|  |
| --- |
|  |
| Simple AMP: Bare-metal Running on both Zynq CPUs |
|  |
|  |
| **John McDougall** |
| **12/4/2012** |

|  |
| --- |
|  |

Table of Contents

[Summary 4](#_Toc345883486)

[Included Systems 4](#_Toc345883487)

[Introduction 4](#_Toc345883488)

[Objective 4](#_Toc345883489)

[Design Overview 5](#_Toc345883490)

[Hardware 7](#_Toc345883491)

[Address Map 7](#_Toc345883492)

[Software 8](#_Toc345883493)

[FSBL 8](#_Toc345883494)

[Bare-Metal Operating System 8](#_Toc345883495)

[CPU0 Application 9](#_Toc345883496)

[CPU1 Application 9](#_Toc345883497)

[Inter-processor Communication 10](#_Toc345883498)

[Reference Design 10](#_Toc345883499)

[Implementation Details 10](#_Toc345883500)

[Generating the Hardware 11](#_Toc345883501)

[Implement the embedded design and export to SDK 11](#_Toc345883502)

[Generating the Applications 12](#_Toc345883503)

[Configure SDK 12](#_Toc345883504)

[Create Custom FSBL Application 13](#_Toc345883505)

[Create Bare-Metal Application For CPU0 14](#_Toc345883506)

[Create Bare-Metal Application For CPU1 19](#_Toc345883507)

[Generating the Boot File 22](#_Toc345883508)

[Copy the Files to SD Card 23](#_Toc345883509)

[Running the Design 24](#_Toc345883510)

[Debugging the Design 27](#_Toc345883511)

[List of References 33](#_Toc345883512)

List of Figures

[Figure 1: PL Block Diagram 7](#_Toc345883513)

[Figure 2: Select Workspace Directory 11](#_Toc345883514)

[Figure 3: Select Repository 12](#_Toc345883515)

[Figure 4: Create FSBL 13](#_Toc345883516)

[Figure 5: Select Custom FSBL Template 14](#_Toc345883517)

[Figure 6: Create app\_cpu0 15](#_Toc345883518)

[Figure 7: CPU0 Empty Application 16](#_Toc345883519)

[Figure 8: CPU0 Import 17](#_Toc345883520)

[Figure 9: CPU0 General Filesystem 18](#_Toc345883521)

[Figure 10: CPU0 Select Source Directory For Import 18](#_Toc345883522)

[Figure 11: CPU0 Select Files to Import 19](#_Toc345883523)

[Figure 12: CPU1 BSP 20](#_Toc345883524)

[Figure 13: CPU1 BSP Add USE\_AMP 20](#_Toc345883525)

[Figure 14: CPU1 Create Application 21](#_Toc345883526)

[Figure 15: CPU1 Select Files to Import 22](#_Toc345883527)

[Figure 16: Terminal Output 24](#_Toc345883528)

[Figure 17: Chipscope Capture of First IRQ 25](#_Toc345883529)

[Figure 18: Console Console Output after Chipscope Trigger 26](#_Toc345883530)

[Figure 19: Chipscope Subsequent Capture 27](#_Toc345883531)

[Figure 20: Connect XMD to CPU0 28](#_Toc345883532)

[Figure 21: Connect SMD to CPU1 28](#_Toc345883533)

[Figure 22: CPU0 Debug Configuration 29](#_Toc345883534)

[Figure 23: CPU0 Debug Configuration 30](#_Toc345883535)

[Figure 24: CPU0 Debug Configuration Name 30](#_Toc345883536)

[Figure 25: CPU0 Debug Initialization 31](#_Toc345883537)

[Figure 26: CPU0 Remote Debug Configuration 31](#_Toc345883538)

[Figure 27: Debug View 33](#_Toc345883539)

List of Tables

[Table 1: IRQ\_GEN Control Register 8](#_Toc345883540)

Simple AMP with bare-metal running on both Zynq CPUs

# Summary

Zynq contains two Cortex-A9 processors that can be configured to concurrently run their own operating systems. This application note describes a method/solution to start up both CPUs, each running its own operating system and application, and communicate with each other through shared memory.

# Included Systems

The design is created and built using the 14.3 version of Xilinx Platform Studio (XPS). The design also includes software build using the Xilinx Software Development Kit (SDK).

The complete set of project files are provided with this application note to allow the designer to examine and rebuild this design or use the files as a template for starting a new design.

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

# Introduction

Zynq 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, and applications, with the possibility to loosely couple the applications via shared resources.

This application note includes both hardware and software necessary to build a reference design that runs both Zynq Cortex-A9 processors in an AMP configuration. Each CPU is running within its own standalone environment. The provided example design prevents the CPUs from colliding with shared hardware resources.

The document also describes how to create a bootable solution and how to debug both CPUs.

# Objective

The objective of this document:

* Create bare-metal applications that run on each CPU
* Use the provided, modified First Stage Boot Loader (FSBL) to boot the system
* Use the provided, modified bare-metal Board Support Package (BSP) to avoid collision of shared hardware resources
* Configure and access a shared common Processing System (PS) peripherals and On Chip Memory (OCM) block
* Interact with peripherals in the Programmable Logic (PL)
* Debug application code on each CPU simultaneously
* Service an interrupt directly from the PL on CPU1
* Use Chipscope to measure the latency of the interrupt

# Design Overview

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

In this AMP example, the bare-metal application, running on CPU0, will be the master of the system and will be responsible for:

1. System initialization
2. Controlling CPU1’s startup
3. Communicating with CPU1
4. Sharing the UART with CPU1

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

1. Communicating with CPU0
2. Servicing interrupts from a core in the PL
3. Sharing the UART with CPU0

The Zynq PS includes resources that are private to each CPU and includes resources that are shared by both CPUs. Running in an AMP configuration, care must be taken to prevent both CPUs from contending for these shared resources. Refer to the TRM for further information on shared vs private resources.

An example of some of the private resources is:

1. L1 Cache
2. Private Peripheral Interrupts (PPI)
3. MMU
4. Private Timers

Examples of some of the shared resources are:

1. Interrupt Control Distributor (ICD)
2. DDR
3. OCM
4. Global Timer
5. SCU and L2 Cache
6. 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. In order to keep the complexity of this XAPP 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 in order to communicate to each other. By disabling Cache accesses to OCM from both processors OCM provides a very low latency and deterministic method for inter processor communications.

Below is a list of actions taken by this design in order to prevent problems with the shared resources:

1. DDR: CPU0 has only been made aware of memory at 0x00100000 to 0x001FFFFF. CPU1 uses memory from 0x00200000 to 0x002FFFFF 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
5. OCM: Accesses to OCM is handled very carefully by each CPU in order 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 will set the flag and only CPU1 will clear the flag.

For demonstration purposes only, a custom embedded core is included in this design example in order to provide a simple interrupt source. By using a Chipscope Virtual I/O core, for stimulus to this core, the user is able to generate interrupts towards the PS at their leisure. Using the Chipscope VIO core provides more control of when an interrupt occurs and therefore makes it easier to measure the latency of interrupts. In a regular design, this core would not exist and instead, the interrupt would be sourced by a different piece of logic in the PL such as a DMA engine.

## Hardware

The PL contains a custom, embedded core connected to a synchronous output of a Chipscope Virtual Input/Output (VIO) core. The VIO core provides a mechanism for a user to interact with hardware from the Chipscope Analyzer application.

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 IRQ during the interrupt service routine. CPU0 can optionally use the control register in order to create an interrupt towards CPU1. The Core1\_nIRQ pin connects directly to CPU1’s Private Peripheral Interrupt (PPI) block so there is no need to modify the configuration of the shared Interrupt Control Distributor (ICD). A Chipscope AXI Monitor core is also included and allows you to measure the latency of the IRQ getting serviced.



Figure : PL Block Diagram

## 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).

Control Register definition:

|  |  |  |
| --- | --- | --- |
| 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. |

Table : IRQ\_GEN Control Register

## Software

The software can be broken down into three sections:

* The FSBL boot loader
* The bare-metal operating system and application for CPU0
* The bare-metal operating system and application for CPU1

### FSBL

The FSBL always runs on CPU0 and is the first software application that is ran after power on reset of the PS. The FSBL is responsible for programming the PL and copies both application ELF files to DDR memory. After loading the applications to DDR, the FSBL will then start executing the first application that was loaded.

The current version of FSBL, that is included in the 14.3 version of ISE, 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 will write it to the PL. Next, whether a bit file is found or not, the FSBL will load one application ELF into memory and execute it.

For this AMP example, 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 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 TRM.

### Bare-Metal Operating System

This design has both CPU0 and CPU1 running bare-metal. CPU0 is responsible for initializing shared resources and starting up CPU1.

The current bare-metal Board Support Package (BSP), named standalone\_v3\_07\_a, includes support for the preprocessor define constant ‘USE\_AMP’. This constant prevents the BSP from re-initializing the PS Snoop Control Unit (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, with this XAPP, to remove the re-mapping when ‘USE\_AMP’ is set.

### CPU0 Application

CPU0’s application does the following:

1. CPU0 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 address 0xFFFF0000-0xFFFFFFFF. Only the high 64KB of OCM is used by this XAPP so cache is disabled on address 0xFFFF0000-0xFFFFFFFF.
2. CPU0 initializes the Interrupt Distributer (ICD)
3. Startup CPU1
4. CPU0 prints to the UART
5. CPU0 sets a semaphore flag in OCM
6. CPU0 waits for the semaphore flag to be cleared
7. CPU0 continues steps 3 to 5 forever

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 bare-metal) starts CPU1 (running bare-metal) by writing the value of 0x00200000 to address 0xFFFFFFF0 and then running the command SEV (Set Event). The SEV will cause CPU1 to wake up, read the value 0x00200000 from address 0xFFFFFFF0 and then jump to address 0x00200000. The FSBL was responsible for placing CPU1’s ELF at 0x00200000.

### CPU1 Application

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

CPU1’s application does the following:

1. CPU1 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 address 0xFFFF0000-0xFFFFFFFF. Only the high 64KB of OCM is used by this XAPP so cache is disabled on address 0xFFFF0000-0xFFFFFFFF.
2. CPU1 initializes the PPI interrupt controller and interrupt subsystem
3. CPU1 waits for a semaphore flag in OCM to be set
4. CPU1 prints to the UART. The string printed will be chosen dependant on whether the interrupt service routine incremented a global variable or not. If the global variable ‘irq\_count’ was not zero, CPU1 will set the value to zero.
5. CPU1 clears the semaphore flag in OCM
6. CPU1 continues steps 3 to 5 forever

### Inter-processor Communication

The inter-processor communication in this design example is a simple flag or a Semaphore. When the flag is set CPU 1 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 is always coherent between the CPUs.

If DDR memory was used for the semaphore, both cpu’s would have to flush cache in order to force coherency or disable both L1 and L2 cache for an area of DDR. Another reason why not to use DDR is because DDR has a higher latency for accesses and is less deterministic due to background refresh cycles. DDR accesses are also bursting in nature so time would be wasted as a write or read burst occurs in order to access a single 32bit value.

# Reference Design

The following files are included in the reference design:

* XPS project
* SDK source files for CPU0 and CPU1 applications
* Generated files including:
  + 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

# Implementation Details

Extract the design files to a directory called ‘design’. Once extracted, create a new directory ‘design\work’. Copy the following:

‘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 creating the hardware design. If you want to skip the Hardware generation part go to section Generating the Applications. The pre-compiled design is available at ‘design\generated\_files\fpga\download.bit’.

### Implement the embedded design and export to SDK

1. Start Xilinx Platform Studio (XPS) and open the embedded project at ‘design\work\edk\_system\system.xmp’
2. Select device\_configuration->update\_bitstream. Once complete, 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 button to ‘Export & Launch SDK’. At this point, XPS will export 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 will open a dialog box asking where the workspace is located. Browse to, and select the directory ‘design\work\edk\_system\SDK’, click OK, and then before clicking OK the second time, add to the end of the selection ‘\Workspace’ as shown in Figure 2: Select Workspace Directory. SDK will automatically create the ‘Workspace’ subdirectory.

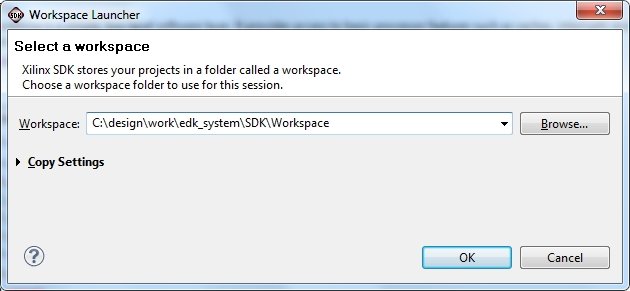


Figure : Select Workspace Directory

## Generating the Applications

### Configure SDK

The standalone BSP files (used by the bare-metal application) and modified FSBL files have been included in this XAPP. In order 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’. Note that this step is not necessary if XPS was used to ‘Export & Launch SDK’.
2. Point SDK to the included repository that contains the modified standalone BSP and modified FSBL.
   1. Select Xilinx\_tools->repositories
   2. Select ‘New’ for Local Repositories
   3. Browse to, and select the directory ‘design\src\sdk\_repo’
   4. Select OK

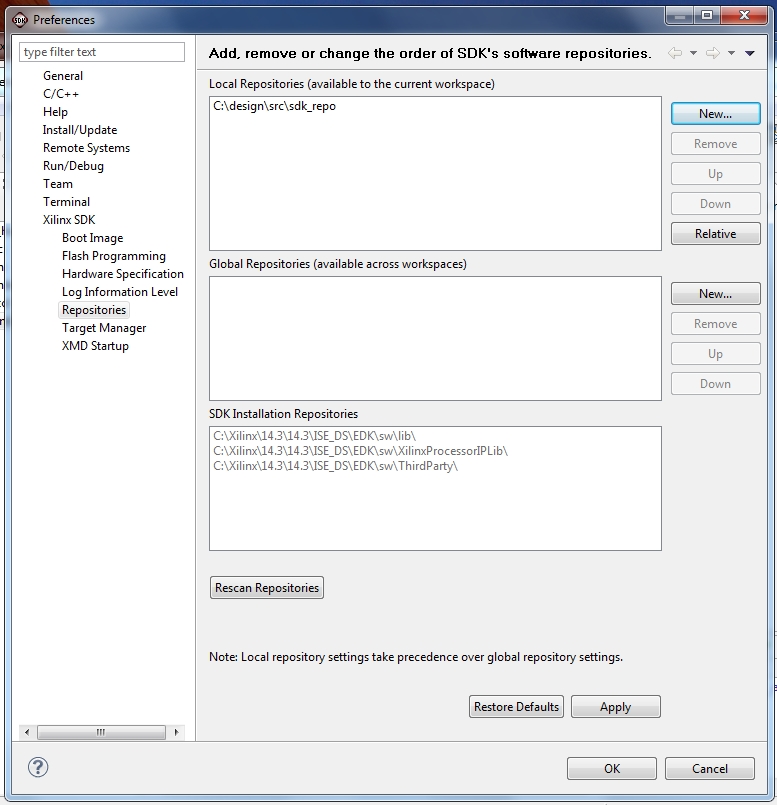


Figure : Select Repository

### Create Custom FSBL Application

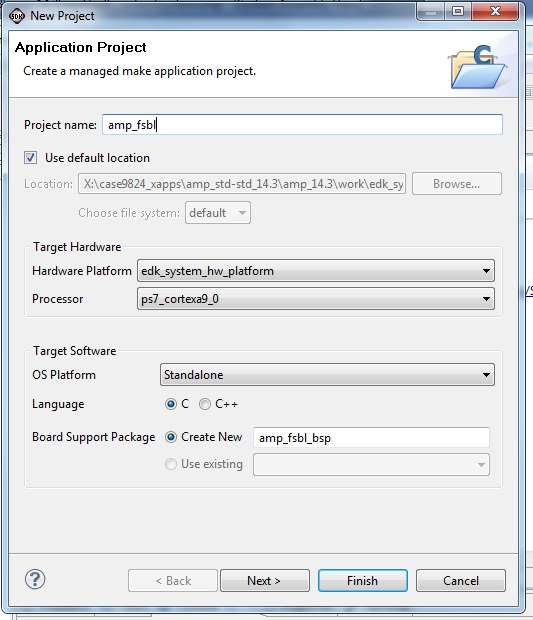
1. Make sure you have already run the Configure SDK section.
2. Create the FSBL using the new template from the repository.
   1. Select file->new->Application\_Project
   2. Set the Project Name to ‘amp\_fsbl’  
      

Figure : Create FSBL

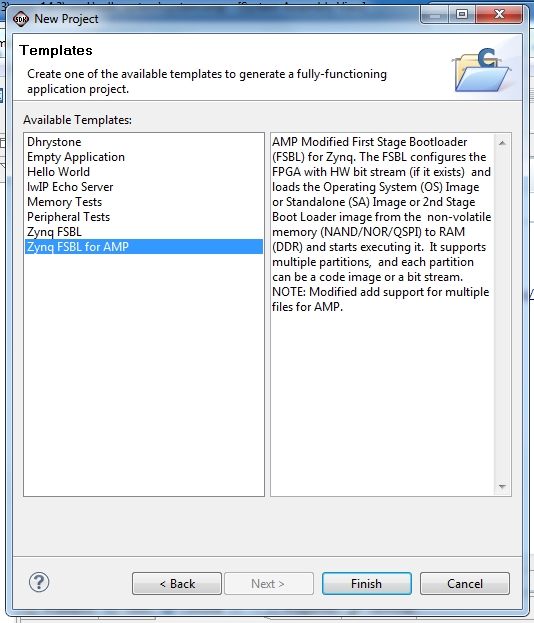
* 1. Click Next
  2. Select the available template ‘Zynq FSBL for AMP’. Note: If this template is not available in the list, verify that the repository is setup correctly.  
     

Figure : Select Custom FSBL Template

* 1. Click Finish

1. Once 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 precompiled version is also available at: ‘design\generated\_files\SDK\_apps\amp\_fsbl.elf’

### Create Bare-Metal Application For CPU0

The following instructions create the application ELF that will run on CPU0 after the FSBL copies the application ELFs to DDR.

Note that this application has already been compiled and is available at ‘design\generated\_files\SDK\_apps\app\_cpu0.elf’.

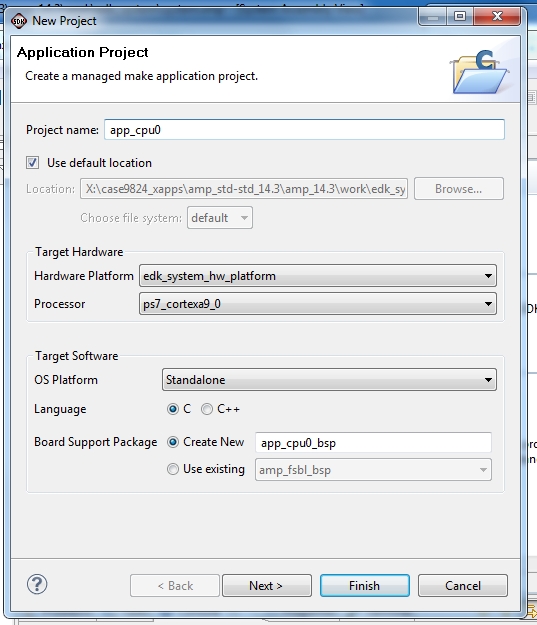
1. Within SDK, create the bare-metal application that will be running on CPU0 and import the included software.
   1. Select file->new->application\_project
   2. change the project name to ‘app\_cpu0’   
      

Figure : Create app\_cpu0

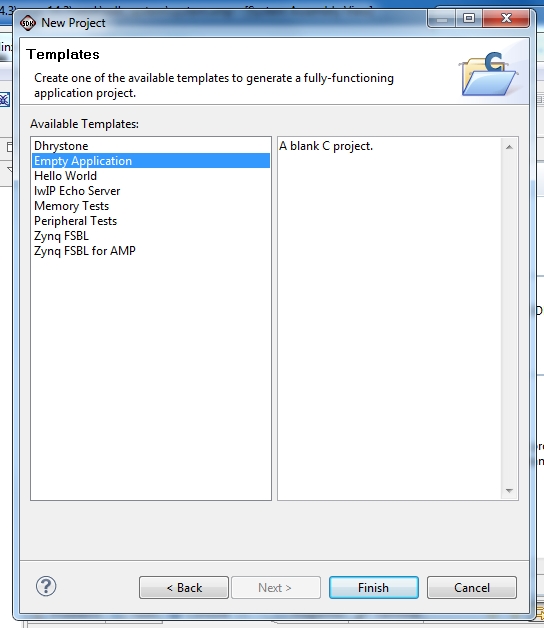
* 1. Click ‘next’.
  2. Choose the ‘Empty Application’ template  
     

Figure : CPU0 Empty Application

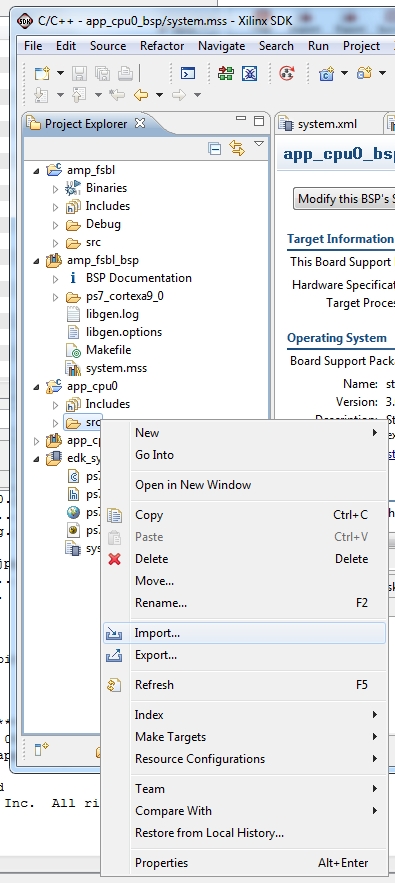
* 1. Select Finish
  2. In SDK’s Project Explorer tab, expand app\_cpu0 and right click on the ‘src’ folder.
  3. Select Import  
     

Figure : CPU0 Import

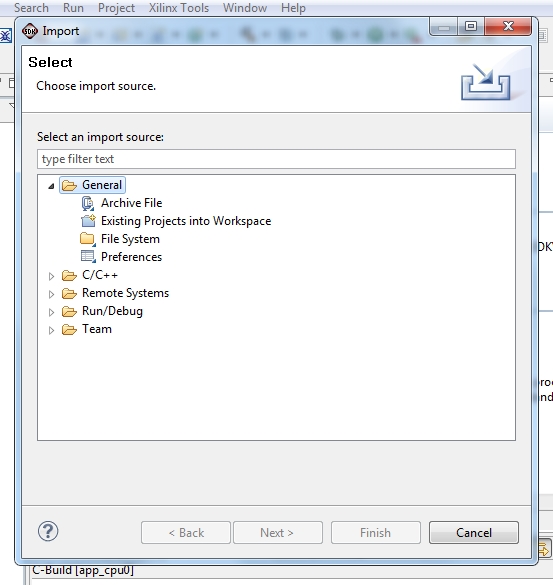
* 1. choose General->File\_System  
     

Figure : CPU0 General Filesystem

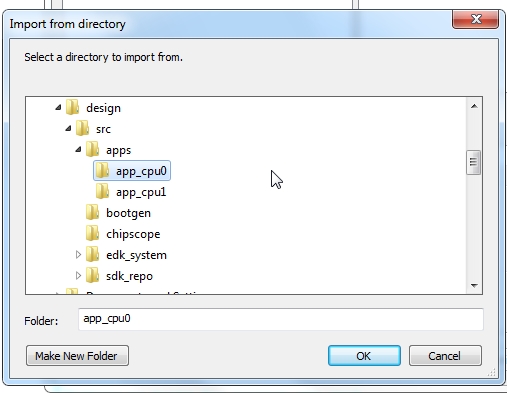
* 1. Click Next
  2. Browse to and select the included directory ‘design\src\apps\app\_cpu0’  
     

Figure : CPU0 Select Source Directory For Import

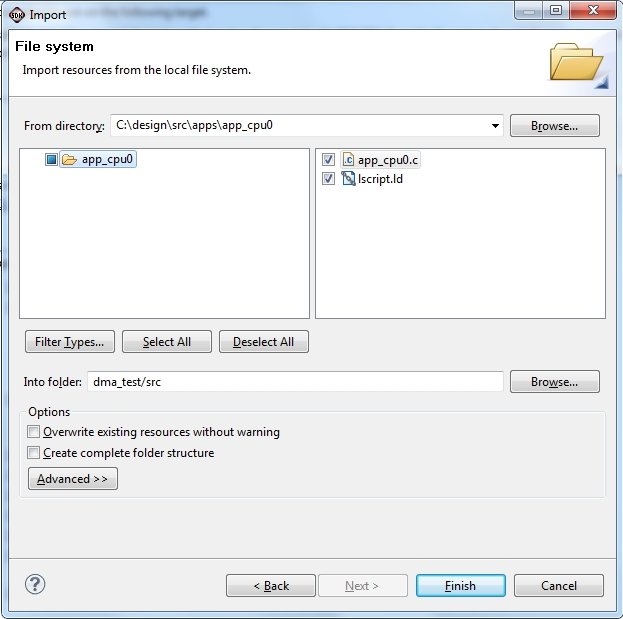
* 1. Click OK
  2. In the Left window pane choose the app\_cpu0 folder but do not add a checkmark. In the right window pane, select all files.   
     

Figure : CPU0 Select Files to Import

* 1. Click Finish
  2. Select ‘yes’ to overwrite lscript.ld

1. Once SDK completes 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 following instructions create the application ELF that will run on CPU1 after the FSBL loads the applications to DDR. This step is slightly different than creating the application for CPU0 because CPU1 will be using the customized BSP. Note that this design takes care to prevent CPU1 from accessing shared resources such as the ICD or SCU.

Note that this application has already been compiled and is available at ‘design\generated\_files\SDK\_apps\app\_cpu1.elf’.

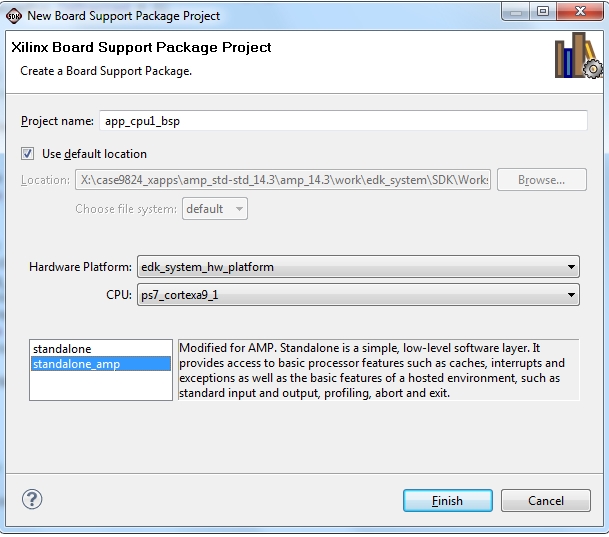
1. Within SDK, create the BSP using the customized standalone BSP from the repository that was included with the design.
   1. Select file->new->board\_support\_package
   2. Change the project name to ‘app\_cpu1\_bsp’
   3. Change the CPU to ‘ps7\_cortexa9\_1’
   4. Change the ‘Board Support Package OS’ to standalone\_amp  
      

Figure : CPU1 BSP

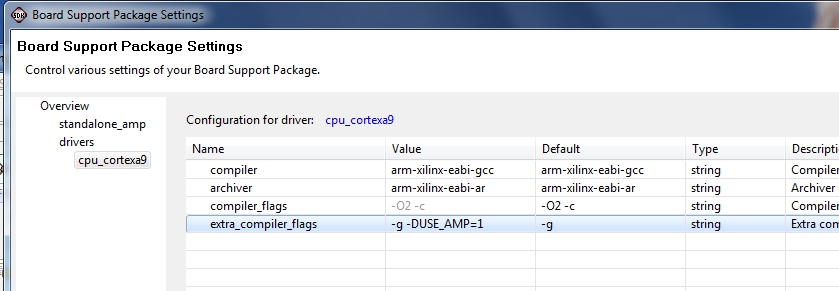
* 1. Click Finish
  2. Select Overview->drivers->cpu\_cortexa9 and add ‘ –DUSE\_AMP=1’ to extra\_compiler\_flags  
     

Figure : CPU1 BSP Add USE\_AMP

* 1. Select OK

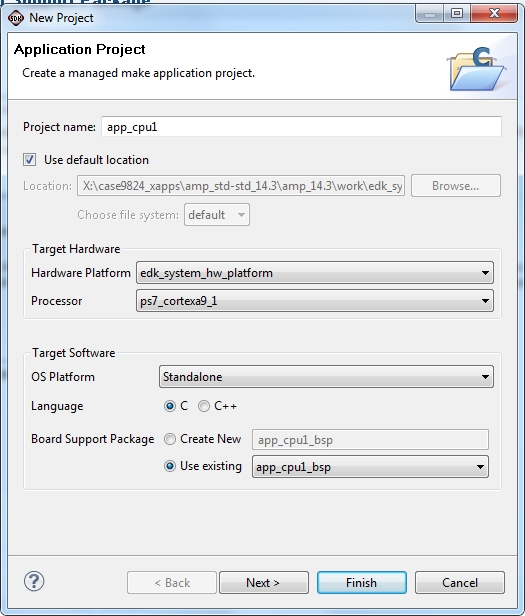
1. Create the bare-metal application that will be running on CPU1 and import the included software
   1. Select file->new->application\_project
   2. Enter the project name ‘app\_cpu1’
   3. Change processor to ps7\_cortexa9\_1
   4. Change Board Support Package to ‘Use existing’ and select app\_cpu1\_bsp  
      

Figure : CPU1 Create Application

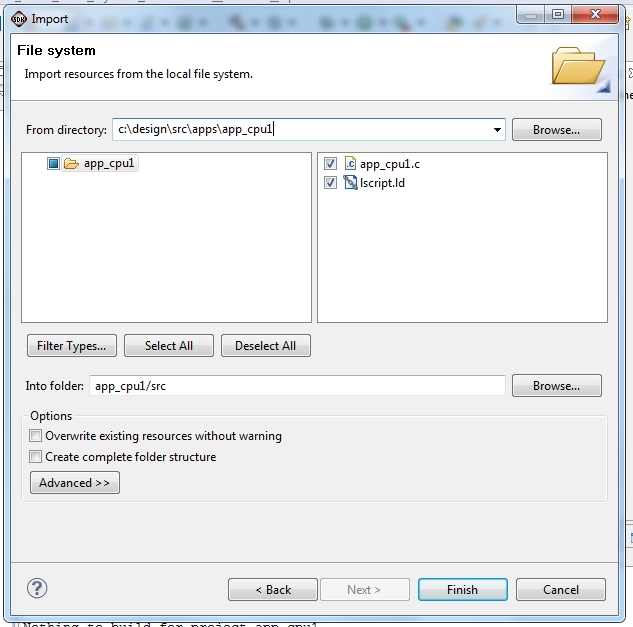
* 1. Select Next
  2. Choose the ‘Empty Application’ template
  3. Select Finish
  4. In SDK’s Project Explorer tab, expand ‘app\_cpu1’ and right click on the ‘src’ folder.
  5. Select Import, choose General->File\_System
  6. Click Next
  7. Browse to and select the included directory ‘design\src\apps\app\_cpu1’
  8. In the Left window pane choose the app\_cpu1 folder but do not add a checkmark. In the right window pane, select all files.   
     

Figure : CPU1 Select Files to Import

* 1. Click Finish
  2. Select ‘yes’ to overwrite lscript.ld

1. Once SDK completes compiling the new application, the ELF is available at ‘design\work\edk\_system\SDK\Workspace\app\_cpu1\Debug\app\_cpu1.elf‘.

## Generating the Boot File

The boot file, BOOT.BIN, normally contains the FSBL, FPGA bit file, and the ELF for the application that will run on CPU0. In this example, the FSBL has been modified to download more than one application so the second application ELF, that will be running on CPU1, is included in BOOT.BIN.

The design files contain a batch file, bootgen configuration file and a bin file (that is used to trigger FSBL to stop downloading further data or ELF files).

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

1. FSBL elf
2. CPU0 application
3. CPU1 application
4. Dummy 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 pre-compiled 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, .bif file and binary file that just contains the hex value 0xFFFFFF00 (swapped for little endian is 0x00, 0xFF, 0xFF, 0xFF). The FSBL will recognize this file’s load address of 0xFFFFFFF0, as configured in bootimage.bif, and will trigger 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 included 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 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’ 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\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’ 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\SDK\_apps\app\_cpu1.elf’
6. Open a Xilinx ‘ISE Design Suite Command Prompt’. This command prompt will have the environment setup for the Xilinx tools
7. In the command prompt, change the directory to ’design\work\bootgen’
8. Run ‘createBoot.bat’ file. This will create the boot file BOOT.BIN in the current directory

## Copy the Files to SD Card

Copy the following files to the SD Card:

1. ’design\work\bootgen\BOOT.BIN’  
   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

Configure the ZC702 demo board to boot from the SD card. Refer to CTT guide for further details. Configure a terminal program to listen to the correct COM port with a baud rate of 115200.

Power up the design and after the board boots, CPU0 starts running, then CPU0 instructs CPU1 to start running. Booting from the SD card could up to 18sec before output will appear on the UART. This UART is dependent upon a third party driver. Refer to the ZC702 Getting Started Guide for further details.

If the files were created correctly, alternating output to the serial port between ‘CPU0: Hello World CPU 0’ and ‘CPU1: Hello World CPU 1’ will be displayed.

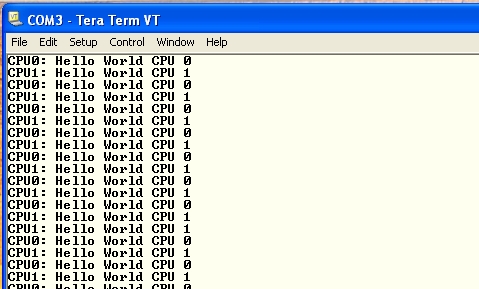


Figure : Terminal Output

During boot, the PS Bootloader will detect that the mode pins have been configured to boot from the SD card. In turn, the PS bootloader will open the BOOT.BIN file and search for the block of data that’s been flagged with ‘bootloader’. As seen in the bootimage.bif file, amp\_fsbl.elf has this flag. The bootloader will load this file into DDR and start running it. In turn, the FSBL will load the bit file, CPU0’s ELF, CPU1’s ELF, and then the dummy file cpu1\_bootvec.bin. At this point, the FSBL that’s running on CPU0 will jump to the execution address of the first application that was loaded after the FSBL. As CPU0 starts to run app\_cpu0.elf, it will write the starting address of CPU1’s application (0x00200000) to OCM at 0xFFFFFFF0 then execute the assembly instruction SEV which sets an event that wakes up CPU1. Before the PS bootloader started running the FSBL it created a small application at 0xFFFFFF00 and set CPU1’s program counter to this location. This application checks the contents of 0xFFFFFFF0 and if 0, will execute the WFE (wait for event) instruction. Every time an event occurs, CPU1 wakes up, and reruns the loop where it checks 0xFFFFFFF0 for a non-zero value. As soon as a non-zero value is detected, CPU1 will jump to the address location that was read from 0xFFFFFFF0. In this case, the value is 0x00200000 which is the starting address of CPU1’s application as defined in the linkerscript, lscript.ld, for the app\_cpu1 application.

The Chipscope VIO core is used to generate interrupts towards CPU1. A Chipscope ILA core is also located in the design to monitor the IRQ signal. The following steps will use the ILA to measure how long the IRQ signal is active (showing IRQ latency) and create interrupts using the VIO console.

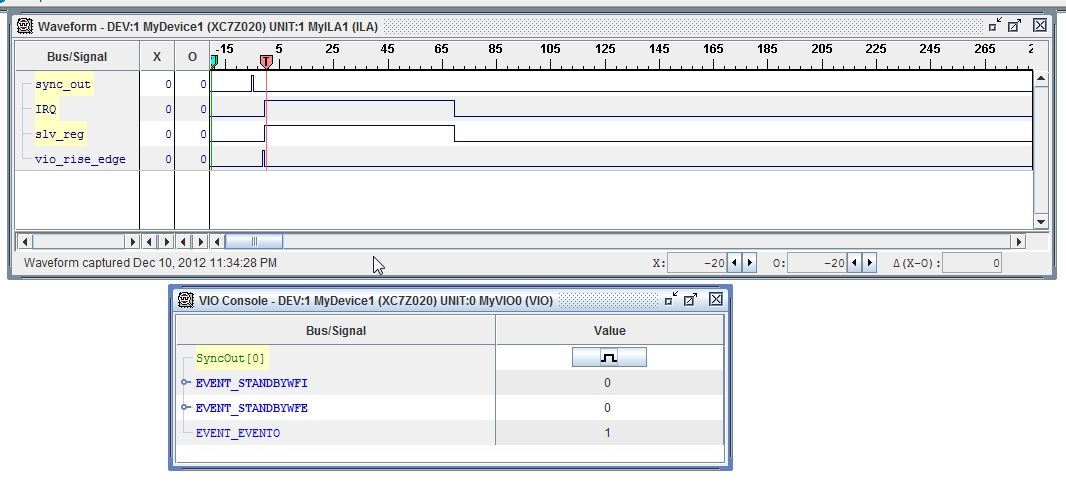
1. While the design is running, start Chipscope Analyzer
2. Connect to the JTAG chain. Chipscope Analyzer will display the two devices (ARM\_DAP and XC7Z020) that are in the chain. Click OK
3. Open the provided Chipscope configuration file: file->open\_project, click No to saving changes, and browse to the Chipscope configuration file at ‘design\src\chipscope\csdefaultproj.cpj’.  
   NOTE: The ILA trigger is already setup to trigger when IRQ is high
4. Select UNIT:1 Trigger Setup and arm the trigger
5. Select the VIO Console and you should see a pulse button called SyncOut[0]. Push the button and CPU1 will service the interrupt, set the global flag, and CPU1’s main() will print ‘CPU1: Hello World With Interrupt CPU 1’.  
   

Figure : Chipscope Capture of First IRQ

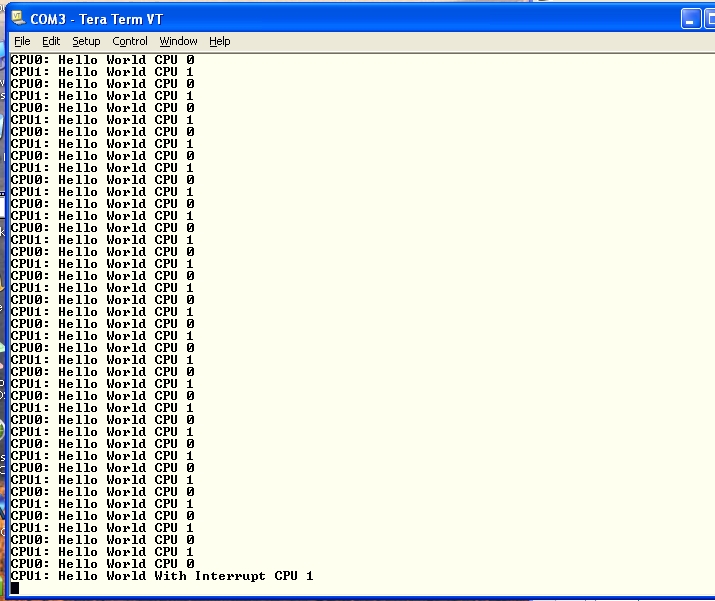


Figure : Console Console Output after Chipscope Trigger

Note: Every time the virtual button is pressed, an interrupt is created and CPU1’s IRQ service routine sets a global variable that CPU1’s main function uses to alter the print output from CPU1. In the following figure, it can be seen how subsequent IRQs have much lower interrupt latency due to the IRQ service routing being cached.

The bare-metal application that services the interrupt is located in DDR. When the first interrupt occurs, CPU1 will be instructed to jump to the service routine. This jump will cause the instructions, located in DDR, to be read into cache and executed. During the execution, the service routine will finish by clearing the interrupt signal that is being generated by the embedded core. In Figure 17, there was 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 will be located in cache so fetches of the instructions for the routine will be sourced by the cache instead of the slower, less deterministic DDR memory. As seen in Figure 19, the interrupt service was 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.

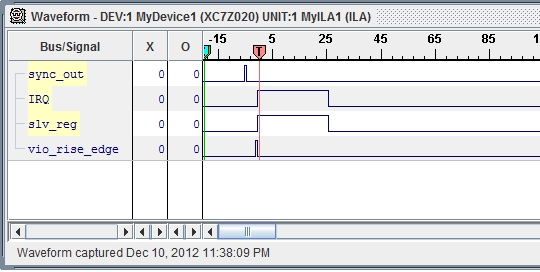


Figure : Chipscope Subsequent Capture

## 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 which connects to the CPU via the JTAG cable. Normally, SDK will automatically start XMD in the background when starting to debug an application. For this example design, XMD will be manually started in order to connect to both CPU0 and CPU1. Then, SDK will be instructed to connect to each XMD GDB server during debug.

Since FSBL was used to boot the design, there is no need to re-initialize the PS registers and care must be taken not to reset the full PS since both CPUs will be debugged simultaneously.

Connect the Platform Cable to the ZC702 and make sure the jumper options are configured for the correct debug cable.

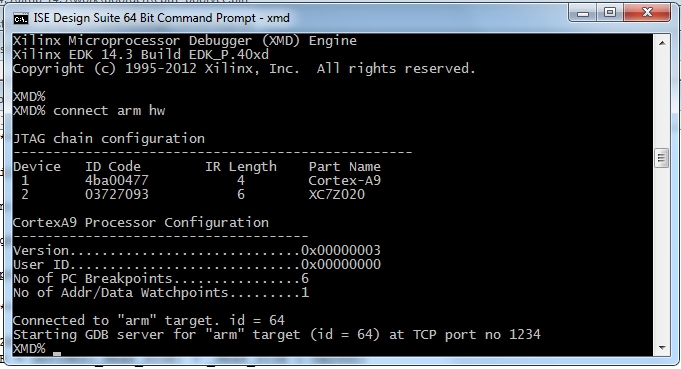
1. From SDK, start XMD and connect to both CPUs
   1. In SDK, open a Xilinx command shell with Xilinx\_Tools->Launch\_shell
   2. In the new command shell enter ‘xmd’
   3. At the XMD prompt, enter the command ‘connect arm hw’
   4. XMD should respond with the TCP port number 1234  
      

Figure : Connect XMD to CPU0

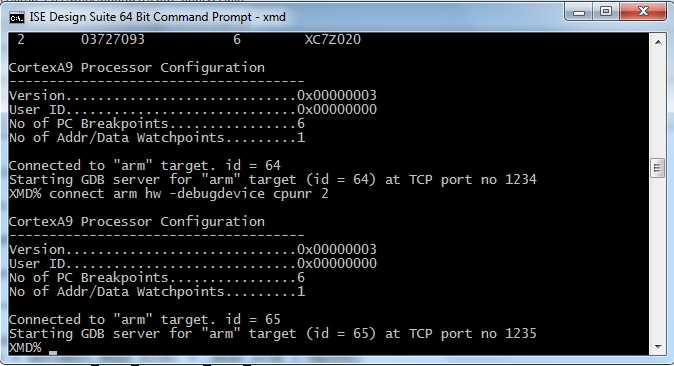
* 1. Enter the command ‘connect arm hw –debugdevice cpunr 2’
  2. XMD should respond with the TCP port number 1235  
     

Figure : Connect SMD to CPU1

Now, two GDB servers are running and listening to TCP ports 1234 and 1235. As XMD connects, the CPU will be halted and output to the terminal should stop.

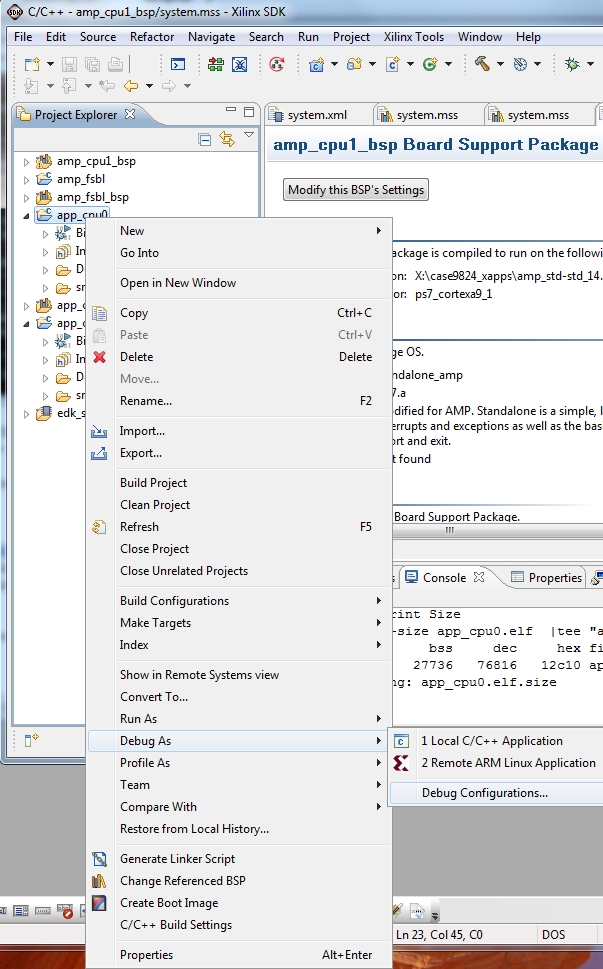
1. Start Debugging CPU0 in SDK
2. In SDK project explorer window, right click on app\_cpu0 and select debug\_as->debug\_configurations  
   

Figure : CPU0 Debug Configuration

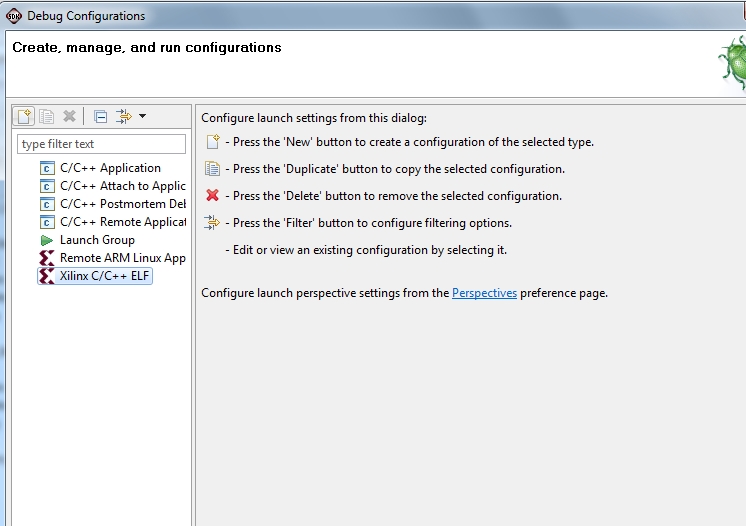
1. Highlight ‘Xilinx C/C++ ELF’ and select the ‘New launch configuration’ icon at the top left  
   

Figure : CPU0 Debug Configuration

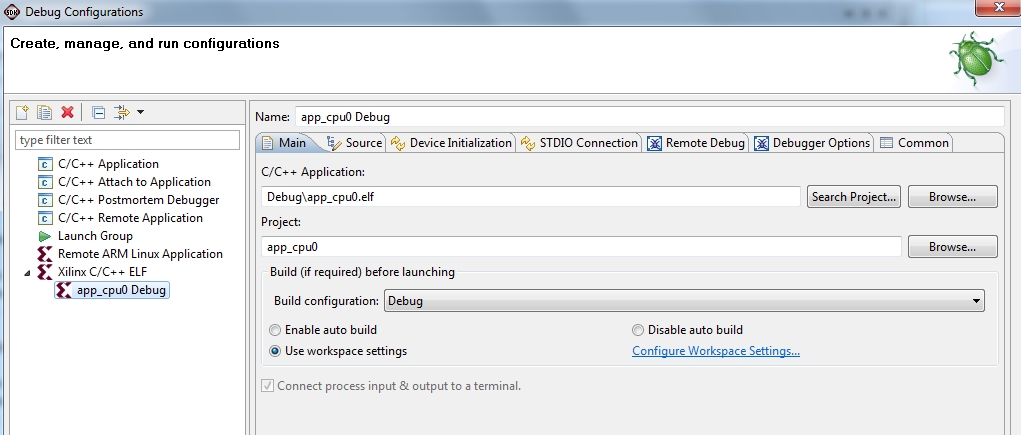
1. The name will be automatically set to ‘app\_cpu0 Debug’  
   

Figure : CPU0 Debug Configuration Name

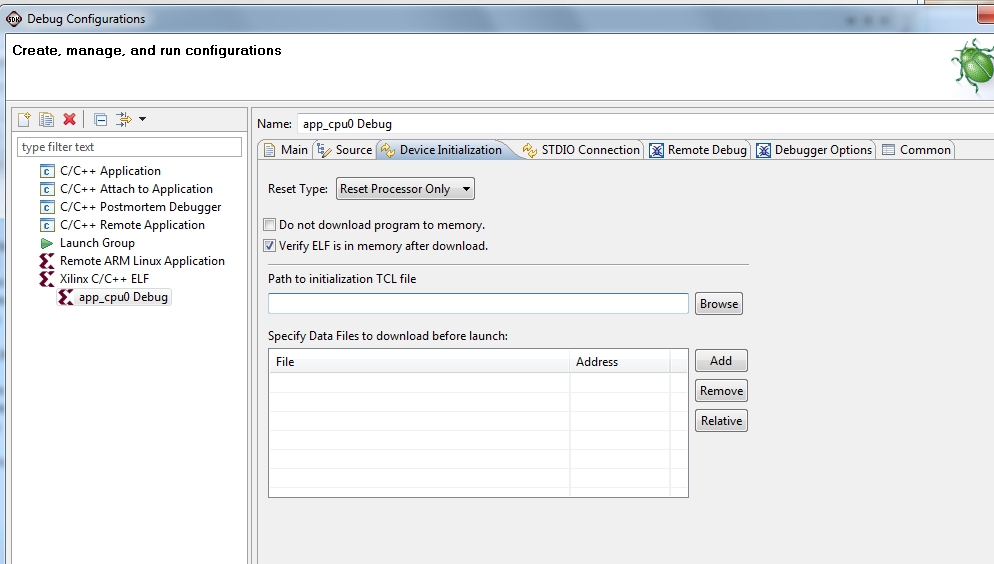
1. Select ‘Device Initialization’ tab and clear out the ‘Path to initialization TCL file’. Initialization has already been done by the FSBL  
   

Figure : CPU0 Debug Initialization

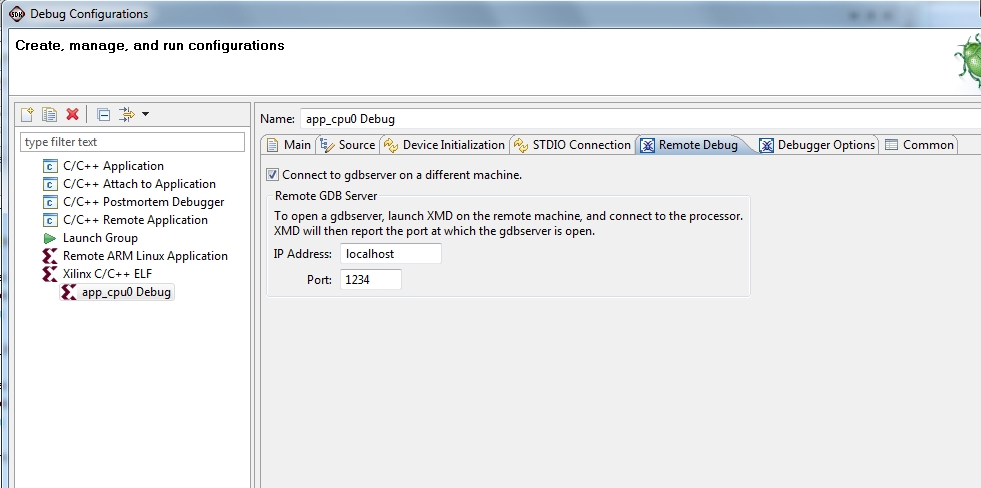
1. Select the ‘Remote Debug’ tab
2. Instruct SDK to connect to the externally created GDB server by selecting ‘Connect to gdbserver on a different machine’. The IP Address should default to ‘localhost’ and the port should be ‘1234’  
   

Figure : CPU0 Remote Debug Configuration

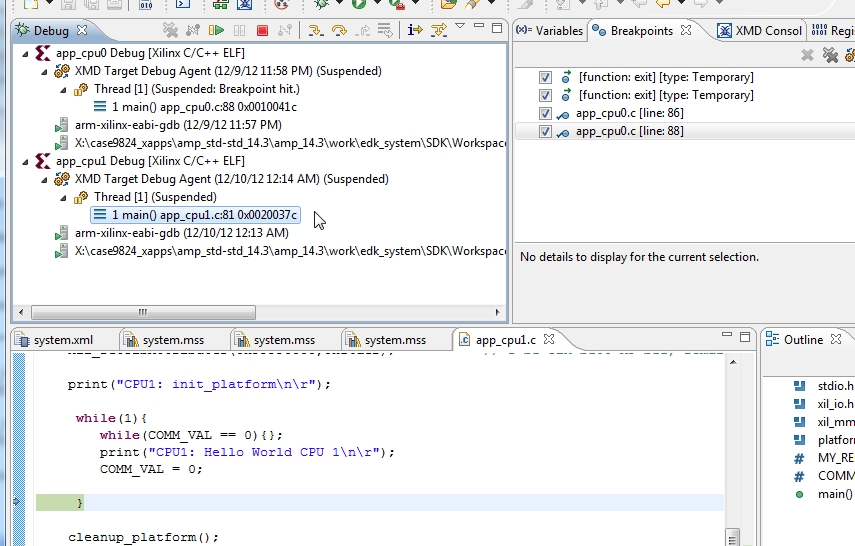
1. Apply
2. Debug. Choose ‘Yes’ to confirm perspective switch
3. The application will be downloaded then executed (the ELF download could have been disabled in the ‘Device Initialization’ Tab). The app will stop at a breakpoint at the first executable line in main(). There are times when the app may not automatically stop at the beginning of main so the pause button (suspend) may have to be pressed. It is possible to restart the app on CPU0 by using SDK to manually set a breakpoint at the beginning of main(), changing the PC (program counter) to 0x00100000, then pressing
4. Press resume, single step, etc to continue running the app.
5. While debugging CPU0, Start Debugging CPU1 in SDK
   1. The SDK view will be the ‘Debug’ view. In the upper right area of SDK, select the C/C+ View
   2. In SDK project explorer window, right click on amp\_standalone\_cpu1 and select debug\_as->debug\_configurations
   3. Highlight ‘Xilinx C/C++ ELF’ and select the ‘New launch configuration’ icon at the top left
   4. The name will be automatically set to ‘amp\_standalone\_cpu1 Debug’
   5. Select ‘Device Initialization’ tab and clear out the ‘Path to initialization TCL file’. Initialization has already been done by Linux and FSBL
   6. Select 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 set to ‘1235’
   8. Apply
   9. Debug
   10. The application will be downloaded then executed. The app will stop at a breakpoint at the first executable line in main().
   11. Press resume, single step, etc to continue running the app.
6. At any point, within the Debug view, the focus can be switched between CPU0 and CPU1 debug by selecting the listed function, under ‘Thread’ on the debug tab. As each function is selected, the visible source will change.   
   

Figure : Debug View

# List of References

* [AMBA AXI4](http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022d/index.html) Protocol Specification
* UG683, EDK Concepts, Tools, and Techniques
* DS768, LogiCORE IP AXI Interconnect
* UG111, Embedded System Tools Reference Manual
* UG761, Xilinx AXI Reference Guide
* UG585, Zynq-7000 EPP Technical Reference Manual (TRM)
* UG873, Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques
* UG642, Platform Specification Format Reference Manual
* UG029, Chipscope Pro Software and Cores User Guide
* UG926, Zynq-7000 ZC702 Getting Started Guide
* UG873, Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT)