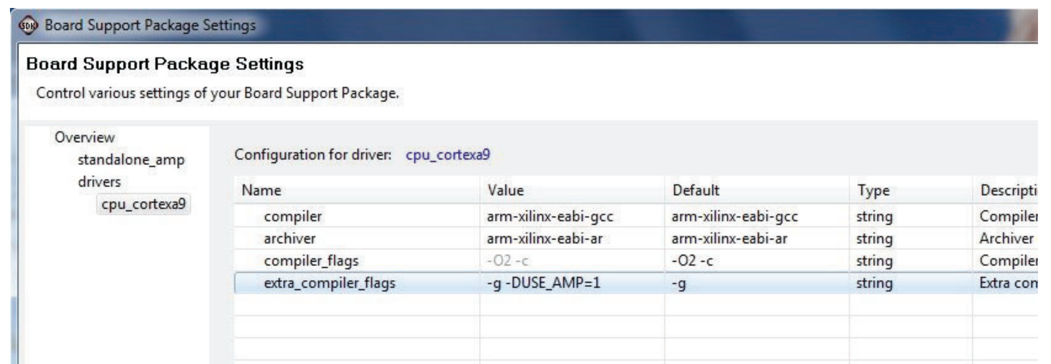

Figure 8: **CPU1 BSP Add USE_AMP**

h. Select **OK**.

2. Create the bare-metal application that will be running on CPU1 and import the included software (Figure 9):
   a. Select **File > new > application_project**.
   b. Enter the project name **app_cpu1**.
   c. Change processor to **ps7_cortexa9_1**.
   d. Change board support package to **Use existing** and select **app_cpu1_bsp**.

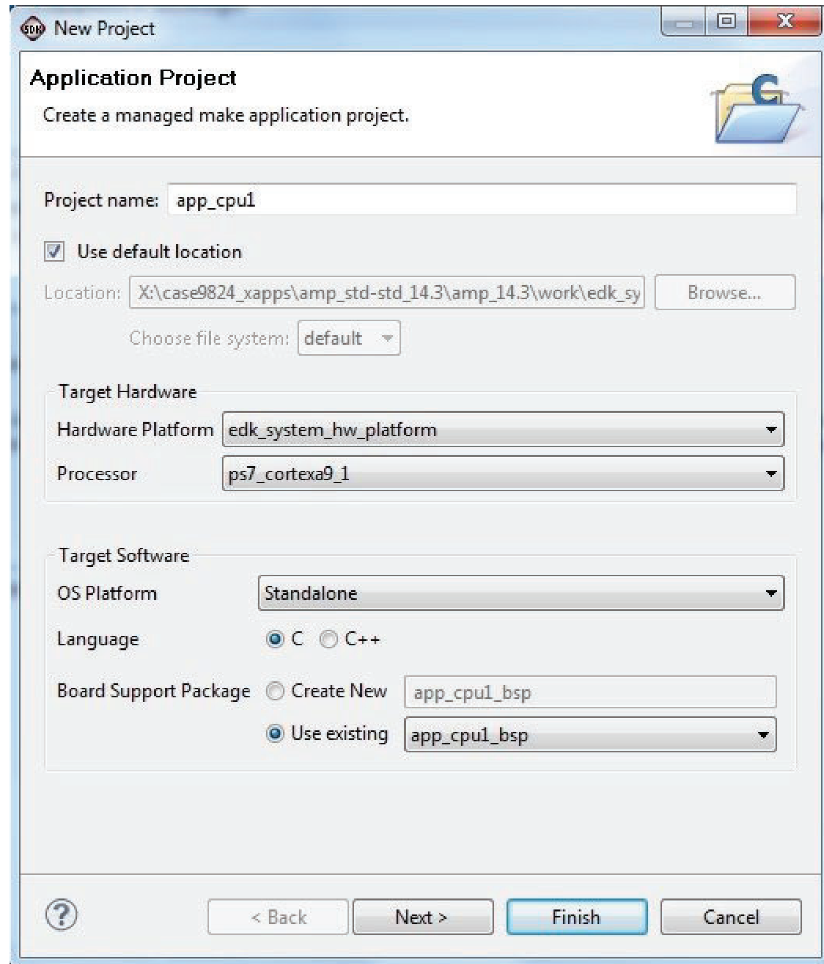

Figure 9: **CPU1 Create Application**

   e. Click **Next**.

f. Choose the Empty Application template (Figure 10).

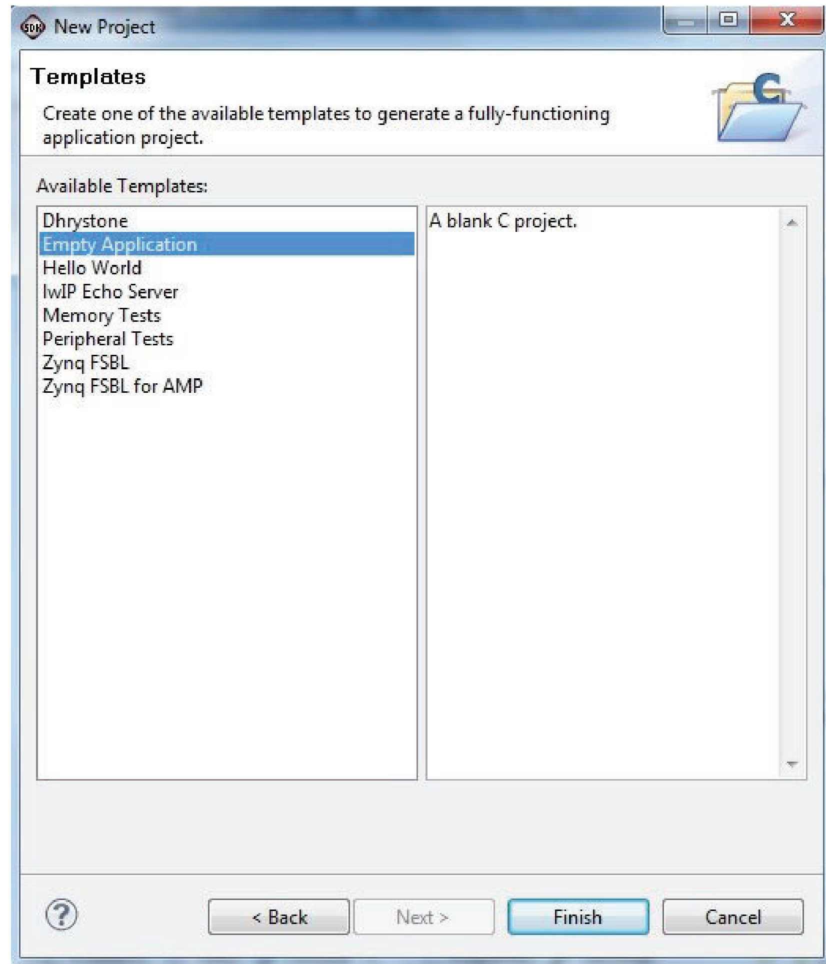

Figure 10: **CPU1 Empty Application**

g. Click **Finish**.

h. In SDK’s Project Explorer tab, expand **app_cpu1** and right click on the **src** folder.

i. Select **Import** (Figure 11).

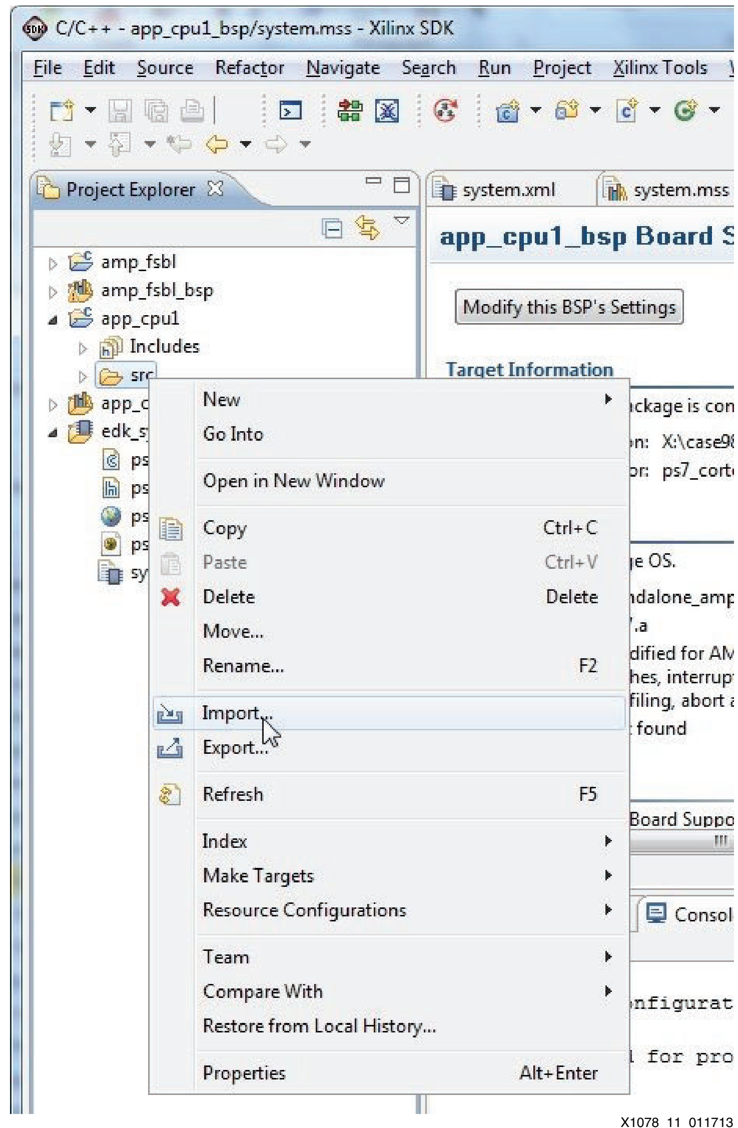

Figure 11: **app_cpu1 SDK Import**

j. Select **General > File_System** (Figure 12).

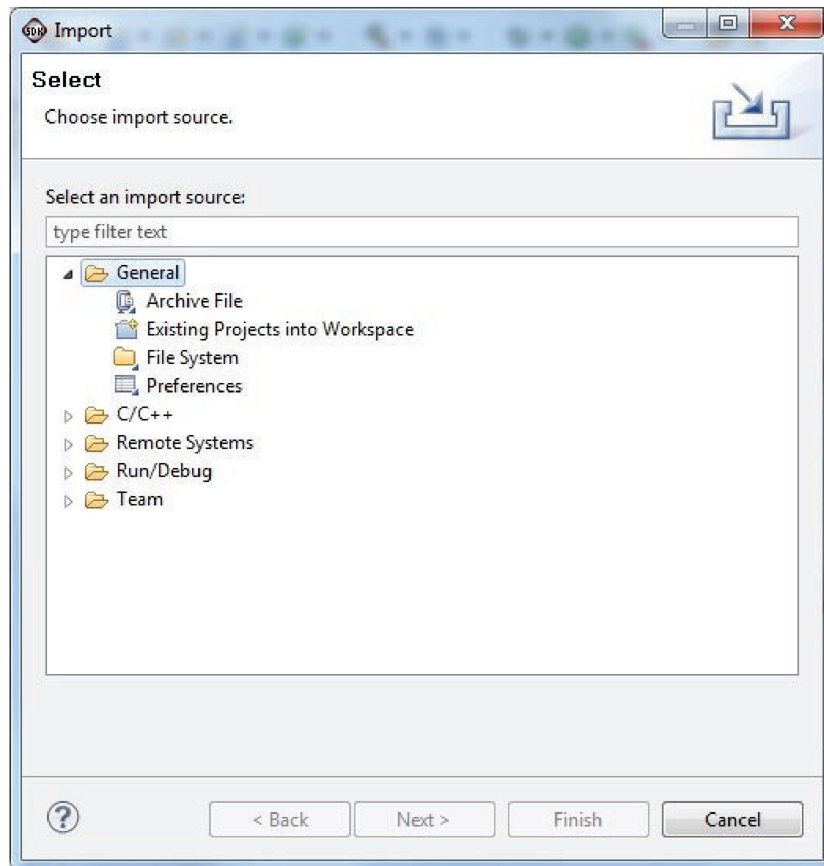

Figure 12: **CPU1 Import General File System**

k. Click **Next**.

l. Browse to and select the included directory `design\src\apps\app_cpu1`.

m. In the left window pane, select the **app_cpu1** folder but do not add a checkmark. In the right window pane, select all files (Figure 13).

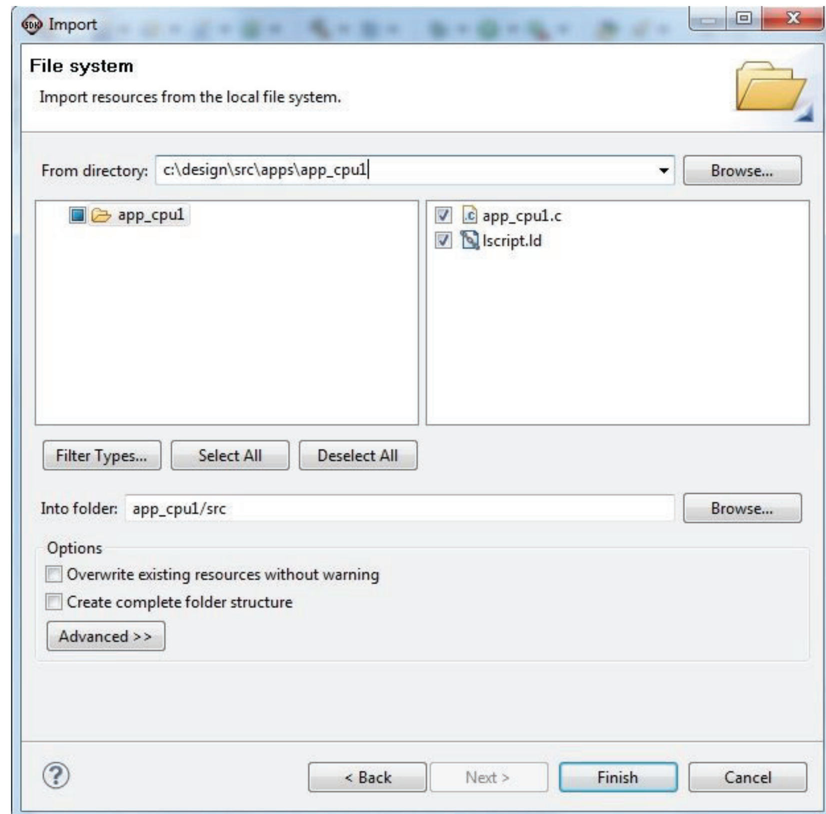

Figure 13: **CPU1 Select Files to Import**

n. Click **Finish**.

o. Select **yes** to overwrite `lscript.ld`.

3. After SDK completes compiling the new application, the ELF is available at `design\work\edk_system\SDK\Workspace\app_cpu1\Debug\app_cpu1.elf`.

Creating Linux Application RWMEM

This utility provides simple read and write accesses to memory locations from the Linux console much like the mrd and mwr commands within Xilinx Microprocessor Debug (XMD). This application has already been compiled and is available at `design\generated_files\SDK_apps\rwmem.elf`.

1. Create a blank Linux application (Figure 14):
   a. Select **File > new > Application_Project**.
   b. Set the Project Name to **rwmem**.
   c. Change the OS Platform to **Linux**.

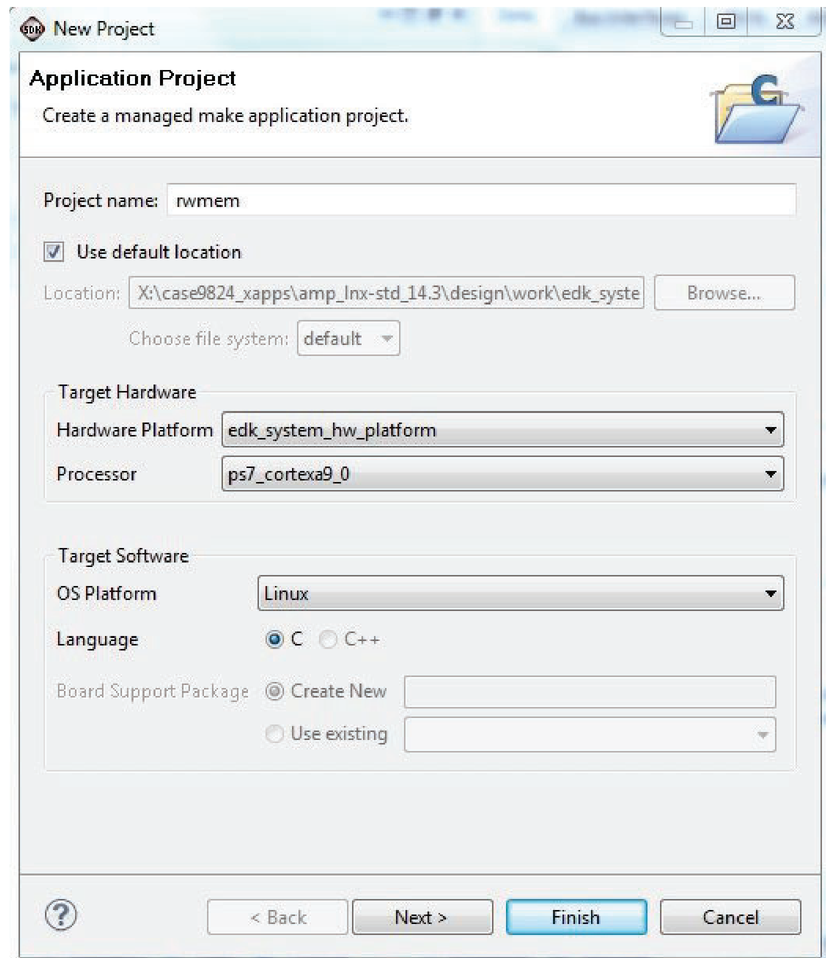

Figure 14: **Create RWMEM**

   d. Click **Next**.

e. Select the **Linux Empty Application** template (Figure 15).

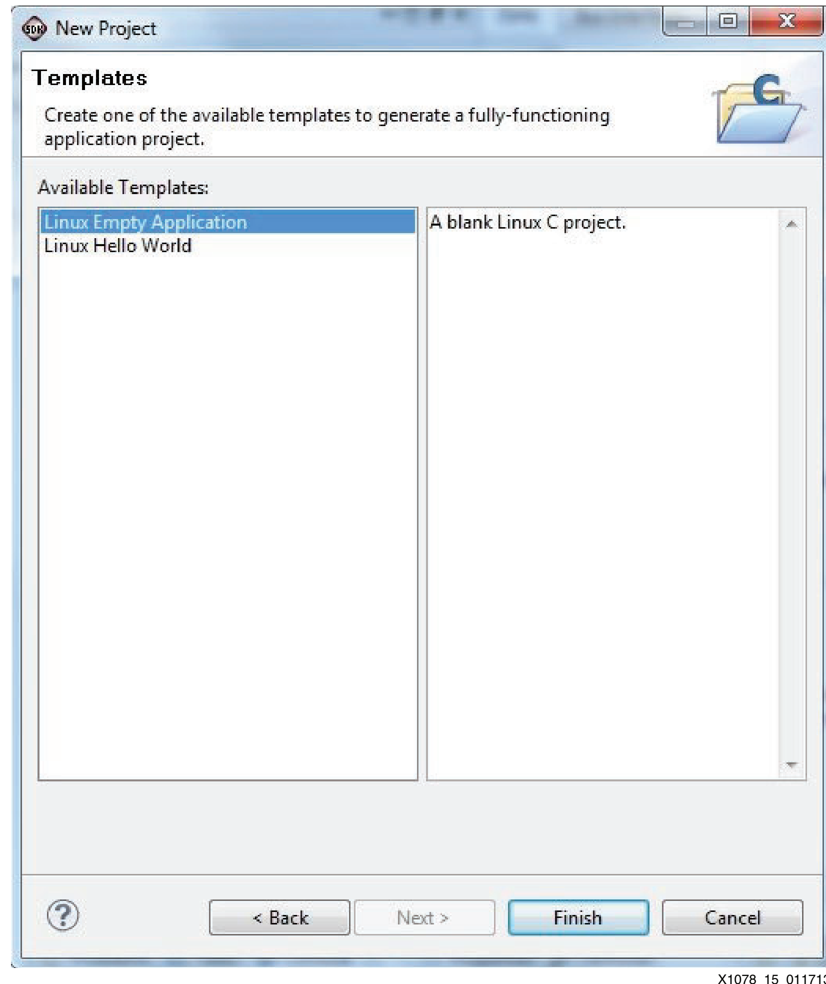

Figure 15: **Linux Empty Application**

f. Click **Finish**.

2. Import the included rwmem source.
   a. In SDK's Project Explorer tab, expand **rwmem** and right click on the **src** folder.
   b. Select **Import**.
   c. Select **General > File_System**
   d. Browse to and select the included directory `design\src\apps\rwmem`.
   e. Click **OK**.
3. In the right window pane, select **rwmem.c**. Do not select anything in the left window pane. Click **Finish**.
4. After SDK completes compiling the new application, the ELF file is available at `design\work\edk_system\SDK\Workspace\rwmem\Debug\rwmem.elf`. A precompiled version is also available at `design\generated_files\SDK_apps\rwmem.elf`.

Creating Linux Application Soft UART

The Soft UART application runs on Linux and continuously monitors OCM to receive data from CPU1. The data is echoed to the Linux terminal.

This application has already been compiled and is available at `design\generated_files\SDK_apps\softUart.elf`.

1. Follow the same steps as Creating Linux Application RWMEM but name the project softUart and import the source from the included `design\src\apps\softUart`.
2. After SDK completes compiling the new application, the ELF file is available at `design\work\edk_system\SDK\Workspace\softUart\Debug\softUart.elf`. A precompiled version is also available at `design\generated_files\SDK_apps\softUart.elf`.

Creating Linux Kernel

Refer to the wiki pages at http://wiki.xilinx.com for instructions on how to download and compile the kernel. This design example uses the build with the tag xilinx-14.3-build2. A copy of a pre-compiled kernel is included at `design\generated_files\linux\uImage`.

The commands used are listed here:

1. Download the kernel using git as described in the wiki and set the build to **`xilinx-14.3-build2`**:

```
git clone git://git.xilinx.com/linux-xlnx.git
cd linux-xlnx
git pull origin xilinx-14.3-build2
```

   or

```
git checkout -b xilinx-14.3-build2 xilinx-14.3-build2
```

2. Build the kernel as described in the wiki:

```
make ARCH=arm xilinx_zynq_defconfig
make ARCH=arm uImage
```

Note: Refer to the wiki for more details.

The kernel is compiled and located at `arch/arm/boot/uImage`.

Creating Linux Device Tree

Refer to the wiki pages at http://wiki.xilinx.com for instructions to compile the device tree. The device tree needs to be changed to instruct Linux SMP to only use one CPU and to decrease the amount of memory available to Linux. A copy of the modified `devicetree.dts` and compiled `devicetree.dtb` is included at `design\generated_files\boot`.

The commands used are listed here:

1. Copy the `zynq-zc702.dts` device tree included with the downloaded linux kernel to a new location:

```
cp arch/arm/boot/dts/zynq-zc702.dts <somewhere>
```

2. Modify the copied device tree to reduce the memory. The memory entry should be:

```
memory {
  device_type = "memory";
  reg = <0x00000000 0x30000000>;
};
```

3. Set the maximum number of CPUs to 1 by adding **`maxcpus=1`** to the bootargs assignment:

```
bootargs = "console=ttyPS0,115200 maxcpus=1 root=/dev/ram rw
ip=:::::eth0:dhcp earlyprintk";
```

4. Compile the new `devicetree.dts` to create `devicetree.dtb` as described in the wiki `scripts/dtc/dtc -I dts -O dtb -o <somewhere>/devicetree.dtb <somewhere>/zynq-zc702.dts`.

Note: Refer to the wiki for more details.

The device tree that is available from the kernel uses DHCP for Ethernet. If the card does not have access to a network with a DHCP server, the Linux boot time is greatly decreased if the IP address is statically set. To do so, the portion of bootargs should be changed from:

```
ip=:::::eth0:dhcp
```

to

```
ip=110.0.0.5::110.0.0.1:255.255.255.0::eth0:dhcp
```

Creating U-Boot

Refer to the wiki pages at http://wiki.xilinx.com for instructions on how to download and compile U-Boot. This design example uses the build with the tag xilinx-14.3-build2. A copy of U-Boot has already been compiled and is included at `design\generated_files\linux\u-boot.elf`.

The commands used are listed here:

1. Download U-Boot, using git as described in the wiki and set the build to xilinx-14.3-build2:

```
git clone git://git.xilinx.com/u-boot-xlnx.git
cd u-boot-xlnx
git pull origin xilinx-14.3-build2
```

or

```
git checkout -b xilinx-14.3-build2 xilinx-14.3-build2
```

2. Build U-Boot as described in the wiki:

```
make zynq_zc70x_config
make
mv u-boot u-boot.elf
```

Note: Refer to the wiki for more details.

U-Boot is compiled and located at `./u-boot.elf`.

Acquiring Root File System

Refer to the wiki pages at http://wiki.xilinx.com for instructions on how to download the root file system ramdisk for version 14.3. A copy of the ramdisk is included at `design\generated_files\linux\uramdisk.image.gz`.

The downloaded ramdisk is already wrapped for usage by U-Boot. If changes such as startup tasks needs to be modified, the downloaded ramdisk needs to be unwrapped, mounted, modified, un-mounted, and then rewrapped. The the steps required are:

1. Unwrap and mount the ramdisk:

```
dd if=./uramdisk.image.gz bs=64 skip=1 of=ramdisk.gz
gunzip ramdisk.gz
mkdir tmp_copy
sudo mount -o loop ramdisk ./tmp_copy
```

2. The ramdisk is now mounted at `./tmp_copy`.
3. Unmount and rewrap the ramdisk:

```
sudo umount ./tmp_copy
gzip ramdisk
<ubootRoot>/tools/mkimage -A arm -T ramdisk -C gzip -d ramdisk.image.gz
uramdisk.image.gz
```

Generating Boot File

The boot file `BOOT.BIN` normally contains the FSBL, FPGA bit file, and the ELF for the application that runs on CPU0 (U-Boot). In this example, the FSBL has been modified to

download more than one application so the second application ELF that runs on CPU1 is included in BOOT.BIN.

The design files contain a batch file, a BootGen configuration file, and a file named cpu1_bootvec.bin (used to trigger the FSBL to stop downloading further data or ELF files).

The configuration file contains the names of the files that are copied to DDR memory. The order of these files is important. For this design, the order is:

1. FSBL ELF
2. CPU0 application
3. CPU1 application
4. Dummy cpu1_bootvec.bin file

A precompiled version of BOOT.BIN is available at design\generated_files\boot\BOOT.BIN. All the files referred to in this step are also precompiled and available at design\generated_files.

Note: The boot file must be named BOOT.BIN.

1. Copy the included directory design\src\bootgen to design\work\bootgen. This directory includes the BootGen batch file (createBoot.bat), a BIF file (bootimage.bif), and a binary file (cpu1_bootvec.bin) that only contains the hexadecimal value 0xFFFFFF00 (swapped for little endian is 0x00, 0xFF, 0xFF, 0xFF). The FSBL recognizes this file's load address of 0xFFFFFFF0 as configured in bootimage.bif. It triggers the FSBL to stop loading ELF or bin files and start running the first ELF that was downloaded.
2. Copy the compiled FSBL ELF from design\work\edk_system\SDK\Workspace\amp_fsbl\Debug\amp_fsbl.elf into design\work\bootgen.

   Note: If the steps were not taken to compile the FSBL in SDK, a copy is provided in the reference design at design\generated_files\SDK_apps\amp_fsbl.elf.
3. Copy the bit file from design\work\edk_system\implementation\download.bit into design\work\bootgen.

   Note: If the steps were not taken to compile the FPGA bit file, a copy is provided in the reference design at design\generated_files\fpga\download.bit.
4. Copy u-boot.elf that was created from the wiki information into design\work\bootgen.

   Note: If the steps were not taken to compile the application, a copy is provided in the included design at design\generated_files\linux\u-boot.elf.
5. Copy the generated bare-metal application for CPU1 from design\work\edk_system\SDK\Workspace\app_cpu1\Debug\app_cpu1.elf into design\work\bootgen.

   Note: If the steps were not taken to compile the application, a copy is provided in the reference design at design\generated_files\SDK_apps\app_cpu1.elf.
6. Open a Xilinx ISE Design Suite command prompt. This command prompt has the environment set up for the Xilinx tools.
7. In the command prompt, change the directory to design\work\bootgen.
8. Run the createBoot.bat file. This creates the boot file BOOT.BIN in the current directory.

Copying Files to SD Card

The Zynq AP SoC requires these files to be present on the SD card in order to boot Linux:

1. BOOT.BIN (contains the FSBL, BIT file, U-Boot, and application for CPU1)

2. uramdisk.image.gz (ramdisk file system extracted to memory by U-Boot)
3. devicetree.dtb (used by U-Boot and Linux for device information)
4. uImage (Linux kernel loaded and executed by U-Boot)

In addition, the two Linux applications are added to the SD card so that they are available to run within Linux:

1. rwmem.elf (allows read/write access to memory)
2. softUart.elf (runs in the background, monitors the OCM, and prints to stdout)

To copy the files to the SD card:

1. devicetree.dtb: This can be the user's devicetree.dtb created from the wiki information or copied from the design\generated_files\boot\devicetree.dtb included in the reference design.
2. uramdisk.image.gz: This can be the user's own file created from the wiki information or copied from the design\generated_files\linux\uramdisk.image.gz included in the reference design.
3. uImage: This can be the user's own file created from the wiki information or copied from the included design\generated_files\linux\uImage included in the reference design.
4. BOOT.BIN: This can be the user's design\work\bootgen\BOOT.BIN created from the above steps or copied from the design\generated_files\boot\BOOT.BIN included in the reference design.
5. rwmem.elf: This can be the user's design\work\edk_system\SDK\Workspace\rwmem\Debug\rwmem.elf or the design\generated_files\SDK_apps\rwmem.elf included in the reference design.
6. softUart.elf: This can be the user's own file from design\work\edk_system\SDK\Workspace\rwmem\Debug\softUart.elf or the design\generated_files\SDK_apps\softUart.elf included in the reference design.

Running the Design

Hardware Requirements

- ZC702 evaluation board
- 12V AC adapter power supply
- USB type-A to USB mini-B cable (for UART communications)
- TeraTerm Pro (or similar) terminal program
- USB-UART drivers from Silicon Labs [Ref 2]

Hardware Setup

Follow the board setup instructions in the "TRD Demonstration Procedure" section of *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.3) Getting Started Guide* [Ref 2]. For this design, a mouse, keyboard, USB hub, monitor, and monitor cable are not required.

The hardware setup configures the ZC702 demonstration board to boot from the SD card. The terminal program should be configured to listen to the correct COM port with a baud rate of 115200.

When the design is powered up, the board boots. CPU0 then starts running U-Boot and boots Linux. When booting from the SD card, the system can take up to 18 seconds before an output appears on the UART. This UART is dependent upon a third-party driver. For further details, refer to *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit*

(ISE Design Suite 14.3) Getting Started Guide. After boot, the console displays the Linux `zynq>` prompt (Figure 16). If an Ethernet cable is not connected to the board, the system takes some time to timeout from the *Sending DHCP requests* state.

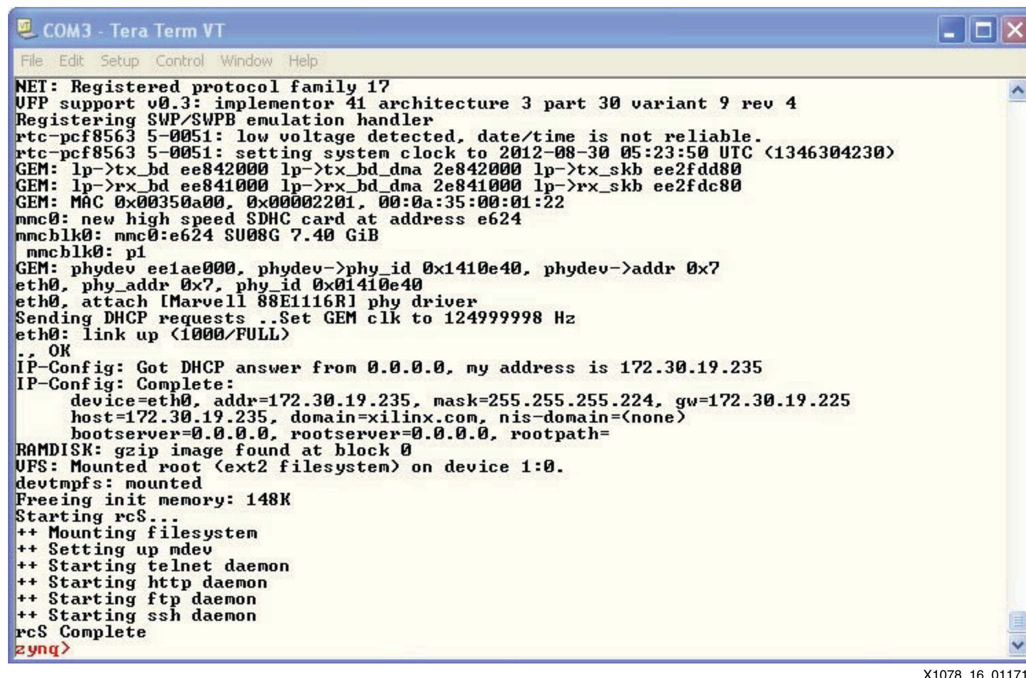

Figure 16: **Console Output**

During boot, the PS bootloader detects that the mode pins have been configured to boot from the SD card. In turn, the PS bootloader opens the `BOOT.BIN` file and searches for the block of data that has been flagged with bootloader. As seen in the `bootimage.bif` file, `amp_fsbl.elf` has this flag. The bootloader loads this file into DDR memory and starts running it. In turn, the FSBL loads the BIT file, U-Boot ELF, CPU1's ELF, and then the dummy file `cpu1_bootvec.bin`. At this point, the FSBL running on CPU0 jumps to the execution address of the first application that was loaded after the FSBL.

The soft UART application is started in linux with the command **`/mnt/softUart.elf &`**. This command runs softUart in the background. SoftUart continues to monitor shared OCM memory at locations `0xFFFF9000` (COMM_TX_FLAG_OFFSET) and `0xFFFF9004` (COMM_TX_DATA_OFFSET). Whenever a 1 is present at COMM_TX_FLAG_OFFSET, softUart reads the value found at COMM_TX_DATA_OFFSET and temporarily stores the value in a string array. When a value of `0x0A` (\n) is received, the string array is displayed on STDOUT. Every time softUart reads a value from COMM_TX_DATA_OFFSET, it clears the COMM_TX_FLAG_OFFSET content, which signals to CPU1 that another character can be sent towards the softUart application running on Linux.

The second CPU is started with the command **`/mnt/rwmem.elf 0xfffffff0 0x30000000`**. Location 0xFFFF8000 should start incrementing every second and can be viewed by using the command **`/mnt/rwmem.elf 0xffff8000`**.

At this point, the bare-metal app running on CPU1 only prints to the softUart when an interrupt is received from the PL. The PL contains a custom core that generates an interrupt from an input signal to the core or from a register within the core. A ChipScope analyzer VIO core is connected to the input of the custom core. Thus, either software or the ChipScope analyzer VIO core can create an interrupt to CPU1. A ChipScope Integrated Logic Analyzer (ILA) core is also located in the design to monitor the IRQ signal.

To force an interrupt to CPU1, the Linux command **`/mnt/rwmem.elf 0x78600000 1`** is entered (Figure 17). Refer to Address Map, page 3 to see the register bit that was just set. By

setting this bit, an interrupt was sent to CPU1 and CPU1's interrupt service routine printed "CPU1: IRQ clr 0" to the soft UART application via the OCM memory.

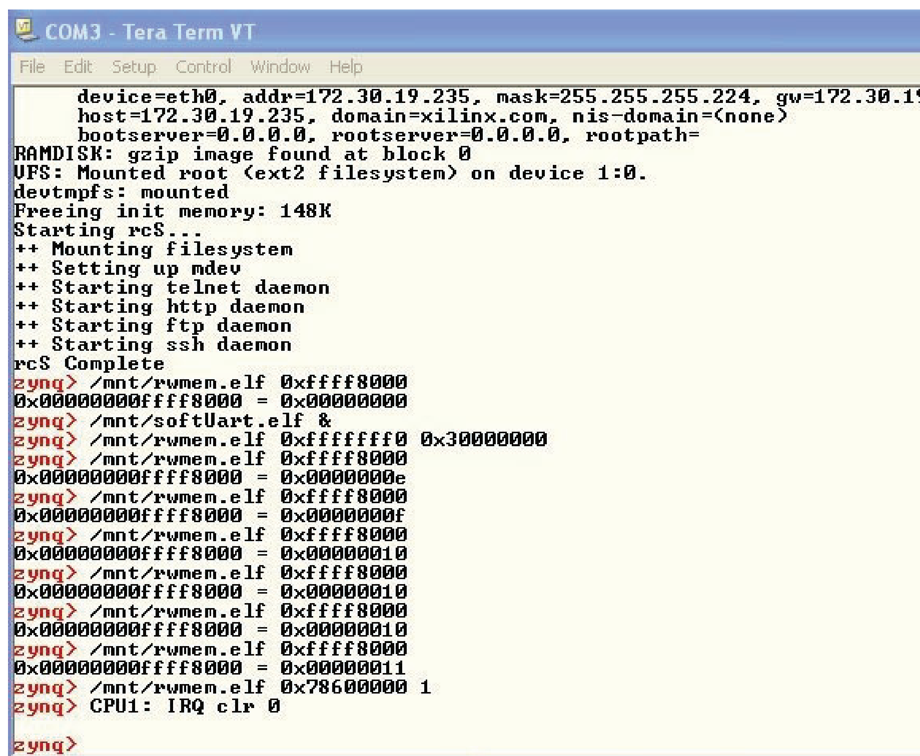

Figure 17: **Console Output of Forced Interrupt**

These steps use the ChipScope analyzer ILA core to measure how long the IRQ signal is active (showing IRQ latency) and create interrupts using the ChipScope analyzer VIO console:

1. While the design is running, start ChipScope analyzer.
2. Connect to the JTAG chain. ChipScope analyzer displays the two devices (ARM_DAP and XC7Z020) that are in the chain. Click **OK**.
3. Open the provided ChipScope configuration file by selecting **File > open_project**. Click **No** to saving changes, and browse to the ChipScope configuration file at `design\src\chipscope\csdefaultproj.cpj`.

   Note: The ILA trigger is already set up to trigger when IRQ is High.
4. Select **UNIT:1 Trigger Setup** and arm the trigger.
5. The VIO console window contains a pulse button called SyncOut[0] (Figure 18). Push this button.

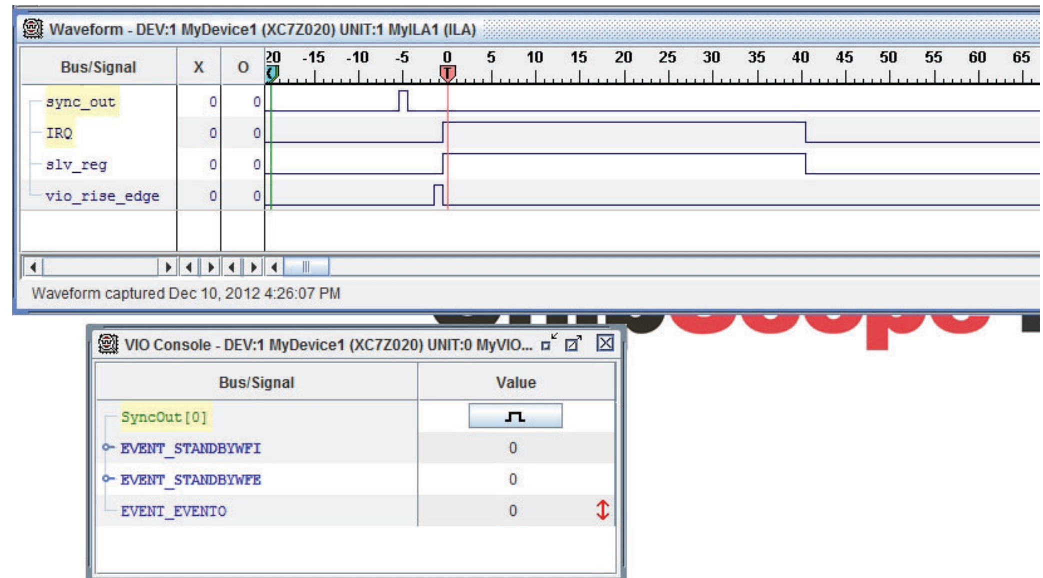

Figure 18: **ChipScope Analyzer Capture of First IRQ**

The softUART application running in the Linux console prints something like "CPU1: IRQ clr 1" (Figure 19).

```
COM3 - Tera Term VT
File  Edit  Setup  Control  Window  Help
RAMDISK: gzip image found at block 0
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing init memory: 148K
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting ssh daemon
rcS Complete
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x00000000
zynq> /mnt/softUart.elf &
zynq> /mnt/rwmem.elf 0xfffffff0 0x30000000
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x0000000e
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x0000000f
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x00000010
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x00000010
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x00000010
zynq> /mnt/rwmem.elf 0xffff8000
0x00000000ffff8000 = 0x00000011
zynq> /mnt/rwmem.elf 0x78600000 1
zynq> CPU1: IRQ clr 0

zynq> CPU1: IRQ clr 1
```

X1078_19_011713

Figure 19: **Console Output after ChipScope Analyzer Trigger**

Note: Every time the virtual button is pressed, an interrupt is created and CPU1's IRQ service routine prints a string towards the softUart. Figure 20 shows how subsequent IRQs have much lower interrupt latency due to the IRQ service routine being cached after servicing the first interrupt.

The bare-metal application that services the interrupt is located in DDR memory. When the first interrupt occurs, CPU1 is instructed to jump to the service routine. This jump causes the instructions located in DDR memory to be read into cache and executed. During execution, the service routine finishes by clearing the interrupt signal being generated by the embedded core. In Figure 18, there is a delay of over 65 clocks between the interrupt being asserted and the service routing clearing the control bit. The delay of the first interrupt could vary depending on whether a DDR refresh is occurring at the same time as the fetching of the service routine.

After the first IRQ occurs, the service routine is located in cache so fetches of the instructions for the routine are sourced by the cache instead of the slower, less deterministic DDR memory. As seen in Figure 20, the interrupt service is complete after 25 clocks. This delay is almost one third of the delay for the first, non-cached, interrupt service.

The time difference between the first and later interrupt services could be reduced by moving the service routine into non-cached OCM.

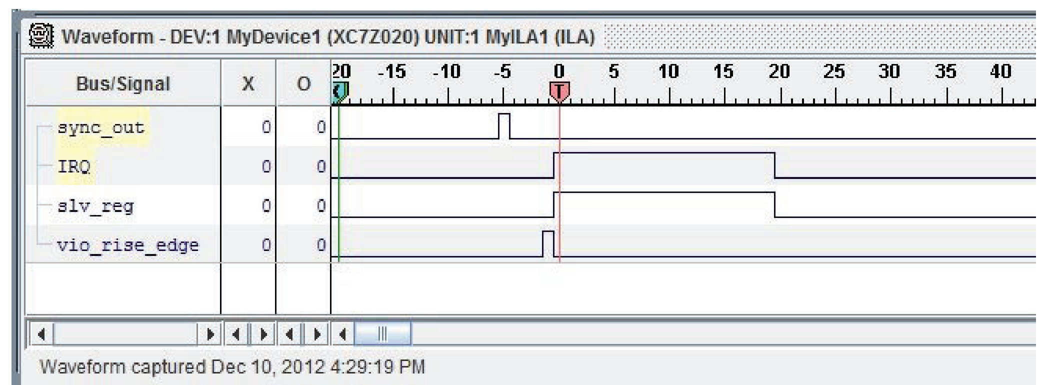

Figure 20: **ChipScope Analyzer Subsequent Capture**

Debugging the Design

SDK can be used to connect and debug the application running on CPU1.

XMD provides a command shell and GNU debugger (GDB) server that connects to the CPU via the JTAG cable. Normally, SDK automatically starts XMD in the background when starting to debug an application. For this example design, XMD is manually started to connect to CPU1. SDK is then instructed to connect to the XMD GDB server during debug.

Because FSBL was used to boot the design, there is no need to re-initialize the PS registers. Care must be taken not to reset the full PS to not upset Linux running on CPU0.

1. Connect the platform cable to the ZC702 board. Ensure the jumper options are configured for the correct debug cable.

2. From SDK, start XMD and connect to CPU1 (Figure 21):
   a. In SDK, open a Xilinx command shell by selecting **Xilinx_Tools** > **Launch_shell**.
   b. In the new command shell, enter **`xmd`**.
   c. At the XMD prompt, enter the command **`connect arm hw -debugdevice cpunr 2`**.
   d. XMD should respond with the TCP port number 1234.

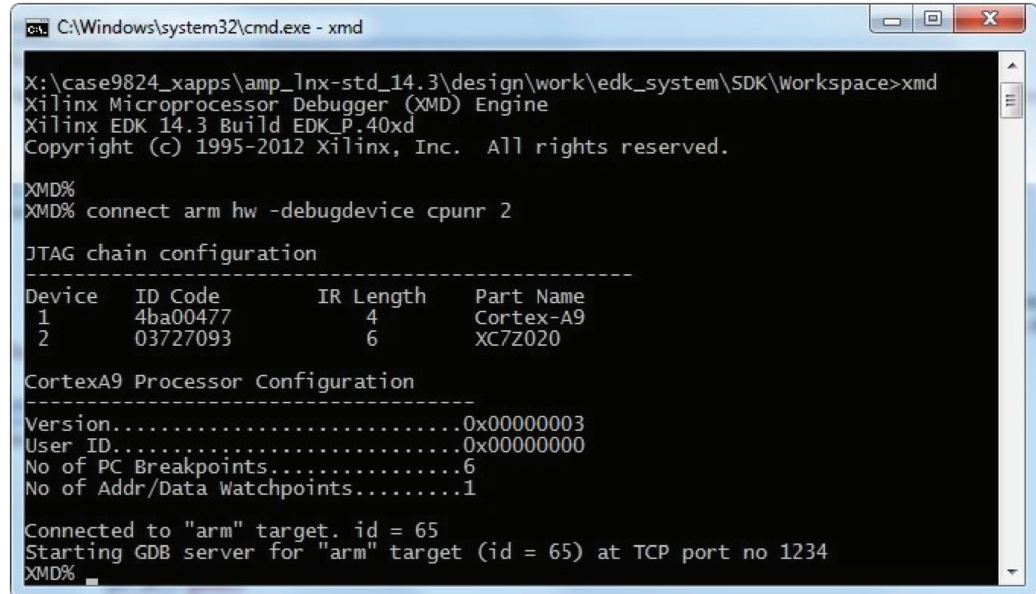

Figure 21: **Connect XMD to CPU1**

A GDB server is now running and listening to TCP port 1234. When XMD connects, CPU1 is halted.

3. Start debugging CPU1 in SDK:

   a. In the SDK project explorer window, right click **app_cpu1** and select **debug_as > debug_configurations** (Figure 22).

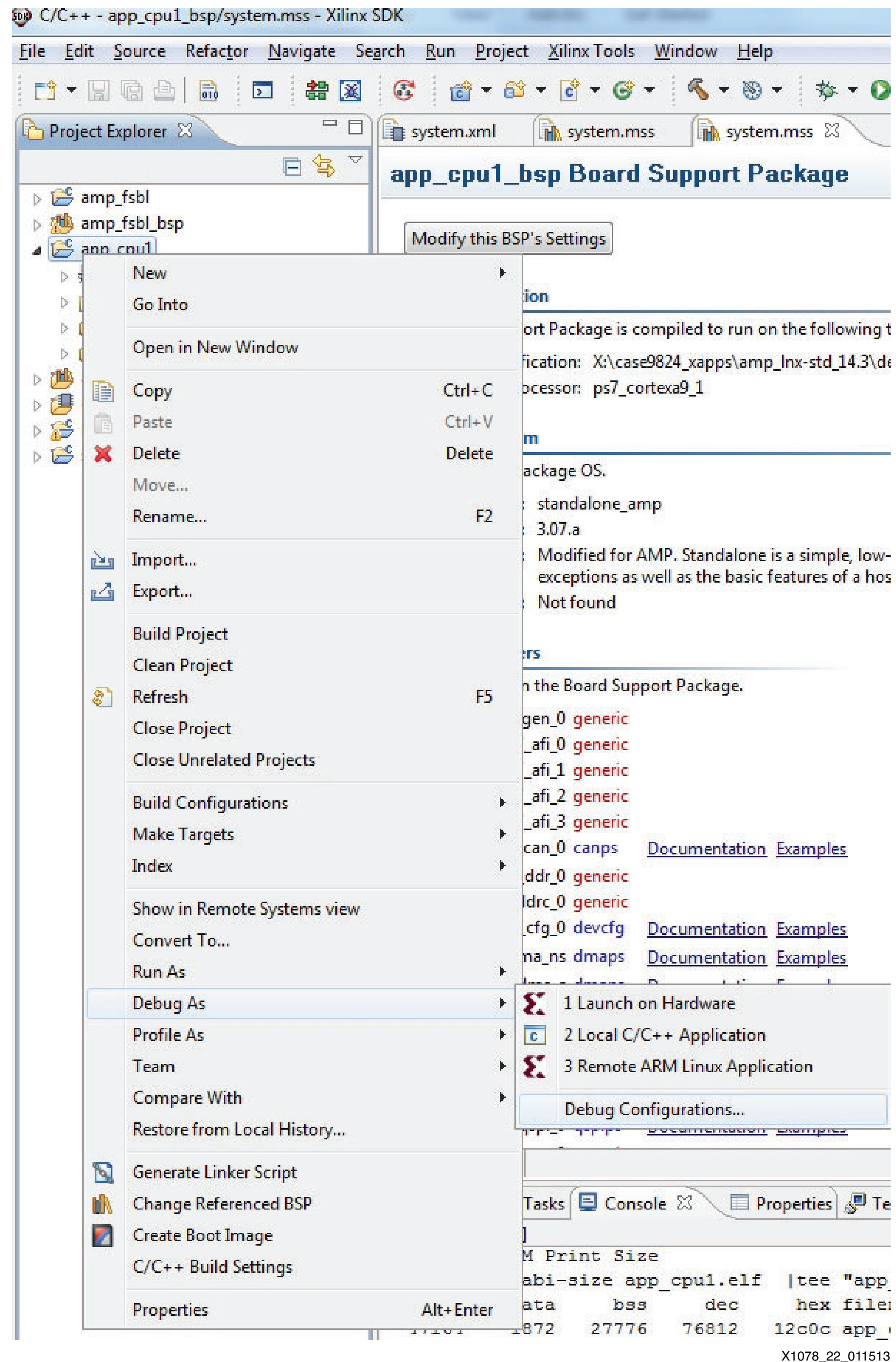

Figure 22: **CPU1 Debug Configuration**

b. Highlight **Xilinx C/C++ ELF** and select the **New launch configuration** icon at the top left (Figure 23).

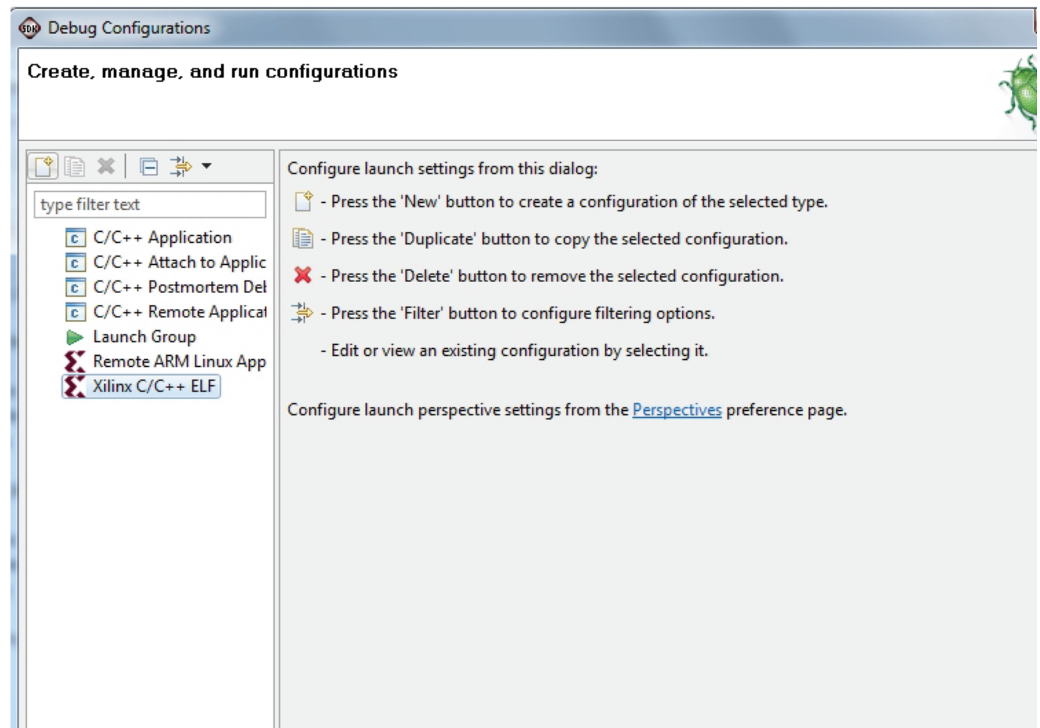

Figure 23: **CPU1 Debug Configuration**

c. The configuration name is automatically set to **app_cpu1 Debug** (Figure 24).

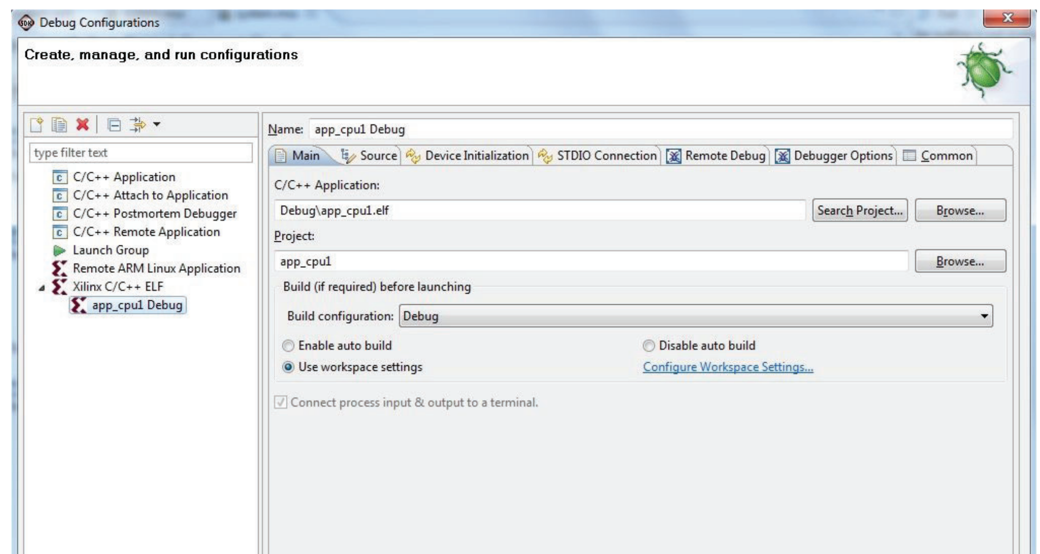

Figure 24: **CPU1 Debug Configuration Name**

d. Select the **Device Initialization** tab and delete the contents of the **Path to initialization TCL file** field. Initialization has already been done by Linux and FSBL (Figure 25).

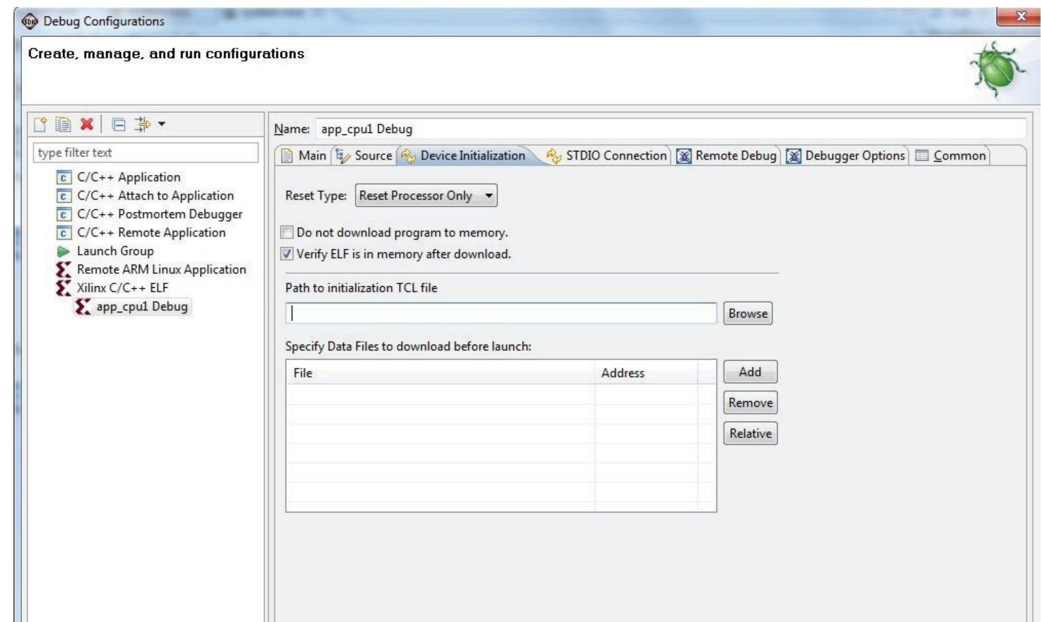

Figure 25: **CPU1 Debug Initialization**

e. Select the **Remote Debug** tab.

f. Instruct SDK to connect to the externally created GDB server by checking the box next to **Connect to gdbserver on a different machine**. The IP address should default to **localhost** and the port should be **1234** (Figure 26).

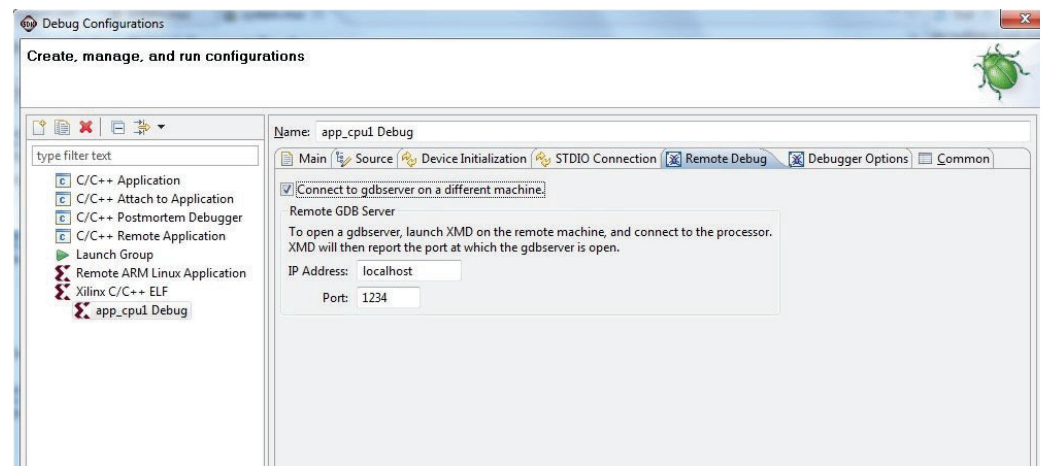

Figure 26: **CPU1 Remote Debug Configuration**

g. Click **Apply**.

h. Click **Debug**. Click **Yes** to confirm the perspective switch.

i. The application is downloaded and then executed (the ELF download could have been disabled in the Device Initialization Tab). The application stops at a breakpoint at the first executable line in main().

j. Press **resume**, **single step**, and other buttons to continue running the application on CPU1.

Conclusion

The example design demonstrates how to boot the Zynq-7000 AP SoC and start two Cortex-A9 processors with CPU0 running Linux and CPU1 running a bare-metal application. Leveraging the low overhead of a bare-metal application on CPU1, an interrupt, sourced from the PL is serviced and communicated to Linux running on CPU0.

References

This application note uses the following references:

1. UG585, *Zynq-7000 All Programmable SoC Technical Reference Manual*
2. UG926, *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.3) Getting Started Guide*
3. UG873, *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT)*
4. AMBA AXI4 Protocol Specification
   http://www.arm.com/products/system-ip/amba/amba-open-specifications.php
5. UG683, *EDK Concepts, Tools, and Techniques*
6. DS768, *LogiCORE IP AXI Interconnect*
7. UG111, *Embedded System Tools Reference Manual*
8. UG761, *AXI Reference Guide*
9. UG642, *Platform Specification Format Reference Manual*
10. UG029, *ChipScope Pro Software and Cores User Guide*
11. Zynq-7000 Base Targeted Reference Design 14.3
    http://wiki.xilinx.com/zynq-base-trd-14-3

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
02/14/13	1.0	Initial Xilinx release.

Notice of Disclaimer

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps.

Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.