



XAPP1079 (v1.0.1) January 24, 2014

# Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors

Author: John McDougall

## Summary

The Zynq®-7000 All Programmable SoC contains two 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 bare-metal software application, and allowing each processor to communicate with the other through shared memory.

## Included Systems

The design is created and built using Xilinx Platform Studio (XPS) 14.3 and 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 if 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. Each CPU runs a bare-metal application within its own standalone environment. 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 (CPU0 and CPU1) is configured to run its own bare-metal application. In this AMP example, the bare-metal application running on CPU0 is the master of the system and is responsible for:

- System initialization
- Controlling CPU1 startup
- Communicating with CPU1
- Sharing the UART with CPU1

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

- Communicating with CPU0
- Servicing interrupts from a core in the programmable logic (PL)
- Sharing the UART with CPU0

The Zynq SoC processing system (PS) includes resources that are private to each CPU and shared by both CPUs. Care must be taken to prevent both CPUs from contending for these shared resources when running the design in an AMP configuration. Refer to *Zynq-7000 All*

*Programmable SoC Technical Reference Manual* [Ref 1] for further information on shared versus private resources. Though some of the approaches to managing these shared or private resources are application specific, many of the resource management approaches used by this reference design can fundamentally be reused as-is.

Examples of some of the private resources are:

- L1 cache
- Private peripheral interrupts (PPI)
- 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
- UART0

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:

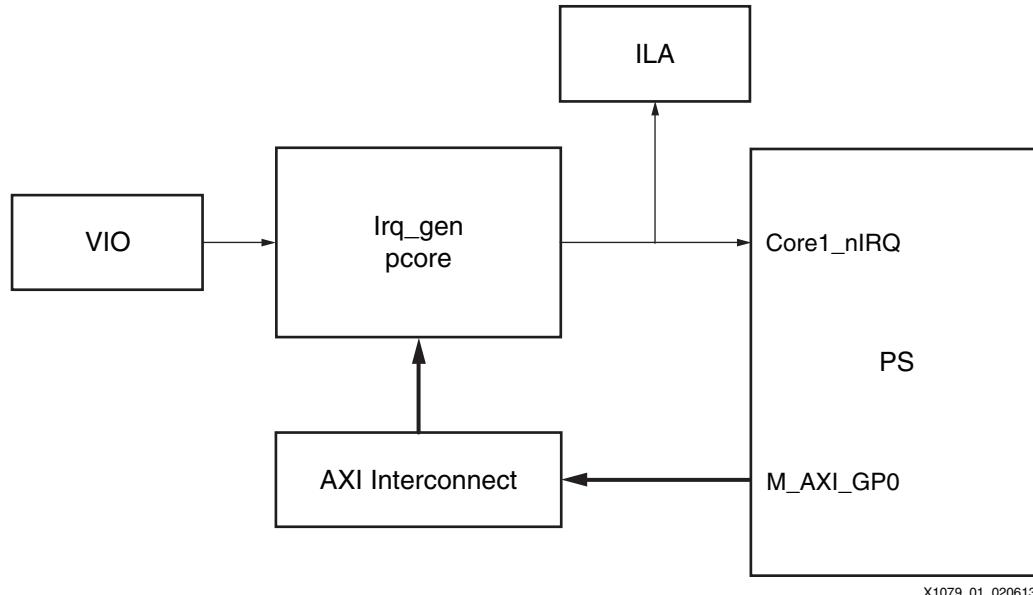
- DDR memory: CPU0 has only been made aware of memory at 0x00100000 to 0x001FFFFF. CPU1 uses memory from 0x00200000 to 0x002FFFFF for its bare-metal application.
- L2 cache: CPU1 does not use L2 cache. L2 cache is a shared resource and CPU0 owns this resource. If CPU1 used L2 cache, L2 cache flushes and invalidates would need to be requested from CPU0 and CPU0 would exercise the action. It is beyond the scope of this example design to include a communication channel that enables CPU1 to request L2 cache interactions.
- 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.
- Timer: CPU1 uses a private timer.
- OCM: Accesses to OCM are handled very carefully by each CPU to prevent contention. A single OCM address location is used as a flag to communicate between the two processors. CPU0 initializes the flag to 0 before starting CPU1. When the flag is zero, CPU0 owns the UART. When the flag is not zero, CPU1 owns the UART. Only CPU0 sets the flag and only CPU1 clears it.

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 analyzer VIO core provides more control for 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. 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 block 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 that allows 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 the CPU1 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 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. The CPU can also 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)
- Bare-metal application for CPU0
- Bare-metal application for CPU1

### FSBL

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

The version of FSBL included in the ISE® Design Suite 14.3 does not support multiple data or ELF files. The current 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 0xFFFFFFFFF0. Upon detection, the FSBL downloads this last file then 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 the *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 1].

### Bare-Metal Application Code

The reference design has both CPU0 and CPU1 running their own bare-metal application code. 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 defined 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 boot 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.

### CPU0 Application

CPU0's application is located in memory starting at address 0x00100000. The linker script is used to set the starting address.

The CPU0 application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched, so OCM exists at addresses 0x00000000 to 0x0002FFFF and addresses 0xFFFFF0000 to 0xFFFFFFFF. Only the high 64 KB of OCM is used by the example design so cache is disabled on addresses 0xFFFFF0000 to 0xFFFFFFFF.
2. Initializes the ICD.
3. Starts CPU1.
4. Prints to the UART.
5. Sets a memory location in OCM that is used as a semaphore flag.
6. Waits for the memory location in OCM that is used as a semaphore flag to be cleared.

The CPU0 application repeats [step 3](#) to [step 6](#) indefinitely.

After the PS powers up and the internal boot ROM completes execution, CPU1 is 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 0xFFFFFFFF0 for a non-zero value, and then continues the loop. If 0xFFFFFFFF0 contains a non-zero value, CPU1 jumps to the fetched address.

CPU0 starts CPU1 (both running bare-metal) by writing the value of 0x00200000 to address 0xFFFFFFFF0 and then running the Set Event (SEV) command. SEV causes CPU1 to wake up, read the value 0x00200000 from address 0xFFFFFFFF0, and then jump to address 0x00200000. The FSBL is responsible for placing the CPU1 ELF at 0x00200000.

### CPU1 Application

The CPU1 application is located in memory starting at address 0x00200000. The linker script is used to set the starting address.

The CPU1 application performs 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 to 0x0002FFFF and addresses 0xFFFF0000 to 0xFFFFFFFF. Only the high 64 KB of OCM is used by this application note, so cache is disabled on addresses 0xFFFF0000 to 0xFFFFFFFF.
2. Initializes the PPI interrupt controller and interrupt subsystem.
3. Waits for a memory location in OCM that is used as a semaphore flag to be set.
4. Prints to the UART. The string printed is chosen dependent on whether or not the interrupt service routine incremented a global variable. If the global variable irq\_count is not zero, CPU1 sets the value to zero.
5. Clears the memory location in OCM that is used as a semaphore flag.

The CPU1 application repeats [step 3](#) to [step 5](#) indefinitely.

### Inter-Processor Communication

The inter-processor communication in the example design is a semaphore flag. When the semaphore is set, CPU1 owns the UART and when it is cleared by CPU1, CPU0 is free to use the UART. This is a simple mechanism to share resources. The OCM memory is chosen because it is a low latency, shared resource. Also, this area of OCM is not cached so the memory accesses are deterministic.

If DDR memory were to be used for the semaphore, there would be a higher latency for accesses during cache misses and less deterministic accesses due to background refresh cycles. DDR memory accesses are also *bursty* in nature so time would be wasted as a write or read burst occurs to access a single 32-bit value.

## Reference Design

The reference design files can be downloaded from:

<https://secure.xilinx.com/webreg/clickthrough.do?cid=203247>

The reference design matrix is shown in [Table 2](#).

**Table 2: Reference Design Matrix**

Parameter	Description
<b>General</b>	
Developer name	John McDougall

Table 2: Reference Design Matrix (Cont'd)

Parameter	Description
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
<b>Simulation</b>	
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
<b>Implementation</b>	
Synthesis software tools/version used	XST 14.3
Implementation software tools/versions used	EDK 14.3
Static timing analysis performed	Yes
<b>Hardware Verification</b>	
Hardware verified	Yes
Hardware platform used for verification	ZC702 board

These files are included in the reference design:

- XPS project
- SDK source files for CPU0 and CPU1 applications
- Generated files:
  - Bit file
  - All files for the SD card
  - Application ELF files for CPU0 and CPU1
- BOOT.BIN build scripts
- Modified bare-metal BSP
- Modified sw\_apps FSBL

Table 3 and Table 4 show the device utilization details.

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

Parameters	Specification/Details	
Device utilization without testbench	Slices	251
	GCLK buffers	2
	PS7	1
	RAMB36	1
HDL language support	Verilog/VHDL	

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

Device	Speed Grade	Package	Pre-Map (Synthesis Constraint)	Post-Route	Slices
XC7Z020	-1	CLG484	240 MHz	146 MHz	251 (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 the Hardware

This section describes how to create the hardware design. The pre-compiled design is available at `design\generated_files\fpga\download.bit` but the following steps must be exercised to export the hardware platform to SDK:

1. Start XPS and open the embedded project at:

`design\work\edk_system\system.xmp`

2. Select **device\_configuration > update\_bitstream**.

After completion, 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**.

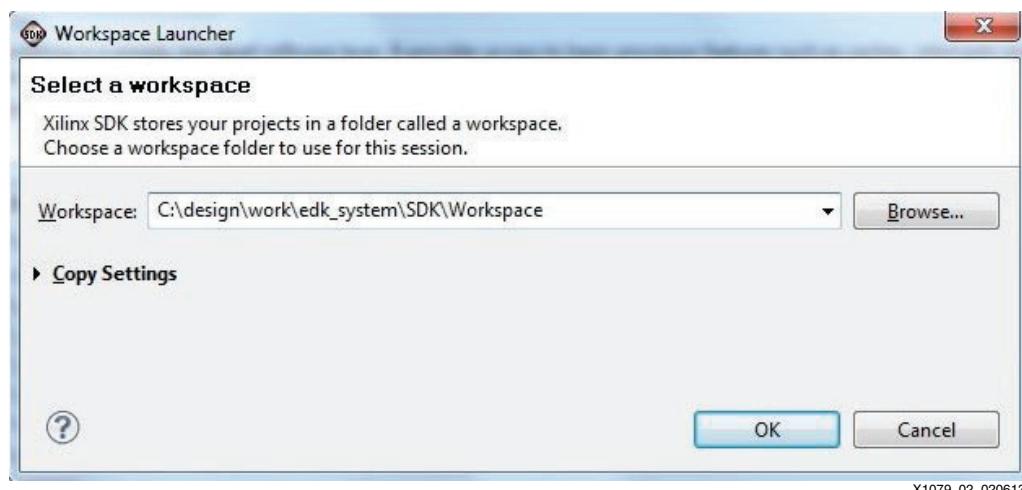
4. Click the **Export & Launch SDK** button.

At this point, XPS exports the embedded system configuration using 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 the `design\work\edk_system\SDK\SDK_Export\hw` directory.

5. SDK displays a dialog box asking where the workspace is located. Browse to and select the `design\work\edk_system\SDK` directory.

6. Click **OK** (once).

7. Add `\Workspace` to the end of the selection, as shown in [Figure 2](#).



X1079\_02\_020613

Figure 2: Select Workspace Directory

8. Click **OK**.

SDK automatically creates the \Workspace subdirectory.

## Generating the Applications

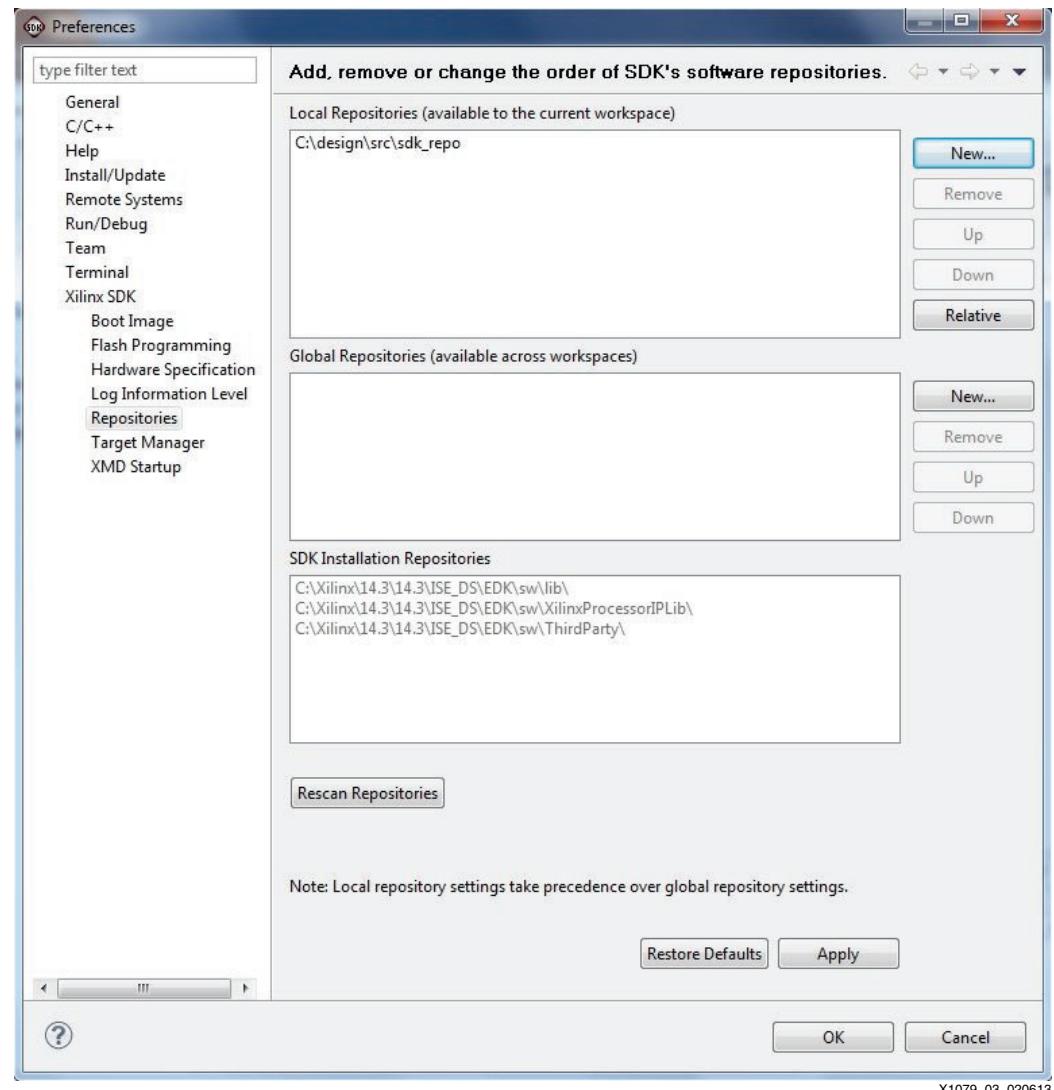
### Configuring SDK

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

To configure SDK:

1. Start SDK and open the workspace at `design\work\edk_system\SDK\Workspace`.  
**Note:** This step is not necessary if XPS was used to export and launch SDK.
2. Point SDK to the included repository that contains the modified standalone BSP and modified FSBL.
  - a. Select **Xilinx\_tools > Repositories**.
  - b. Next to **Local Repositories**, click **New**.
  - c. Browse to, and select the `design\src\ sdk_repo` directory.
  - d. Click **OK**.

The Preferences window is displayed, as shown in [Figure 3](#).



*Figure 3: Select Repository*

### Creating Custom FSBL Application

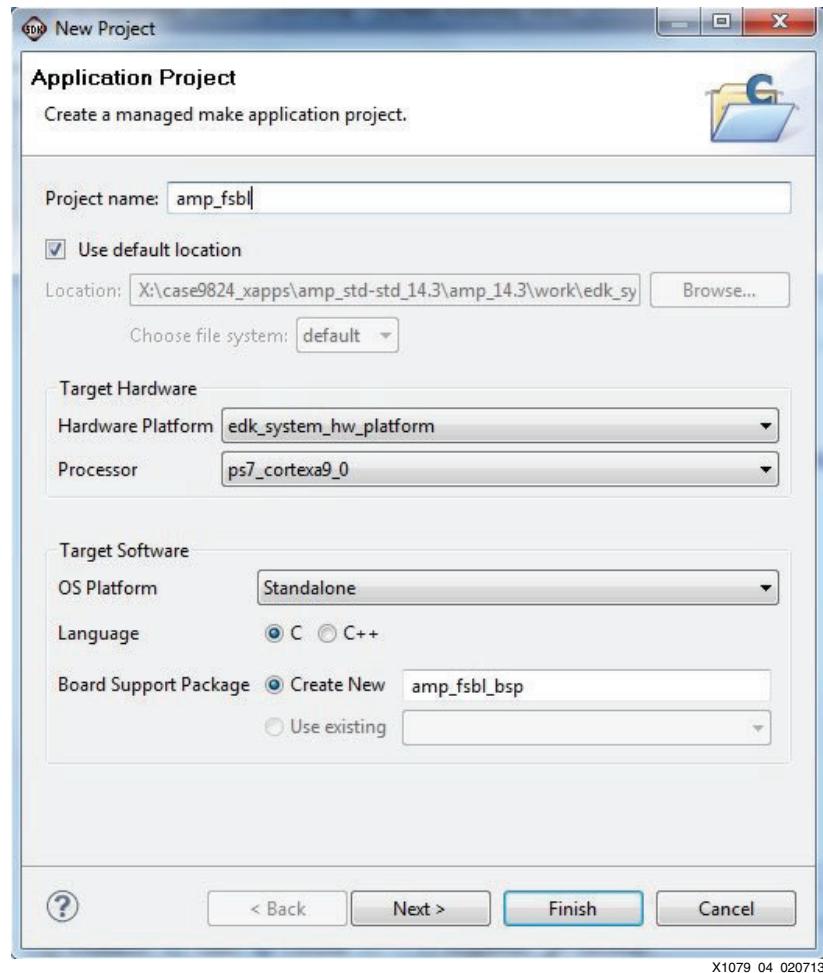
The custom FSBL is created using a new template from the repository.

**Note:** Before proceeding, the SDK must have been configured as described in [Configuring SDK](#).

To create a custom FSBL application:

1. Select **File > New > Application\_Project**.

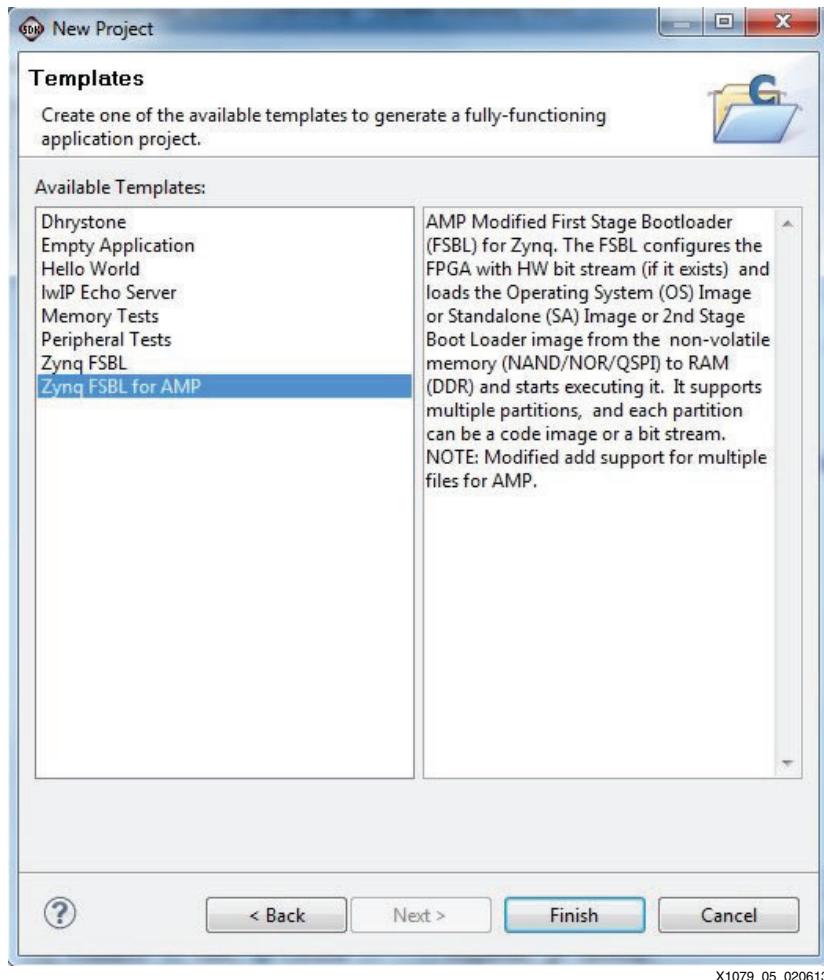
2. In the **Project Name** field enter **amp\_fsbl**, as shown in [Figure 4](#).



*Figure 4: Create Custom FSBL*

3. Click **Next**.

4. In Available Templates select **Zynq FSBL for AMP**, as shown in [Figure 5](#).



*Figure 5: Select Custom FSBL Template*

**Note:** If this template is not available in the list, verify that the repository is set up correctly.

5. Click **Finish**.

When SDK finishes 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 precompiled version is also available at:

```
design\generated_files\SDK_apps\amp_fsbl.elf
```

### **Creating Bare-Metal Application For CPU0**

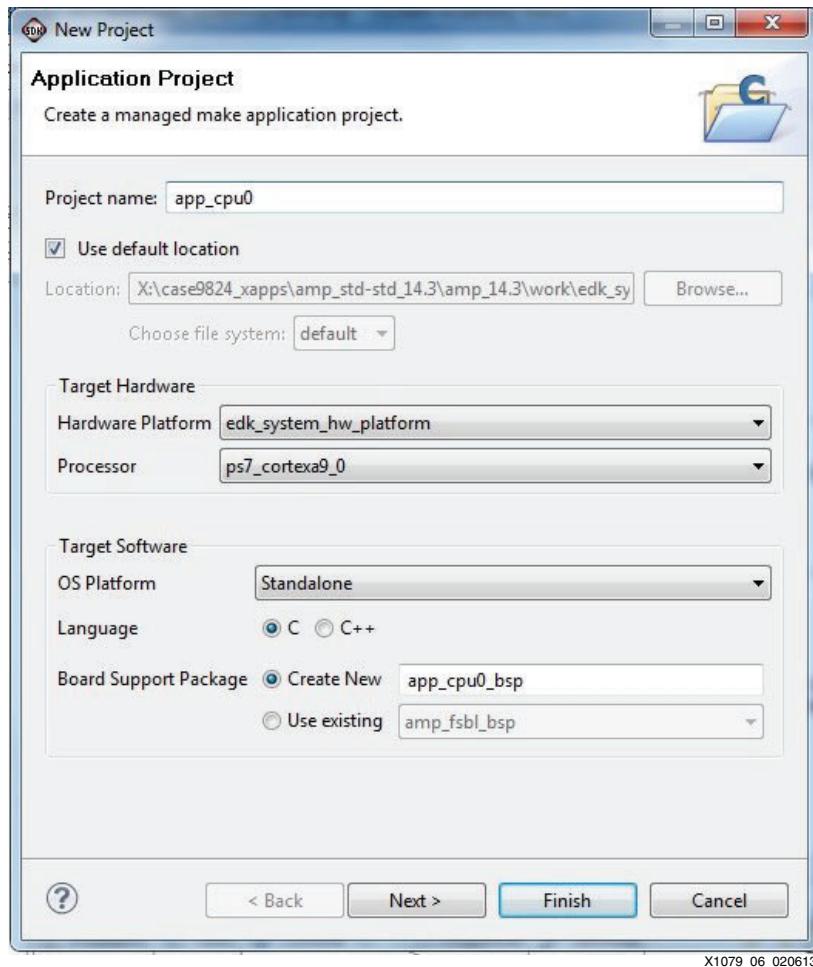
The instructions in this section describe how to create the application ELF that runs on CPU0 after the FSBL copies the application ELF to DDR memory.

**Note:** This application has already been compiled and is available at  
design\generated\_files\SDK\_apps\app\_cpu0.elf.

To create the CPU0 application:

1. Start the SDK.
2. Select **File > New > Application\_project**.

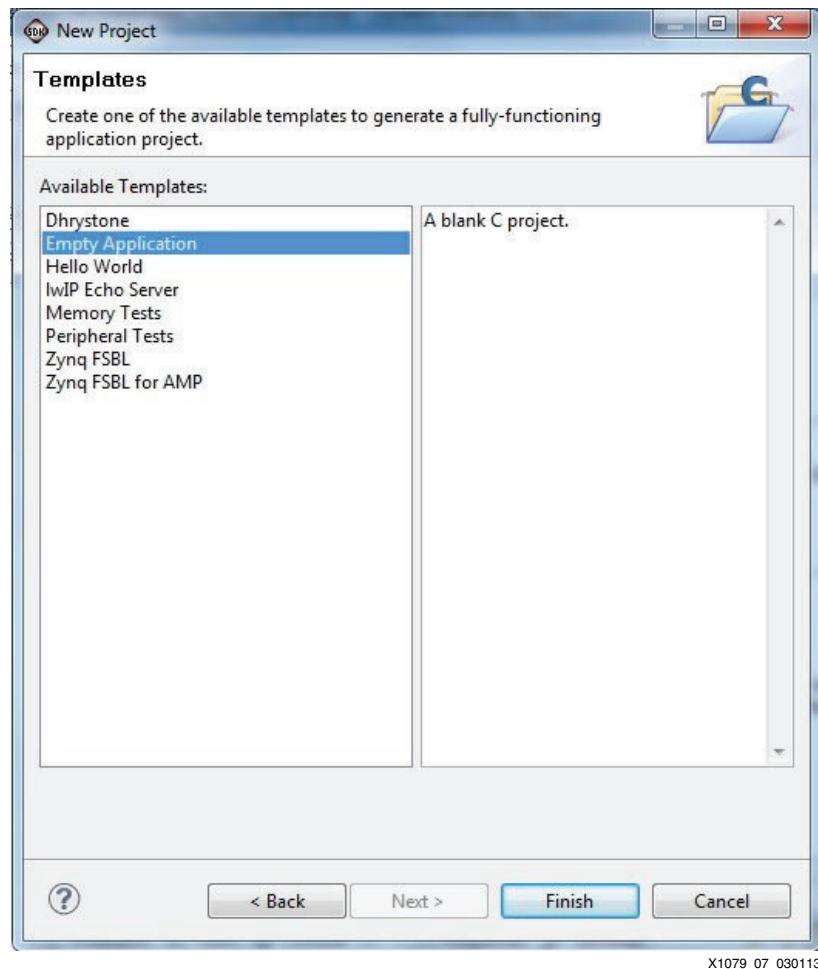
3. Change the Project Name to **app\_cpu0**, as shown in [Figure 6](#).



*Figure 6: Create app\_cpu0*

4. Click **Next**.

5. Select **Empty Application**, as shown in [Figure 7](#).



*Figure 7: CPU0 Empty Application*

6. Click **Finish**.

7. In the Project Explorer tab, expand **app\_cpu0** and right-click on the **src** folder, as shown in Figure 8.

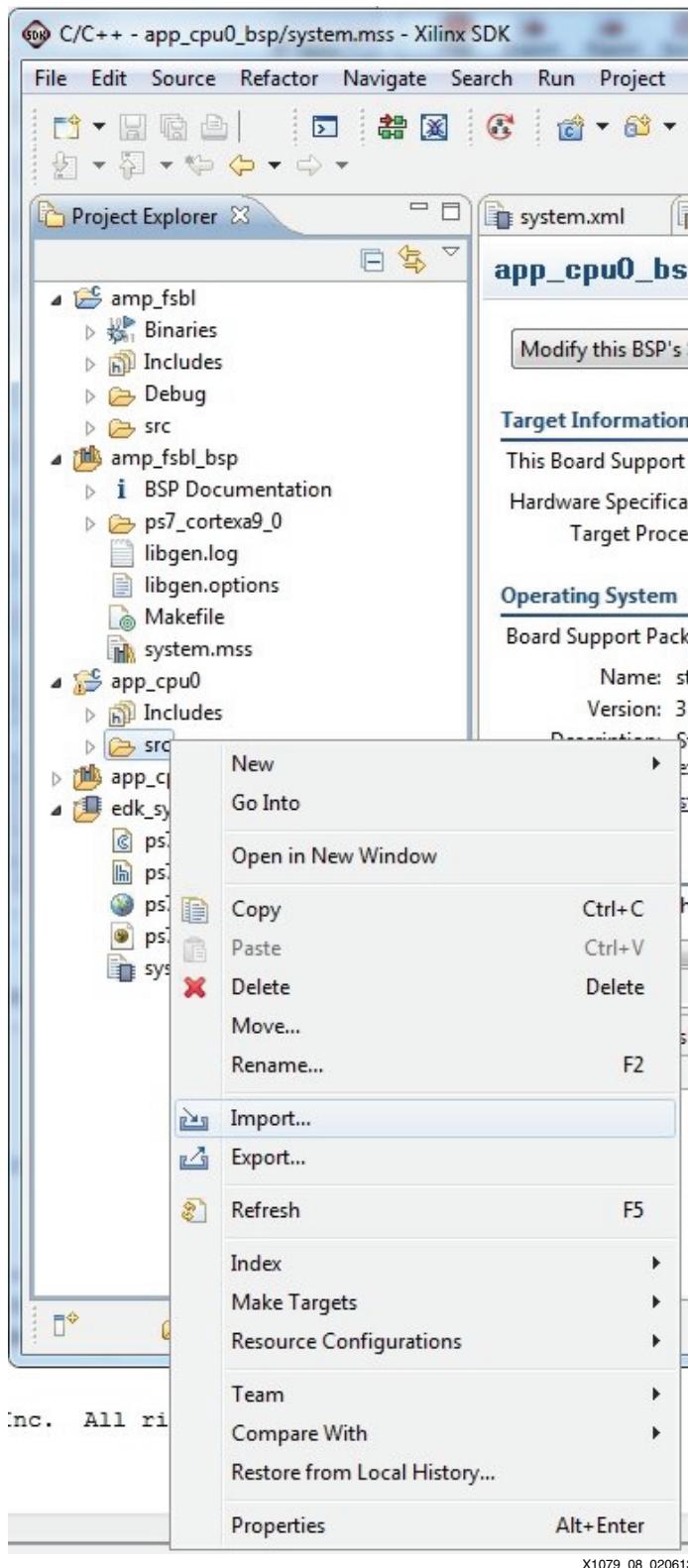
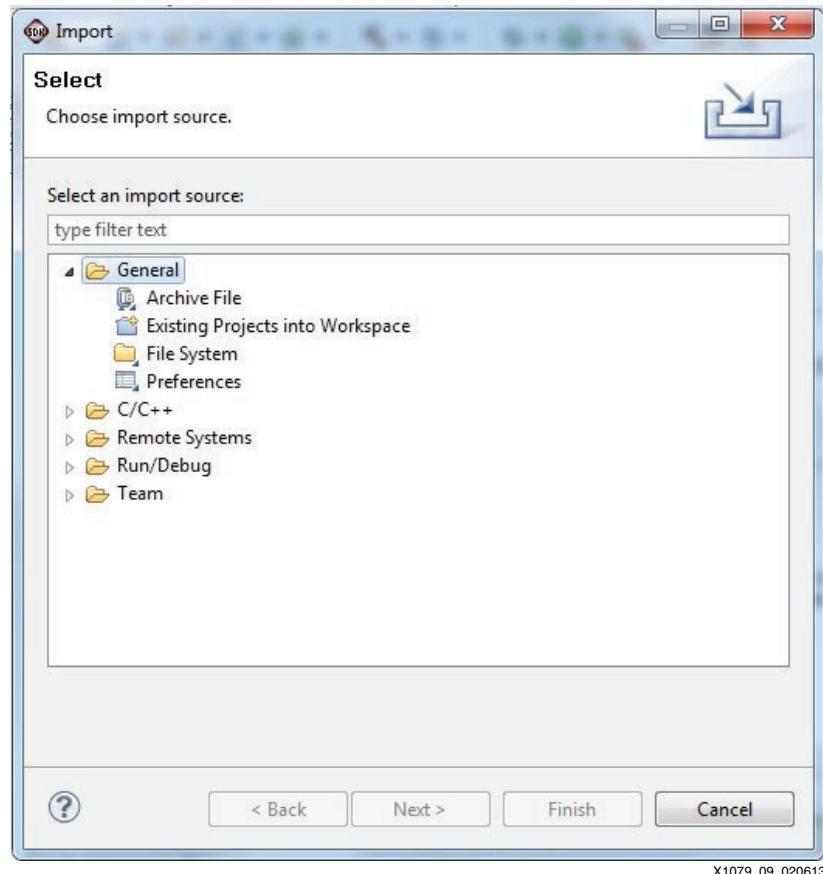


Figure 8: CPU0 Import

8. Expand **General** and select **File System**, as shown in [Figure 9](#).



*Figure 9: CPU0 General File System*

9. Click **Next**.

10. Browse to and select the `design\src\apps\app_cpu0` directory, as shown in Figure 10.

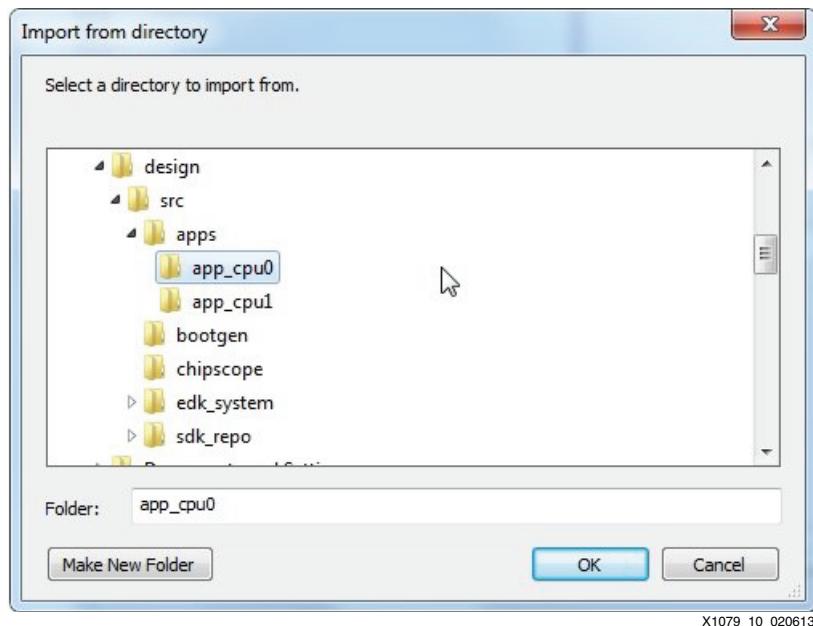
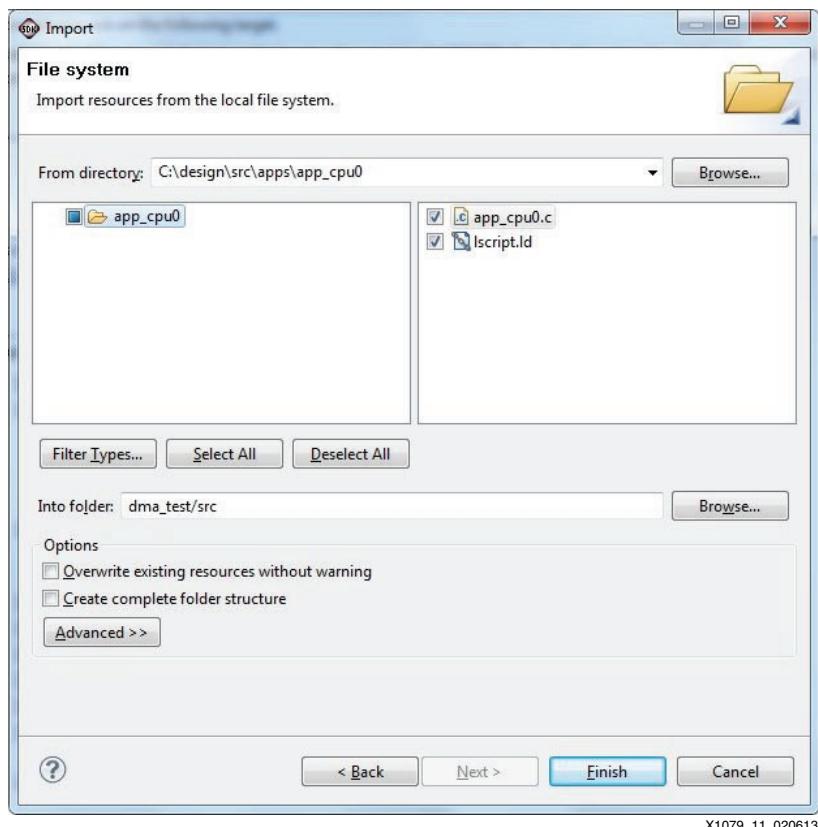


Figure 10: CPU0 Select Source Directory For Import

11. Click **OK**.  
12. In the left pane, click the `app_cpu0` folder but do not select the check box.

13. In the right pane select all files, as shown in [Figure 11](#).



X1079\_11\_020613

**Figure 11: CPU0 Select Files to Import**

14. Click **Finish**.

15. In the pop-up window, click **Yes** to overwrite **lscript.ld**.

After SDK finishes compiling the new application, the ELF is available at:

```
design\work\edk_system\SDK\Workspace\app_cpu0\Debug\app_cpu0.elf
```

### Create Bare-Metal Application For CPU1

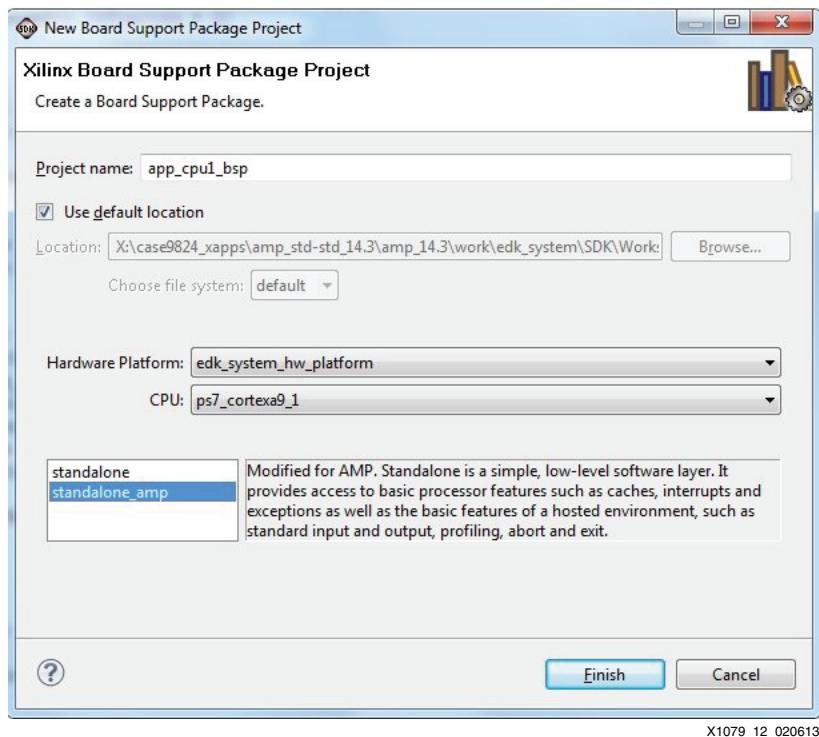
The instructions in this section describe how to create the application ELF that runs on CPU1 after the FSBL loads the applications to DDR memory. This step is slightly different than creating the application for CPU0 because CPU1 uses the customized BSP. An important aspect of this design is that it prevents CPU1 from accessing shared resources such as the ICD or SCU. SDK is used to create the BSP using the customized standalone BSP from the repository that is included with the design.

**Note:** This application has already been compiled and is available at  
design\generated\_files\SDK\_apps\app\_cpu1.elf.

Prepare to create the application:

1. Select **File > New > board\_support\_package**.
2. Change the Project Name to **app\_cpu1\_bsp**.
3. Change the CPU to **ps7\_cortexa9\_1**.

4. Change the Board Support Package Project to **standalone\_amp**, as shown in [Figure 12](#).



*Figure 12: CPU1 BSP*

5. Click **Finish**.
6. Select **Overview > Drivers > cpu\_cortexa9**.
7. Add **-DUSE\_AMP=1** to **extra\_compiler\_flags**, as shown in [Figure 13](#).

Name	Value	Default	Type	Description
compiler	arm-xilinx-eabi-gcc	arm-xilinx-eabi-gcc	string	Compiler
archiver	arm-xilinx-eabi-ar	arm-xilinx-eabi-ar	string	Archiver
compiler_flags	-O2 -c	-O2 -c	string	Compiler
extra_compiler_flags	-g -DUSE_AMP=1	-g	string	Extra con

*Figure 13: CPU1 BSP Add USE\_AMP*

8. Click **OK**.

To create the bare-metal application that runs on CPU1 and to import the included software:

1. Select **File > New > application\_project**.
2. Enter the project name as **app\_cpu1**.
3. Change processor to **ps7\_cortexa9\_1**.

4. Change board support package to **Use existing**.
5. Select **app\_cpu1\_bsp**, as shown in Figure 14.

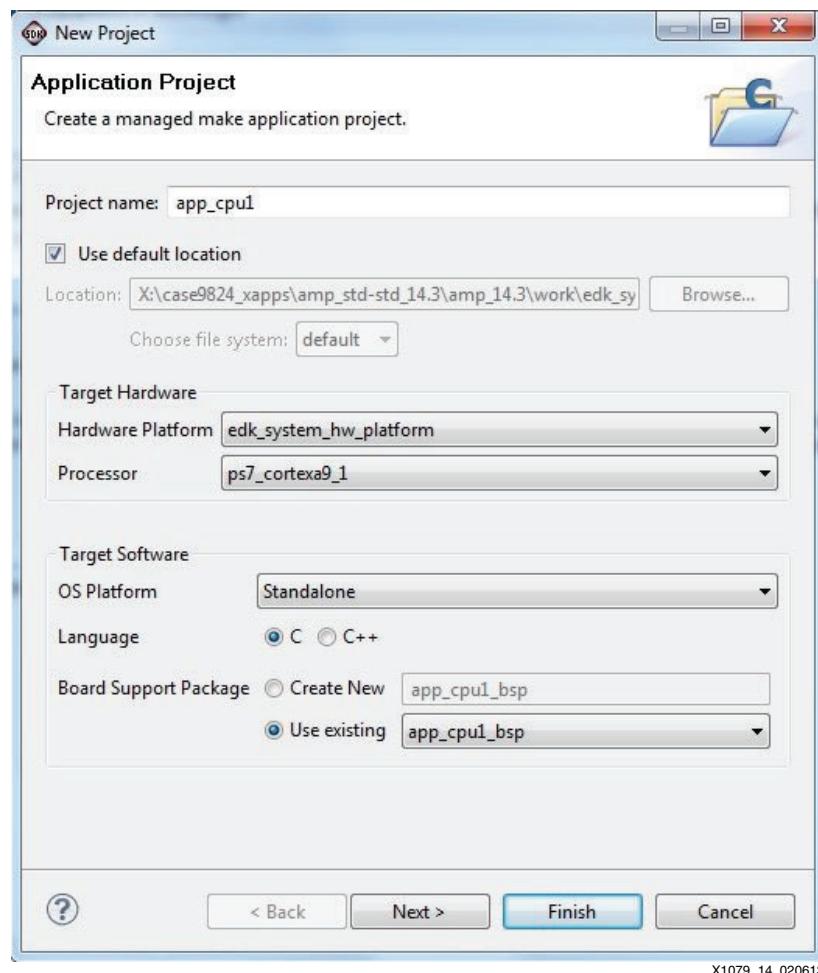
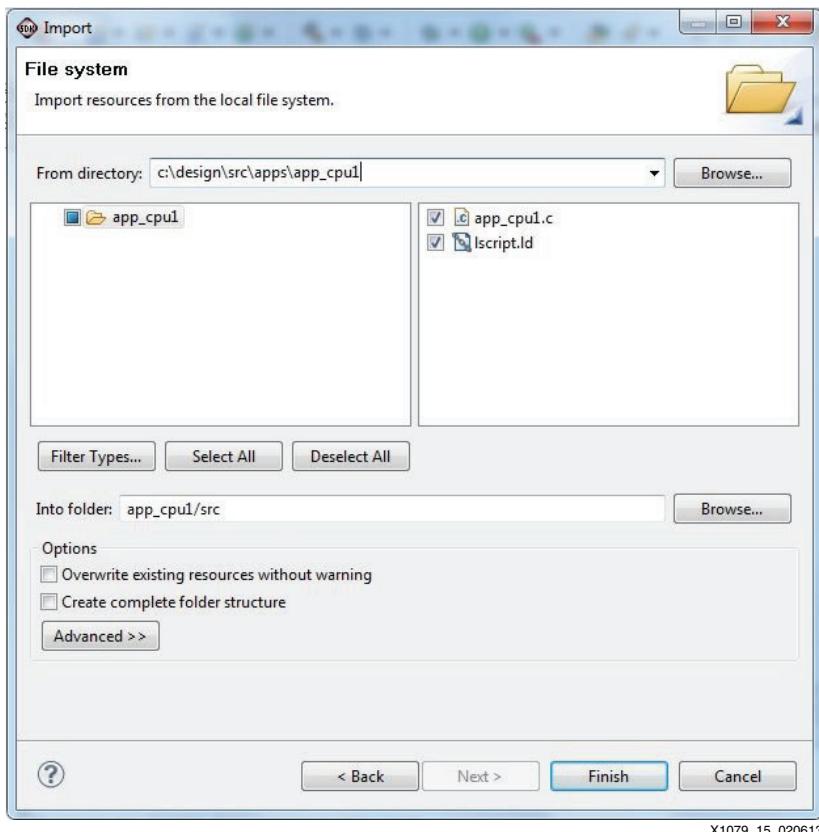


Figure 14: CPU1 Create Application

6. Click **Next**.
7. Select the **Empty Application** template.
8. Click **Finish**.
9. In the **Project Explorer** tab, expand **app\_cpu1**.
10. Right-click the **src** folder and select **Import** from the pop-up menu.
11. Select **General > File\_System**.
12. Click **Next**.
13. Browse to and select the **design\src\apps\app\_cpu1** directory.
14. In the left pane, select the **app\_cpu1** folder but do not add a check mark.

15. In the right pane select all files, as shown in [Figure 15](#).



*Figure 15: CPU1 Select Files to Import*

16. Click **Finish**.

17. In the pop-up window, click **Yes** to overwrite `lscript.ld`.

After SDK finishes compiling the new application, the ELF file is available at:

`design\work\edk_system\SDK\Workspace\app_cpu1\Debug\app_cpu1.elf`

## 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. In the example design, 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 will be 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 the following steps are precompiled and available at `design\generated_files`.

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

To generate the boot file:

1. Copy the included directory `design\src\bootgen` to `design\work\bootgen`.  
This directory includes the BootGen batch file (`createBoot.bat`), the `.bif` file (`bootimage.bif`), and a binary file (`cpu1_bootvec.bin`) that contains only the hexadecimal value `0xFFFFFFF0` (swapped for little endian is `0x00, 0xFF, 0xFF, 0xFF`). The FSBL recognizes the load address of this file as `0xFFFFFFFF0` (as it was configured in `bootimage.bif`) and triggers the FSBL to stop loading ELF or binary 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` to `design\work\bootgen`.  
**Note:** If the steps were not taken to compile the FSBL in SDK, a copy is provided in the included design at `design\generated_files\SDK_apps\amp_fsbl.elf`.
3. Copy the `design\work\edk_system\implementation\download.bit` file to `design\work\bootgen`.  
**Note:** If the steps were not taken to compile the FPGA bit file, a copy is provided in the included design at `design\generated_files\fpga\download.bit`.
4. Copy the generated bare-metal application for CPU0 from `design\work\edk_system\SDK\Workspace\app_cpu0\Debug\app_cpu0.elf` to `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\SDK_apps\app_cpu0.elf`.
5. Copy the generated bare-metal application for CPU1 from `design\work\edk_system\SDK\Workspace\app_cpu1\Debug\app_cpu1.elf` to `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\SDK_apps\app_cpu1.elf`.
6. Start an ISE Design Suite command prompt.  
This command prompt has the environment setup for the Xilinx tools.
7. From the command prompt, change directories to `design\work\bootgen`.
8. Run the `createBoot.bat` file.  
Running this file creates the `BOOT.BIN` file in the current (`bootgen`) directory.

## Copying Boot File to SD Card

Copy the `design\work\bootgen\BOOT.BIN` file to the SD card.

**Note:** If the previous steps were not taken to generate `BOOT.BIN`, a precompiled version is available at `design\generated_files\boot\BOOT.BIN`.

## 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 [\[Ref 2\]](#)

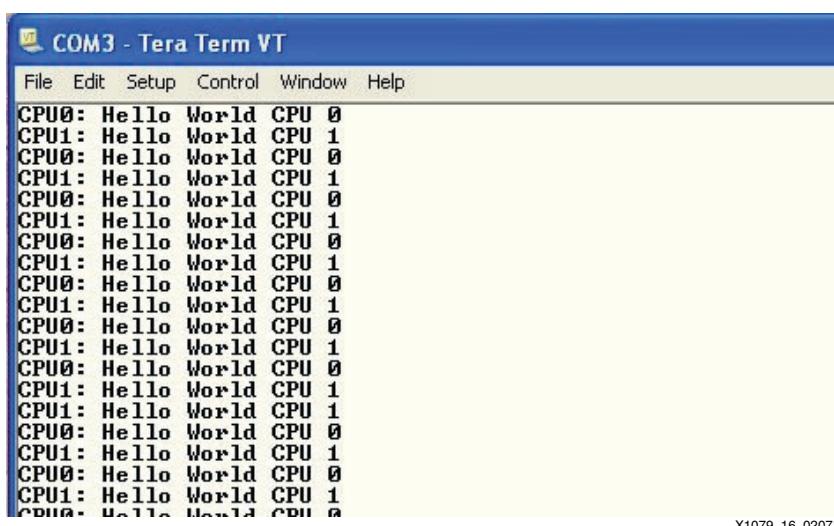
## **Hardware Setup**

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. A 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 from the SD card. The system can take up to 18 seconds before an output begins to appear on the UART. This UART is dependent upon a third-party driver. For more information, see *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.3) Getting Started Guide*.

If the files were created correctly, alternating output to the serial port between *CPU0: Hello World CPU 0* and *CPU1: Hello World CPU 1* is displayed, as shown in [Figure 16](#).



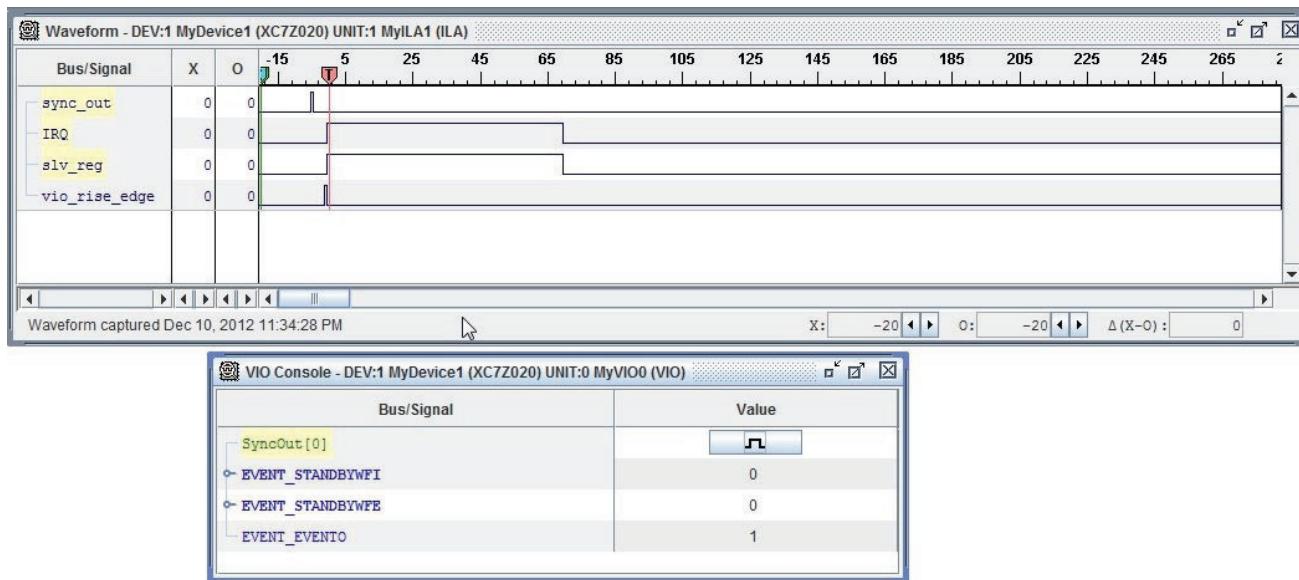
*Figure 16: Terminal Output*

During boot, the PS bootloader detects that the mode pins have been configured to boot from the SD card. The PS bootloader then 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, the CPU0 and CPU1 ELF files, and then the dummy file `cpu1_bootvec.bin`. At this point, the FSBL that is running on CPU0 jumps to the execution address of the first application that was loaded after the FSBL. As CPU0 starts to run `app_cpu0.elf`, it writes the starting address of the CPU1 application (`0x00200000`) to OCM at `0xFFFFFFFF0`, then executes the assembly instruction `SEV`, which sets an event that wakes up CPU1. Before the bootloader started running the FSBL, it created a small application at `0xFFFFFFFF00` and set the CPU1 program counter to this location. This application checks the contents of `0xFFFFFFFF0` and if it is 0, executes the Wait For Event (WFE) instruction. Every time an event occurs, CPU1 wakes up and reruns the loop where it checks `0xFFFFFFFF0` for a non-zero value. As soon as a non-zero value is detected, CPU1 jumps to the address location that was read from `0xFFFFFFFF0`. In this case, the value is `0x00200000`, which is the starting address of the CPU1 application as defined in the `lscript.ld` linkerscript for the `app_cpu1` application.

The ChipScope analyzer VIO core is used to generate interrupts towards CPU1. A ChipScope Integrated Logic Analyzer (ILA) core is also located in the design to monitor the IRQ signal.

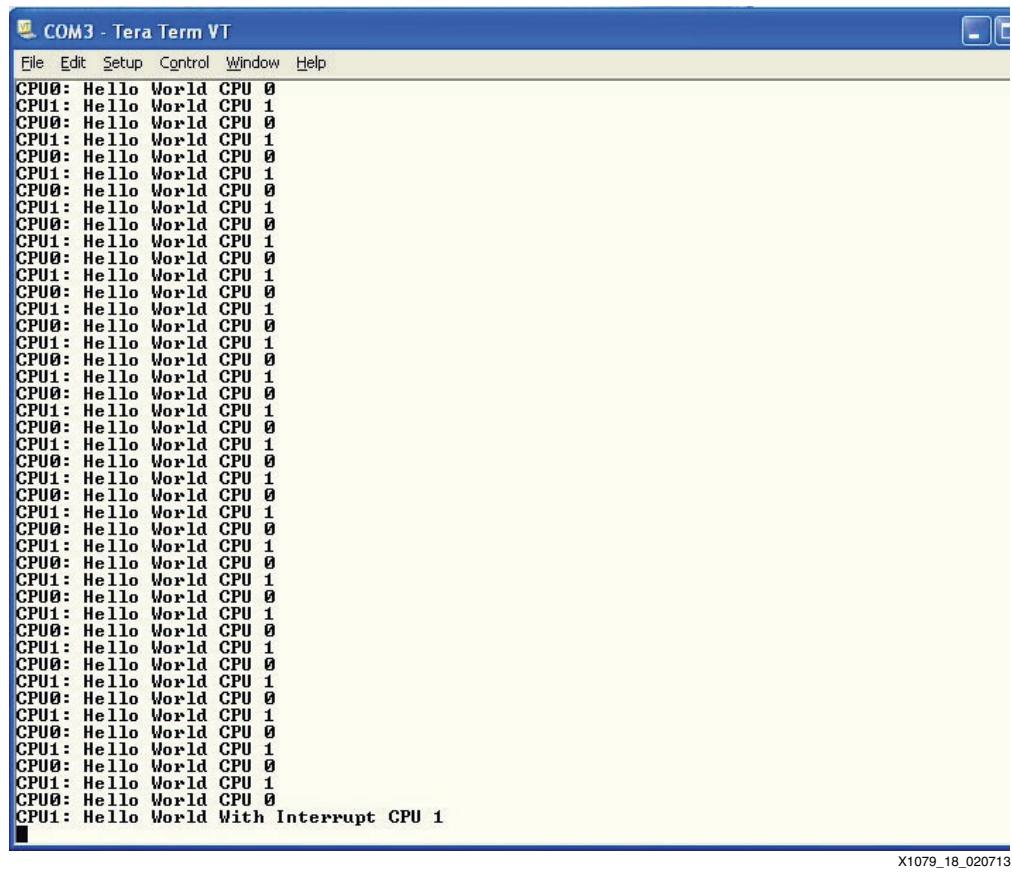
To use the ILA core to measure how long the IRQ signal is active (showing IRQ latency) and create interrupts using the ChipScope analyzer VIO console:

1. While the design is running, start ChipScope analyzer.
  2. Connect to the JTAG chain.
  3. Click **OK**.
  4. Open the existing ChipScope analyzer configuration file by selecting **File > open\_project**.
  5. Click **No** to saving changes.
  6. Browse to the Chipscope analyzer configuration file at:  
design\src\chipscope\csdefaultproj.cpj
- Note:** The ILA trigger is already set up to trigger when IRQ is High.
7. Select **UNIT:1 Trigger Setup** and arm the trigger.
  8. Select the ChipScope analyzer **VIO Console**.
  9. Click the **SyncOut[0]** button as shown in [Figure 17](#). [Figure 17](#) also shows the waveform for the interrupt generated after the SyncOut[0] button is pressed.



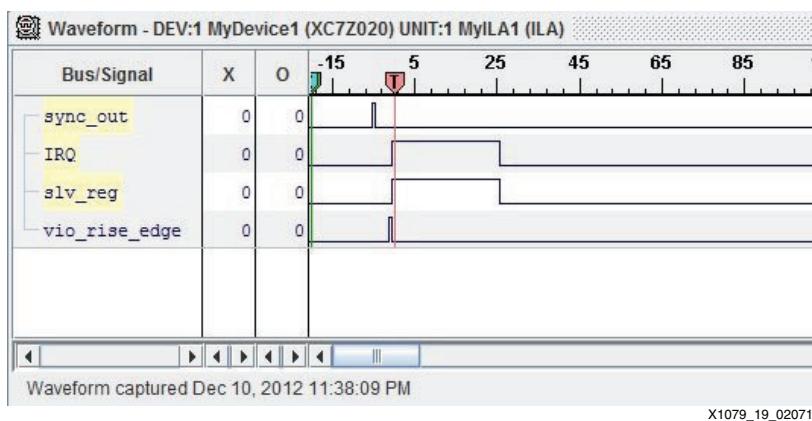
*Figure 17: ChipScope Analyzer Capture of First IRQ*

CPU1 services the interrupt, sets the global flag, and the CPU1 `main()` prints *CPU1: Hello World With Interrupt CPU 1*, as shown in [Figure 18](#).



*Figure 18: Console Output after ChipScope Analyzer Trigger*

Every time the virtual button is clicked, an interrupt is created and the CPU1 IRQ service routine sets a global variable that the CPU1 `main()` function uses to alter the print output from CPU1. In [Figure 19](#) it can be seen how subsequent IRQs have much lower interrupt latency due to the IRQ service routine being cached.



*Figure 19: ChipScope Analyzer Subsequent Capture*

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 the execution, the service routine finishes by clearing the interrupt signal that is being generated by the embedded core. In Figure 17, 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 memory refresh is occurring at the same time as the fetching of the service routine.

After the first IRQ occurs, the service routine is stored 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 19](#), the interrupt service completed after 25 clocks. This delay is approximately 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.

## Debugging the Design

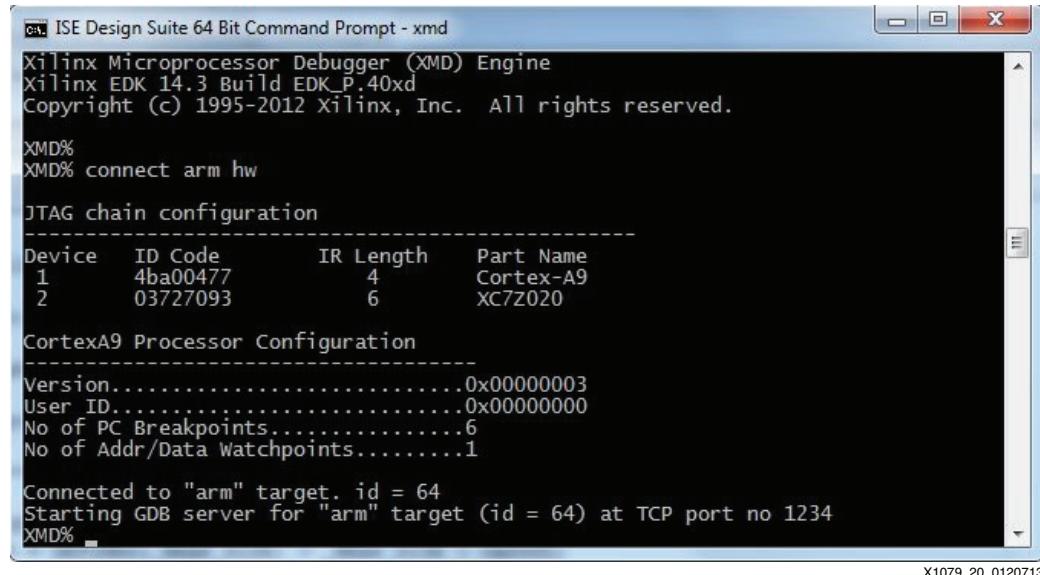
SDK can be used to connect and debug the applications running on both CPUs in tandem. XMD provides a command shell and GDB server that connects to the CPU by way of the JTAG cable. Normally, SDK automatically starts XMD in the background when starting to debug an application. For this example design, XMD is manually started to connect to both CPU0 and CPU1. Then, SDK is instructed to connect to each XMD GDB server during debug.

Because FSBL was used to boot the design, there is no need to reinitialize the PS registers. Care must be taken not to reset the full PS because both CPUs are to be debugged simultaneously.

To prepare to debug the design:

1. Connect the platform cable to the ZC702 board and ensure that the jumper options are configured for the correct debug cable.
2. In SDK, select **Xilinx\_Tools > Launch\_shell** to open a Xilinx command shell.
3. At the command shell prompt enter **xmd**.
4. At the XMD prompt enter the **connect arm hw** command.

XMD responds with the TCP port number 1234, as shown in [Figure 20](#).



```

ISE Design Suite 64 Bit Command Prompt - xmd
Xilinx Microprocessor Debugger (XMD) Engine
Xilinx EDK 14.3 Build EDK_P.40xd
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.

XMD%
XMD% connect arm hw
JTAG chain configuration
-----
Device ID Code IR Length Part Name
1 4ba00477 4 Cortex-A9
2 03727093 6 XC7Z020

CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....1

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD%

```

X1079\_20\_0120713

*Figure 20: Connect XMD to CPU0*

5. Enter the command **connect arm hw -debugdevice cpunr 2**.

XMD responds with the TCP port number 1235, as shown in [Figure 21](#).

```
ISE Design Suite 64 Bit Command Prompt - xmd
2      03727093      6      XC7Z020
CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....1

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD% connect arm hw -debugdevice cpunr 2

CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....1

Connected to "arm" target. id = 65
Starting GDB server for "arm" target (id = 65) at TCP port no 1235
XMD%
```

X1079\_21\_020713

*Figure 21: Connect SMD to CPU1*

Two GDB servers are now running and listening to TCP ports 1234 and 1235. As XMD connects, the CPU is halted and output to the terminal should stop.

To start debugging CPU0:

1. In an SDK Project Explorer window, right-click **app\_cpu0** and select **Debug As > debug\_configurations**, as shown in [Figure 22](#).

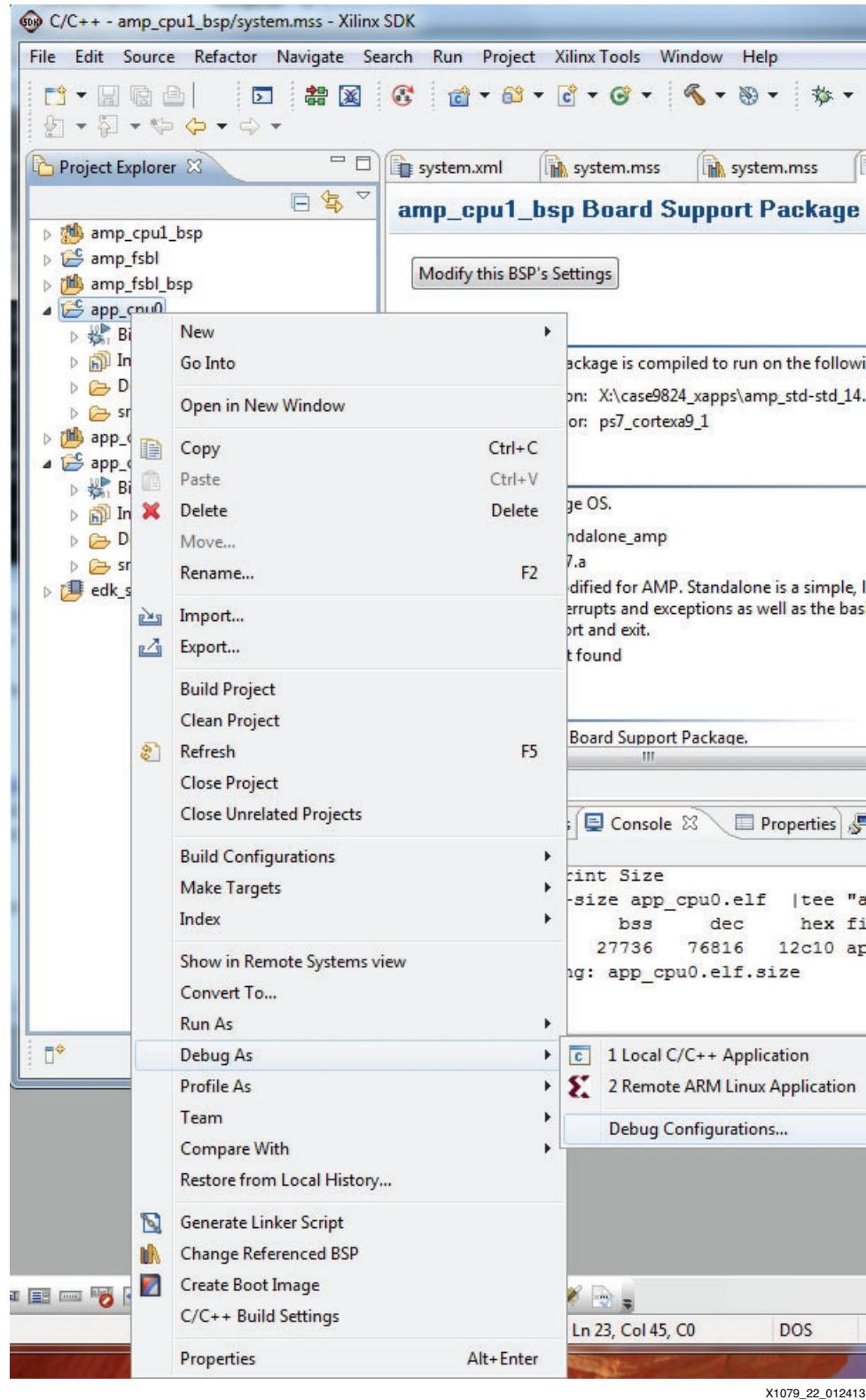
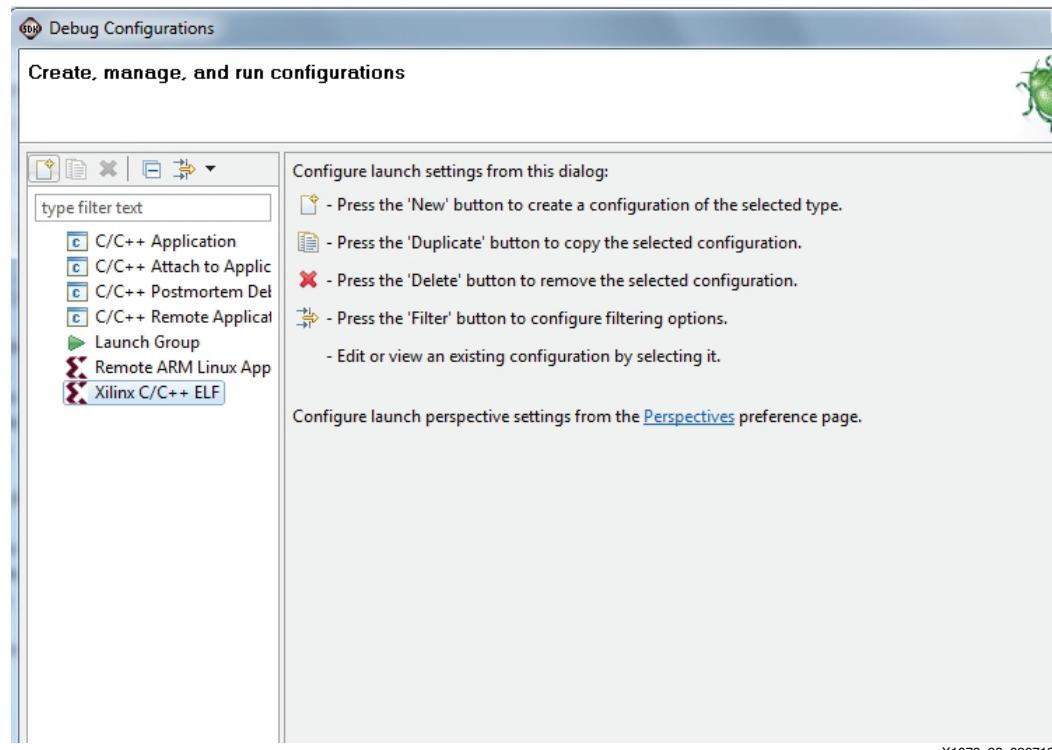


Figure 22: CPU0 Debug Configuration

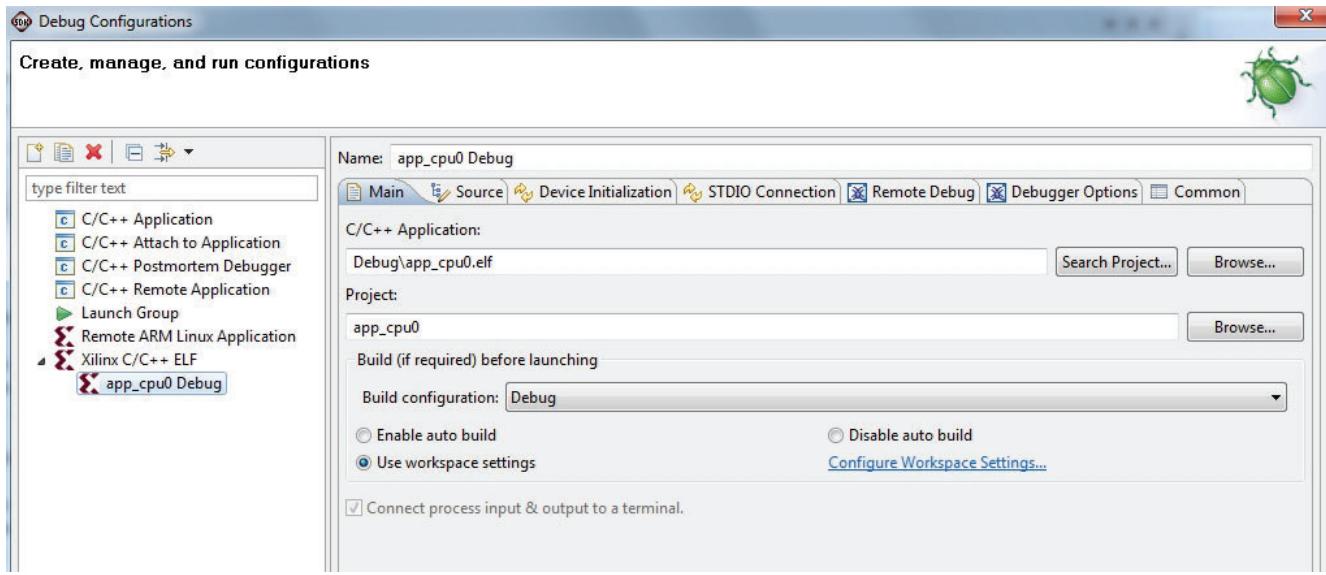
2. Select **Xilinx C/C++ ELF**, then click the **New launch configuration** icon at the top left, as shown in [Figure 23](#).



X1079\_23\_020713

**Figure 23: CPU0 Debug Configuration**

The name is automatically set to **app\_cpu0 Debug**, as shown in [Figure 24](#).

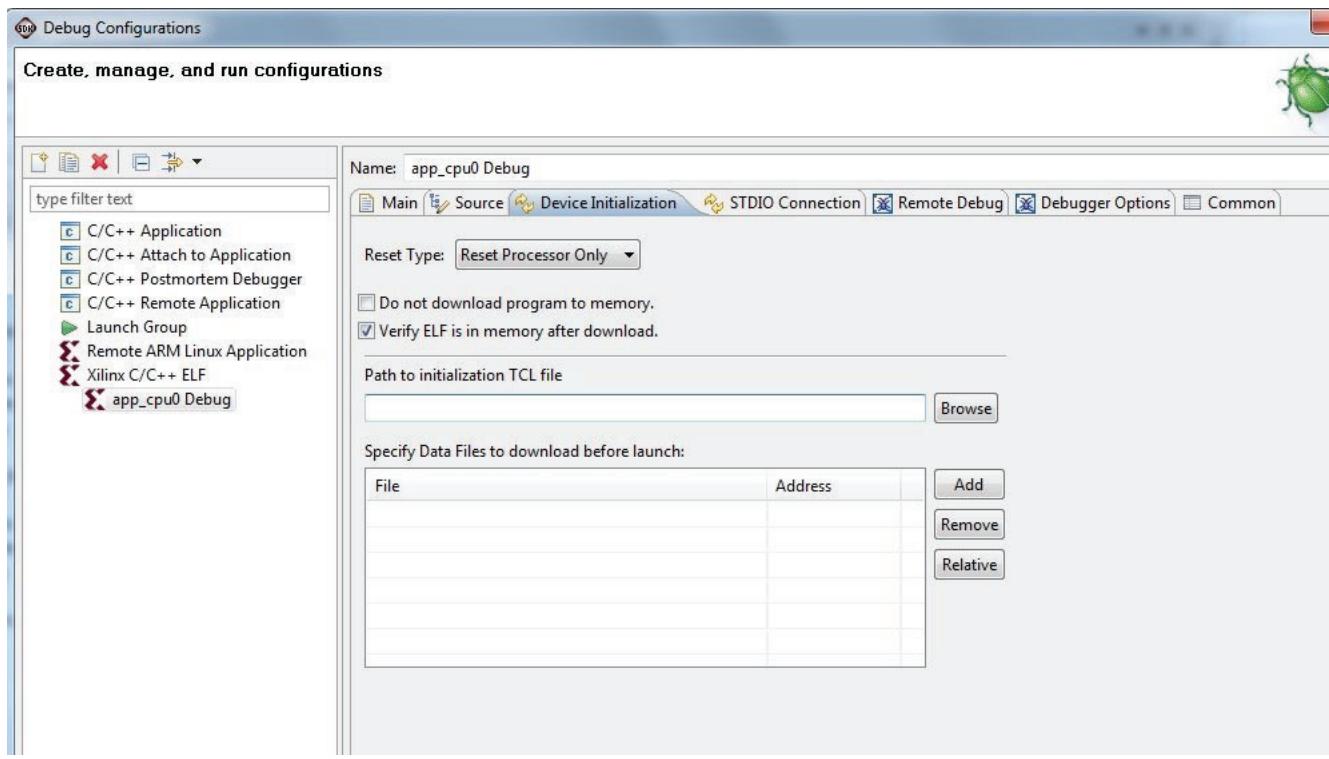


X1079\_24\_012413

**Figure 24: CPU0 Debug Configuration Name**

3. Click the **Device Initialization** tab.

4. Clear the **Path to initialization TCL file** field (initialization has already been done by the FSBL), as shown in [Figure 25](#).

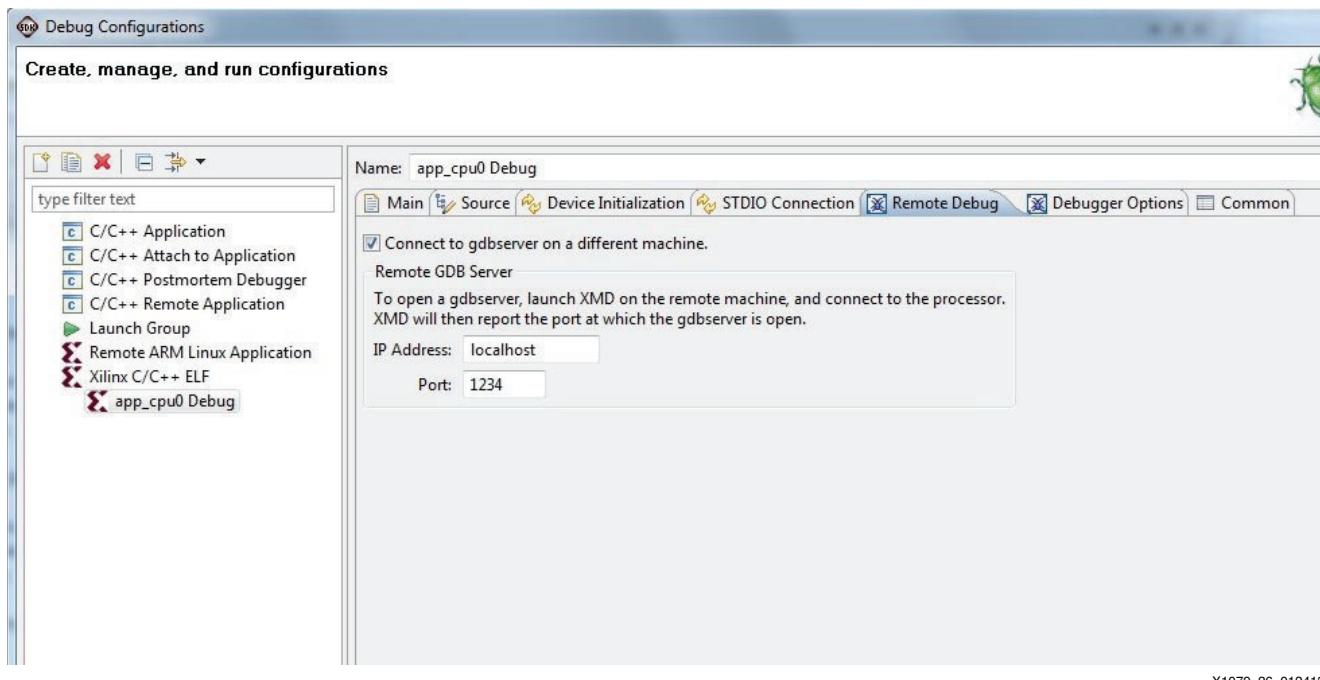


X1079\_25\_012413

*Figure 25: CPU0 Debug Initialization*

5. Click the **Remote Debug** tab.
6. Instruct SDK to connect to the externally created GDB server by selecting **Connect to gdbserver on a different machine**.

The IP address should default to **localhost** and the port should be **1234**, as shown in [Figure 26](#).



[Figure 26: CPU0 Remote Debug Configuration](#)

7. Click **Apply**.
8. Click **Debug**.
9. Click **Yes** to confirm the perspective switch.

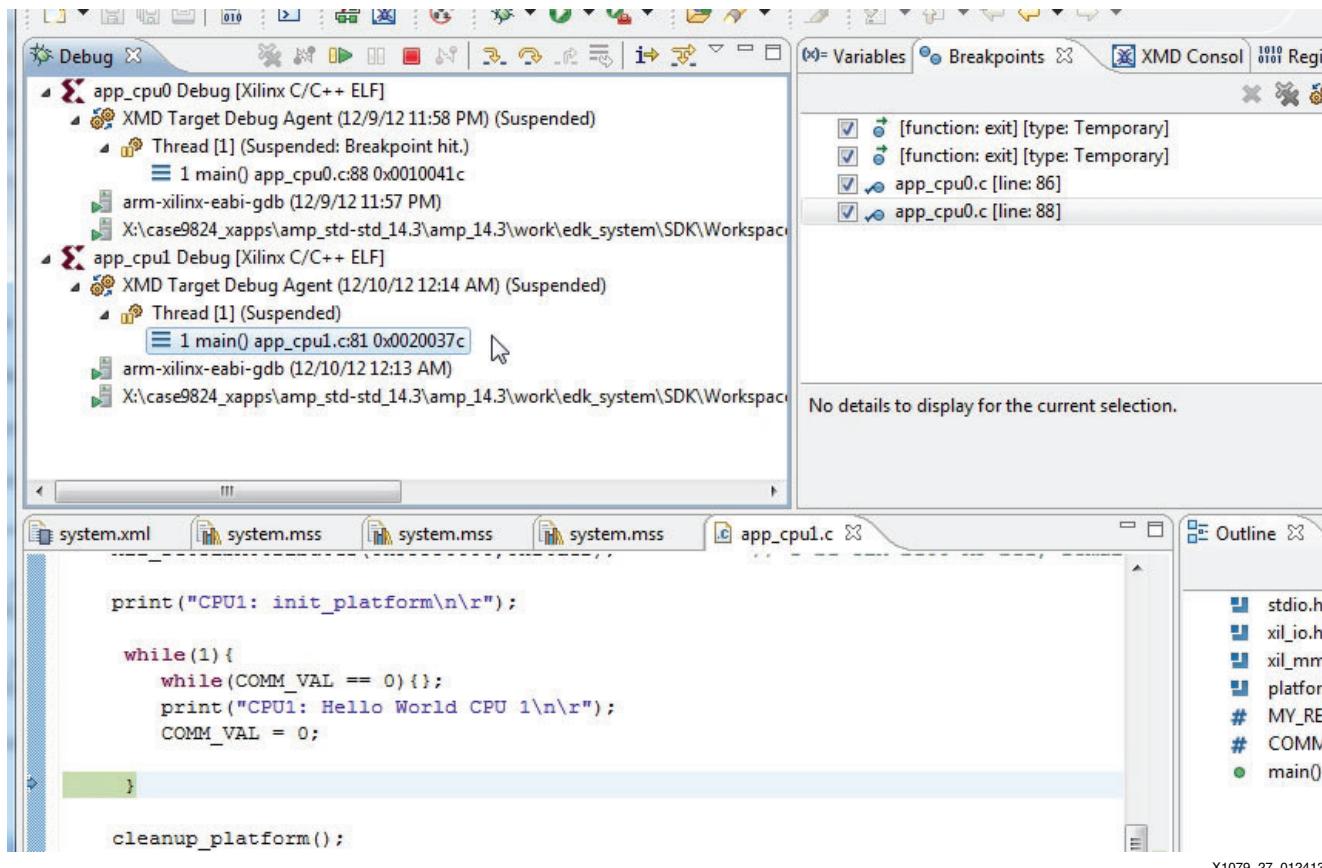
The application is downloaded then executed (the ELF download could have been disabled in the Device Initialization tab). The application stops at a breakpoint at the first executable line in the `main()` function. There are times when the application might not automatically stop at the beginning of `main()` so the **Pause** button (suspend) might have to be clicked. It is possible to restart the application on CPU0 by using SDK to manually set a breakpoint at the beginning of `main()`, changing the PC (program counter) to `0x00100000`, then clicking **Resume**.

10. Click the **Resume**, **Single Step**, or other debugging buttons to continue running the application.

While debugging CPU0, start debugging CPU1 by following these steps:

1. In the SDK window, select **C/C+ View**.
2. In the Project Explorer window, right-click **app\_cpu1** and select **Debug As > debug\_configurations**.
3. Select **Xilinx C/C++ ELF**, then click the **New launch configuration** icon at the top left. The name is automatically set to **app\_cpu1 Debug**.
4. Click the **Device Initialization** tab.
5. Clear the **Path to initialization TCL file** field (initialization has already been done by CPU0 and the FSBL).
6. Click the **Remote Debug** tab.
7. Instruct SDK to connect to the externally created GDB server by selecting **Connect to gdbserver on a different machine**.

- The IP address should default to **localhost** and the port should be **1235**.
8. Click **Apply**.
  9. Click **Debug**.
- The application is downloaded, then executed. The application stops at a breakpoint at the first executable line in `main()`.
10. Click the **Resume**, **Single Step**, or other debugging buttons to continue running the application.
- At any point within the Debug view, the focus can be switched between CPU0 and CPU1 debug by selecting the listed function under **Thread**. As each function is selected, the visible source changes, as shown in [Figure 27](#).



*Figure 27: Debug View*

## Conclusion

The example design demonstrates how to boot the Zynq-7000 AP SoC and start two Cortex-A9 processors, each running their own 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 the bare-metal application 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, Xilinx 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
04/03/2013	1.0	Initial Xilinx release.
01/24/2014	1.0.1	Minor typographical corrections.

---

## 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.