XILINX®

XAPP1093 (v1.0.1) January 24, 2014

Simple AMP: Zynq SoC Cortex-A9 Bare-Metal System with MicroBlaze Processor

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. The programmable logic within the Zynq SoC can also contain MicroBlaze™ embedded processors. This application note describes a method of starting up one of the Cortex®-A9 processors and a MicroBlaze processor, 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) 14.5 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 the designer wants to skip the steps of reproducing hardware, software, or boot file targets. The design uses the ZC702 rev. C board with CES silicon and rev. 1.0 board with rev. C silicon.

Introduction

The Zynq-7000 AP SoC provides two Cortex-A9 processors that share common memory and peripherals that are also accessible from a MicroBlaze processor that resides in the programmable logic (PL). Asymmetric multiprocessing (AMP) is a mechanism that allows multiple 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 one Cortex-A9 processor and one MicroBlaze processor in an AMP configuration. The second Cortex-A9 processor (CPU1) is not used in this design. 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.

Some of the low-level features of this application note are:

- SDK to debug multiple processors simultaneously
- MicroBlaze processor access to the Zynq SoC DDR3 through the high-performance (HP) ports
- MicroBlaze processor access to the Zynq SoC processing system (PS) peripherals and on-chip memory (OCM) through the slave general-purpose port (S_GP0)
- Relocation of the MicroBlaze processor vector table
- MicroBlaze Fast Interrupts
- ChipScope™ analyzer virtual input/output (VIO) to control logic
- Measurement of interrupt latency using ChipScope analyzer integrated logic analyzer (ILA)

Design Overview

In this reference design, the Cortex-A9 processor (CPU0) and MicroBlaze processor (MB0) are configured to run their own bare-metal applications. In this AMP example, the bare-metal application running on CPU0 is the master of the system and is responsible for:

- System initialization
- Releasing PL reset
- Communicating with MB0
- Sharing the UART with MB0

The bare-metal application running on MB0 is responsible for:

- Communicating with CPU0
- Servicing interrupts from a core in the PL
- Sharing the UART with CPU0

The Zynq SoC PS includes resources that are private to CPU0 and other resources that are accessible from MB0. 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
- 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 MB0 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 MB0 limits 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 `0x2FFFFFFF`. MB0 uses DDR memory from `0x30000000` to `0x3FFFFFFF` for its bare-metal application.
- 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 MB0. When the flag is zero,

CPU0 owns the UART. When the flag is not zero, MB0 owns the UART. Only CPU0 sets the flag and only MB0 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 VIO core is connected to this core, enabling the user to generate interrupts towards MB0 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 the MicroBlaze processor and 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 the ChipScope analyzer.

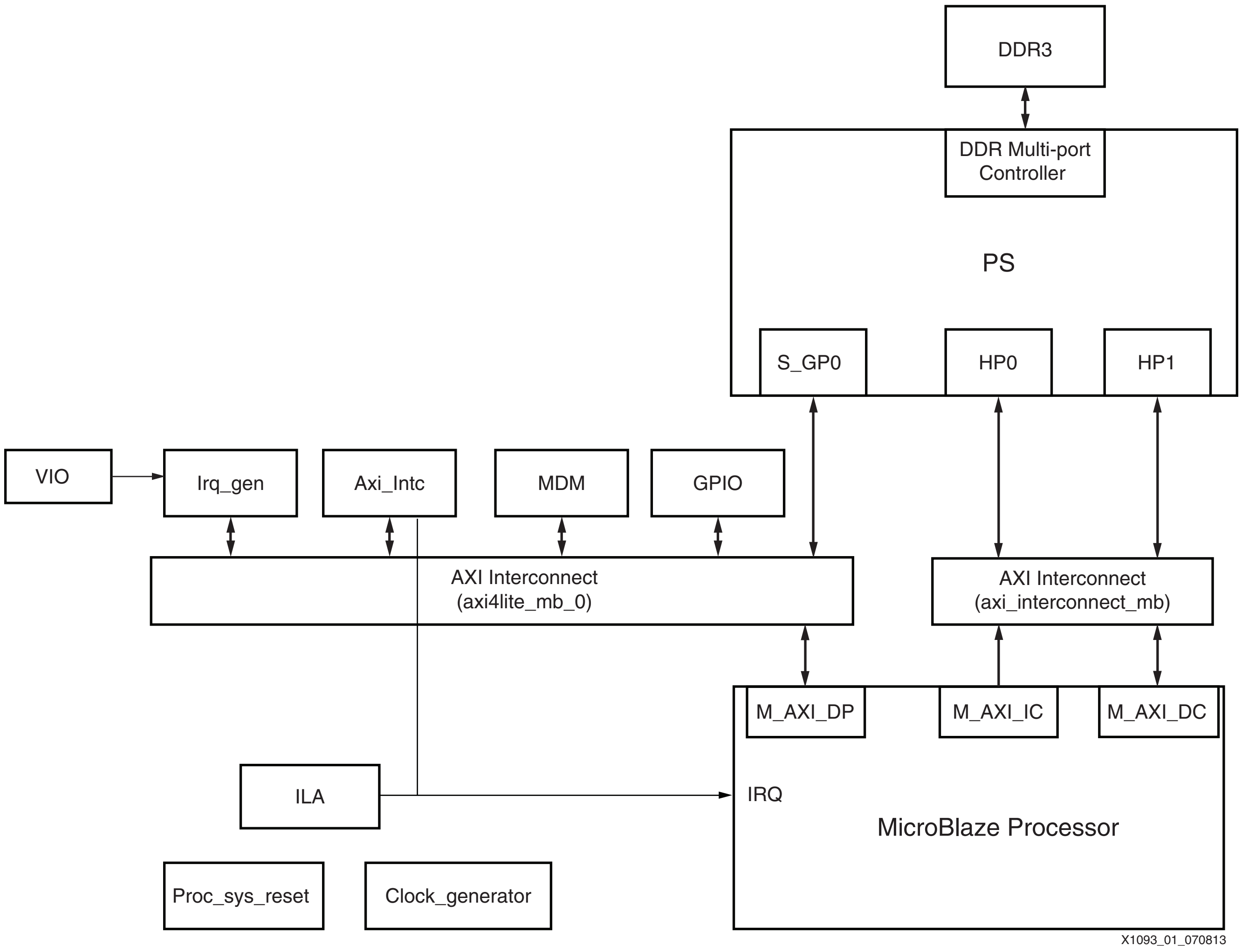


Figure 1: **Block Diagram**

In this design, when the VIO generates a pulse, the custom core forwards an interrupt to the axi_intc core connected to MB0's interrupt input. The core is also connected to MB0's data port (M_AXI_DP) through an AXI Interconnect that allows MB0 access to the control register within the core. MB0 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 MB0. A ChipScope analyzer ILA core is also included and allows the user to measure the latency of the IRQ being serviced.

Address Map

From the point of view of MB0, any instruction or data fetches to the address range of `0x30000000` to `0x3FFFFFFF` are forwarded to the PS HP0 and HP1 ports from the MicroBlaze processor's M_AXI instruction and cache ports. Any accesses outside of that range are forwarded through the MicroBlaze processor's M_AXI_DP data port.

Table 1 contains the address map as viewed by the MicroBlaze processor.

Table 1: **MicroBlaze Address Map**

Address Range	Device/Peripheral
`0x30000000–0x3FFFFFFF`	PS DDR via MB0 cache accesses through the HP ports. Refer to *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 1] for more details regarding access to the PS DDR from the HP ports.
`0x40000000–0x4000FFFF`	GPIO for 4-bit LED access.
`0x41000000–0x4100FFFF`	MicroBlaze Interrupt Controller.
`0x42000000–0x4200FFFF`	MicroBlaze Debug Module.
`0x50000000–0x5000FFFF`	IRQ_GEN custom core.
`0xE0001000–0xE0001FFF`	PS Uart1.
`0xFFFF0000–0xFFFF0000`	PS OCM via the S_GP0 PS port.

Table 2 contains the register description for the IRQ_GEN custom core whose base address is located at `0x50000000`.

Table 2: **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 MB0 interrupt controller. If the VIO_IRQ_TICK pin is asserted (by the VIO), this bit is set. Also, the MB0 can set this bit. Only MB0 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 MB0

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.

Bare-Metal Application Code

The reference design has both CPU0 and MB0 running their own bare-metal application code. CPU0 is responsible for initializing shared resources and starting up MB0 by removing the PL reset.

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. Remaps the full 256 KB of OCM to the top of the address map (`0xFFFC0000` to `0xFFFFFFFF`) and disables DDR memory filtering which enables DDR memory at the bottom of CPU0's address map. Refer to *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 1] for further information regarding the OCM remapping and DDR memory filtering functions.
2. Configures the MMU to disable cache for OCM accesses in the address range of `0xFFFC0000` to `0xFFFFFFFF`.
3. Releases the PL reset.
4. Prints "CPU0: Hello World" to the UART.
5. Waits for the UART TX FIFO to empty.
6. Sets a memory location in OCM that is used as a semaphore flag.
7. Waits for the memory location in OCM that is used as a semaphore flag to be cleared.

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 `0xFFFFFFF0` for a specific value, and then continues the loop. If `0xFFFFFFF0` does not contain the specific value, CPU1 jumps to the fetched address. In this application note, CPU1 is not used so it continues running the wait for event loop indefinitely.

The CPU0 application repeats step 4 to step 7 indefinitely.

The MicroBlaze processor in this design has the parameter 'C_BASE_VECTORS=0x30000000' that instructs MB0 to use memory location `0x30000000` as its reset vector. When CPU0 releases PL reset, MB0 starts executing code at `0x30000000`. The FSBL is responsible for placing the MB0 ELF at `0x30000000`.

MB0 Application

The MB0 application is located in memory starting at address `0x30000000`. The linker script is used to set the starting address.

The MB0 application performs the following:

1. Initializes the interrupt controller and interrupt subsystem.
2. Waits for a memory location in OCM that is used as a semaphore flag to be set.
3. Prints "MB0 : Waiting for IRQ" to the UART.
4. Waits for the interrupt service routine to increment the variable irq_count.
5. Prints "MB0 : IRQ Acknowledged" to the UART and clears the variable irq_count.
6. Waits for the UART TX FIFO to empty.
7. Clears the memory location in OCM that is used as a semaphore flag.

The MB0 application repeats step 2 to step 7 indefinitely.

Inter-Processor Communication

The inter-processor communication in the example design is a semaphore flag. When the semaphore is set, MB0 owns the UART, and when it is cleared by MB0, 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 coherent and deterministic.

MB0 accesses the OCM via the PS slave GP0 port.

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 bursty in nature because the minimum read access reads eight consecutive words. Time would thus be wasted as a read burst occurs to access a single 32-bit value. Additionally, software running on each processor has to flush and invalidate the caches after modifying the shared data to enforce coherency.

Reference Design

The reference design files can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=329031

The reference design matrix is shown in Table 3.

Table 3: **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 14.5
Implementation software tools/versions used	EDK 14.5
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	ZC702 rev. C board with CES silicon and rev. 1.0 board with rev. C silicon

These files are included in the reference design:

- XPS project
- SDK source files for CPU0 and MB0 applications
- Generated files:
    - Bit file
    - All files for the SD card
    - Application ELF files for CPU0 and MB0
- `BOOT.BIN` build scripts
- Modified sw_apps FSBL

Table 4 and Table 5 show the device utilization details.

Table 4: **Device Utilization (1)**

Parameters	Specification/Details	
Device utilization without testbench	Slices	2,155
	BUFGs	3
	DSP48E1s	3
	MMCME2_ADVs	1
	PS7	1
	RAMB36	7
HDL language support		Verilog/VHDL

Table 5: **Device Utilization (2)**

Device	Speed Grade	Package	Pre-Map (Synthesis Constraint)	Post-Route	Slices
XC7Z020	-1	CLG484	176 MHz	357 MHz	2,155 (16%)

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\system.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 **Hardware > generate_bitstream**.

   After completion, the downloadable FPGA bit file is available at:

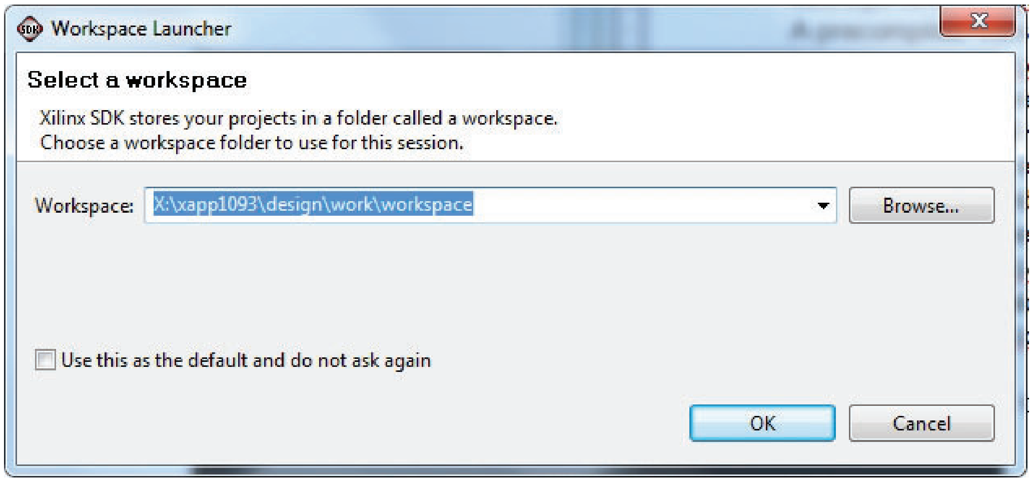
   `design\work\edk_system\implementation\system.bit`

   A precompiled version of the bit file is also available at:

`design\generated_files\system.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.


Workspace Launcher
Select a workspace
Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.
Workspace:
Browse...
Use this as the default and do not ask again
OK
Cancel
X1093_02_041713

Workspace: X:\xapp1093\design\work\workspace

Figure 2: **Select Workspace Directory**

8. Click **OK**.

   SDK automatically creates the `\Workspace` subdirectory.

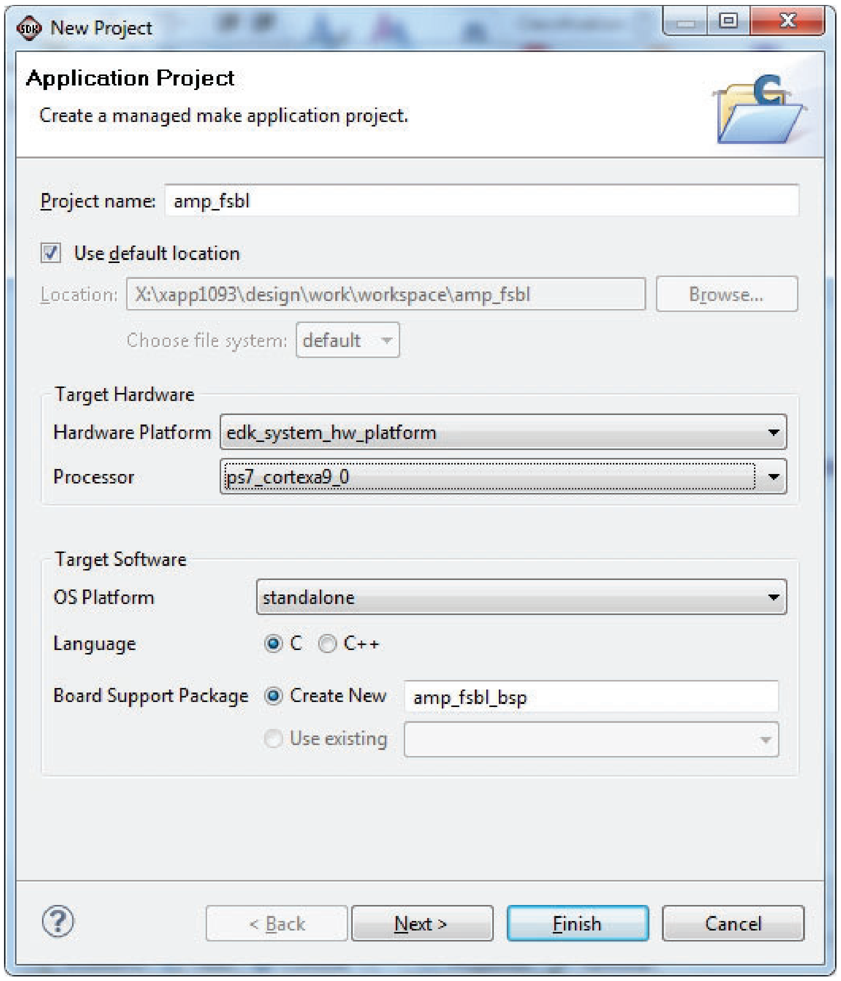
Generating the Applications

Creating FSBL Application

To create the FSBL application:

1. Select **File > New > Application_Project**.

2. In the Project Name field enter **amp_fsbl** and change the Processor field to **ps7_cortexa9_0**, as shown in Figure 3.



Application Project

Create a managed make application project.

Project name: amp_fsbl

[✓] Use default location

Location: X:\xapp1093\design\work\workspace\amp_fsbl Browse...

Choose file system: default

Target Hardware

Hardware Platform: edk_system_hw_platform

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

Language: [✓ C] C++

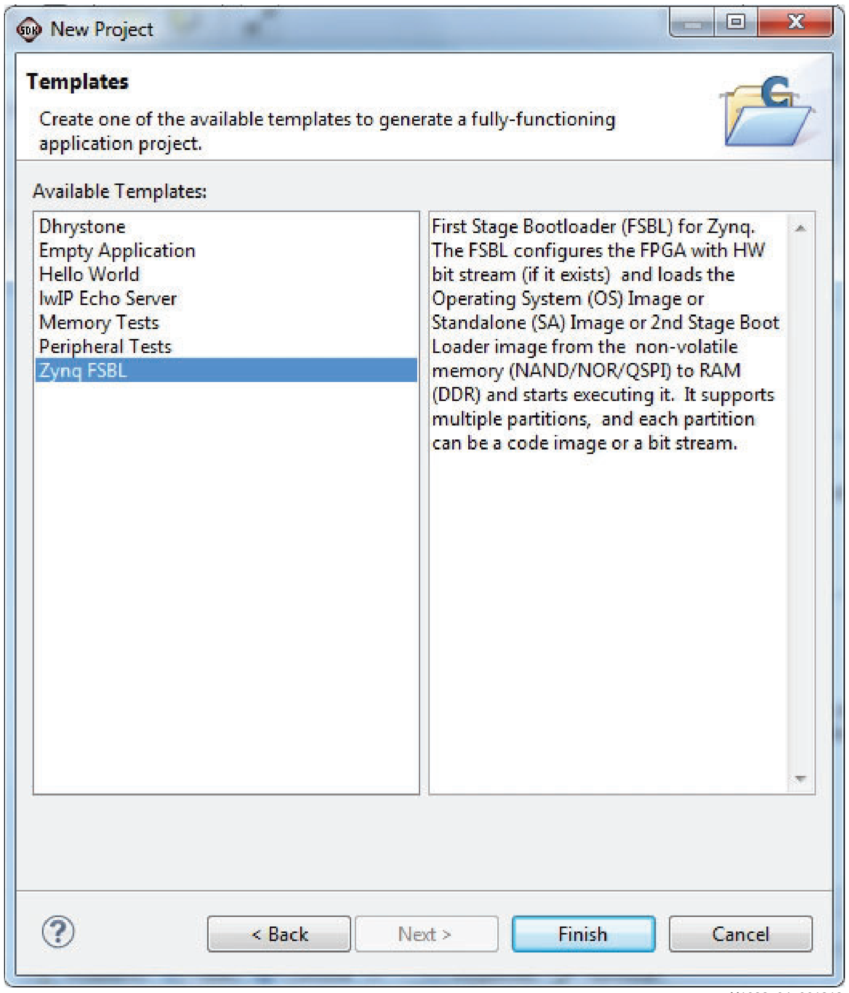
Board Support Package: [✓ Create New] amp_fsbl_bsp

Use existing

Figure 3: **Create FSBL**

3. Click **Next**.

4. In Available Templates select **Zynq FSBL**, as shown in Figure 4.



Create one of the available templates to generate a fully-functioning application project.

First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Figure 4: **Select FSBL Template**

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\amp_fsbl.elf
```

Create Bare-Metal Application For CPU0

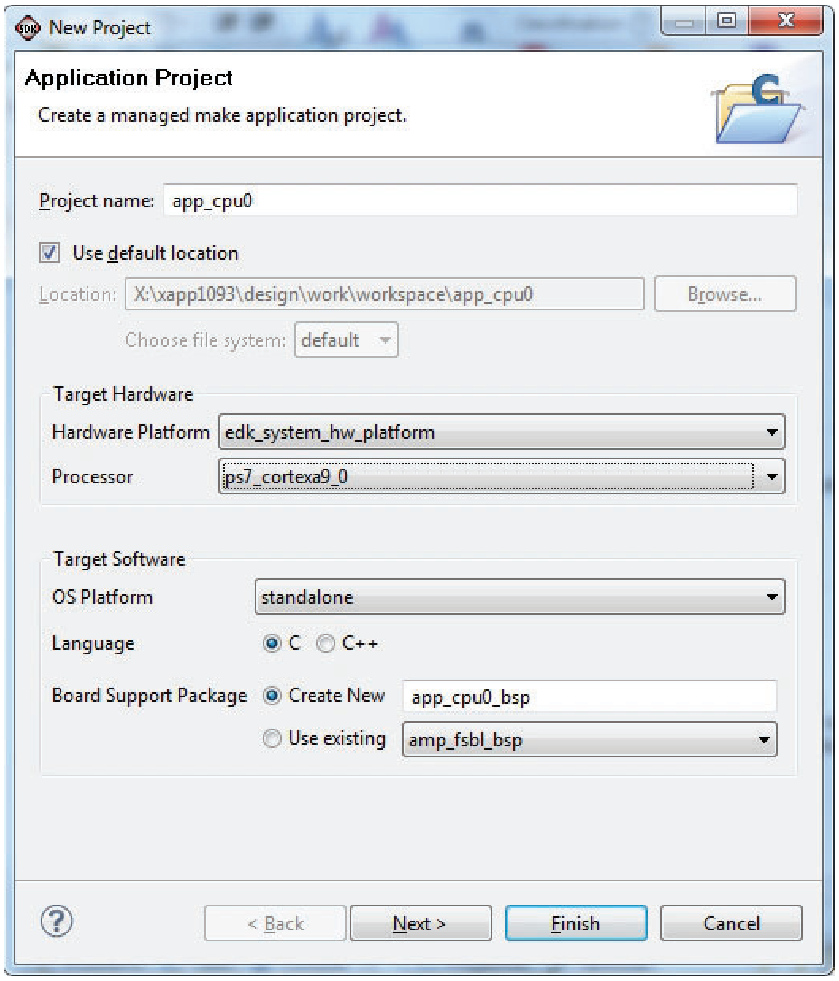
The instructions in this section describe how to create the application ELF that runs on CPU0 after the FSBL copies the application ELFs to DDR memory.

Note: This application has already been compiled and is available at `design\generated_files\app_cpu0.elf`.

To create the CPU0 application:

1. Start SDK.
2. Select **File > New > Application_project**.

3. Change the Project Name to **app_cpu0** and change Processor to **ps7_cortexa9_0**, as shown in Figure 5.



New Project

Application Project

Create a managed make application project.

Project name: app_cpu0

[✓] Use default location

Location: X:\xapp1093\design\work\workspace\app_cpu0

Choose file system: default

Target Hardware

Hardware Platform: edk_system_hw_platform

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

Language: [✓ C] C++

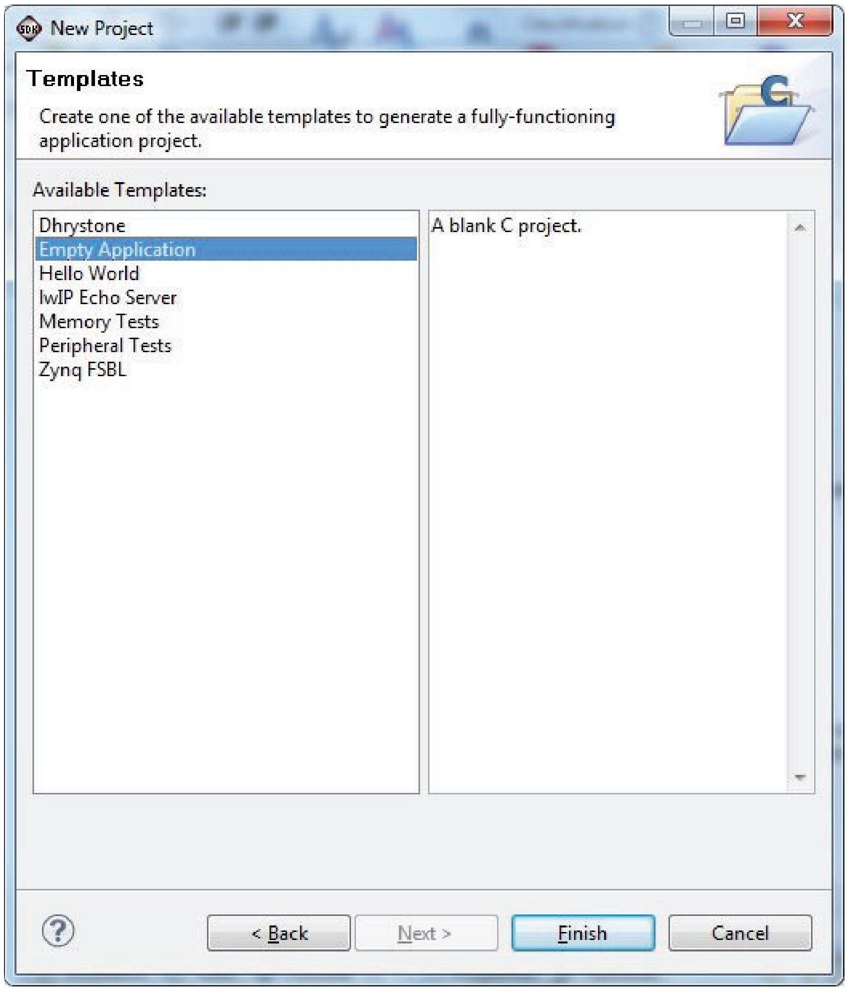
Board Support Package: [✓ Create New] app_cpu0_bsp

Use existing: amp_fsbl_bsp

Figure 5: **Create app_cpu0**

4. Click **Next**.

5. Select **Empty Application**, as shown in Figure 6.



Create one of the available templates to generate a fully-functioning application project.

Available Templates:

Dhrystone
Empty Application
Hello World
lwIP Echo Server
Memory Tests
Peripheral Tests
Zynq FSBL

A blank C project.

X1093_06_061313

Figure 6: **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 7.

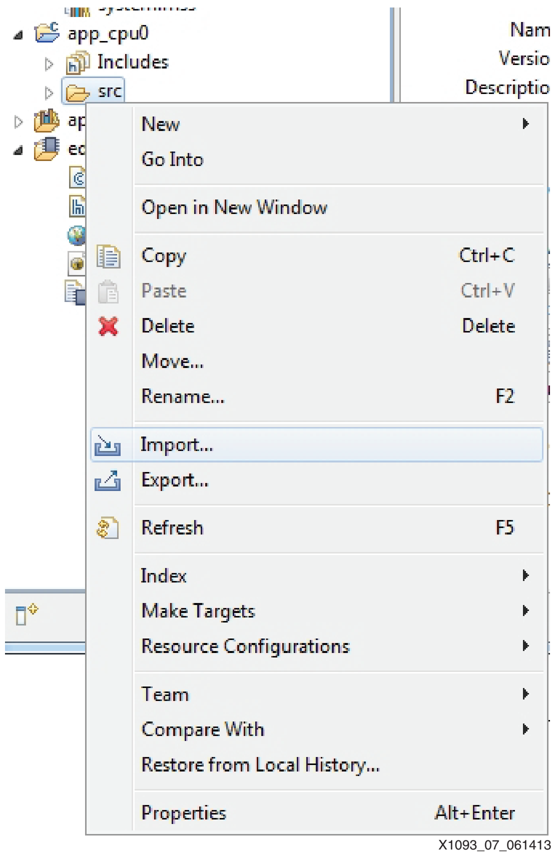


Figure 7: **CPU0 Import**

8. Expand **General** and select **File System**, as shown in Figure 8.

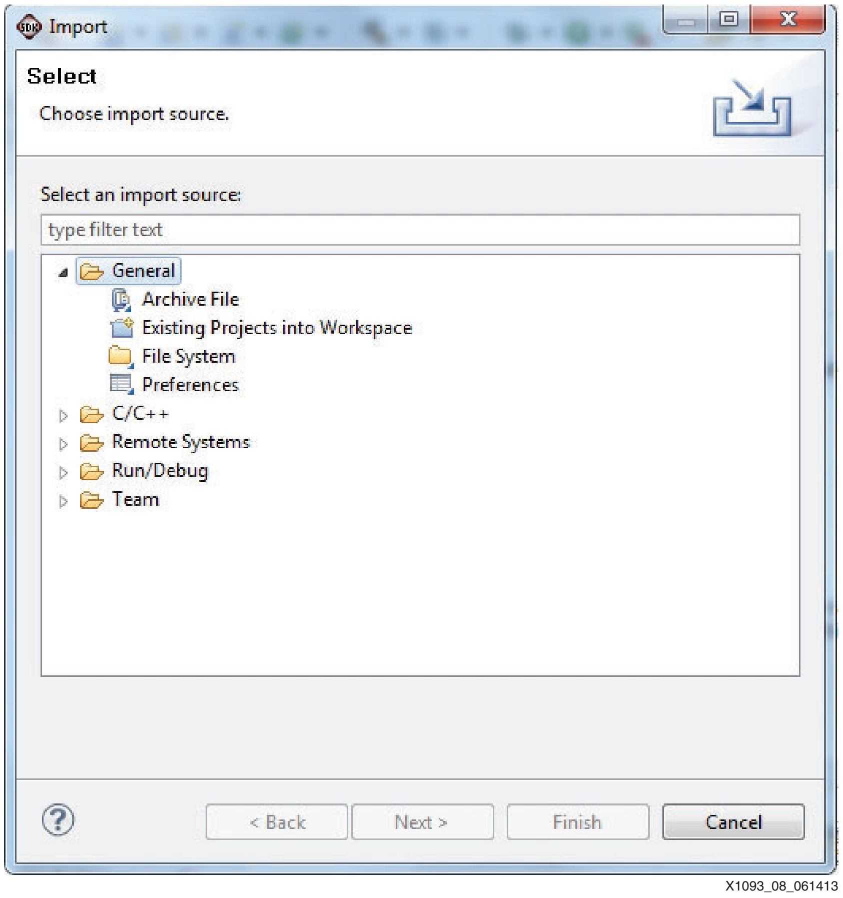


Figure 8: **CPU0 General File System**

9. Click **Next**.

10. Browse to and select the **design\src\apps\app_cpu0** directory, as shown in Figure 9.

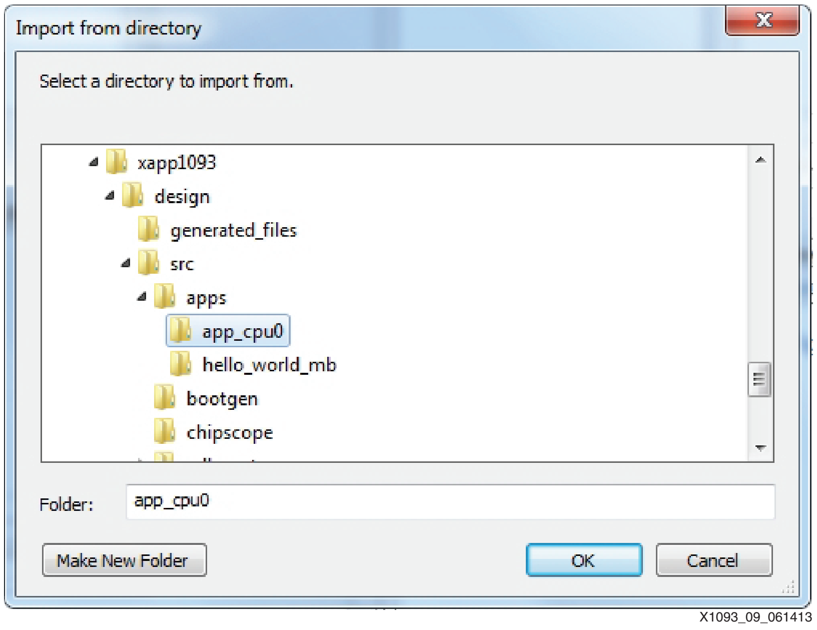
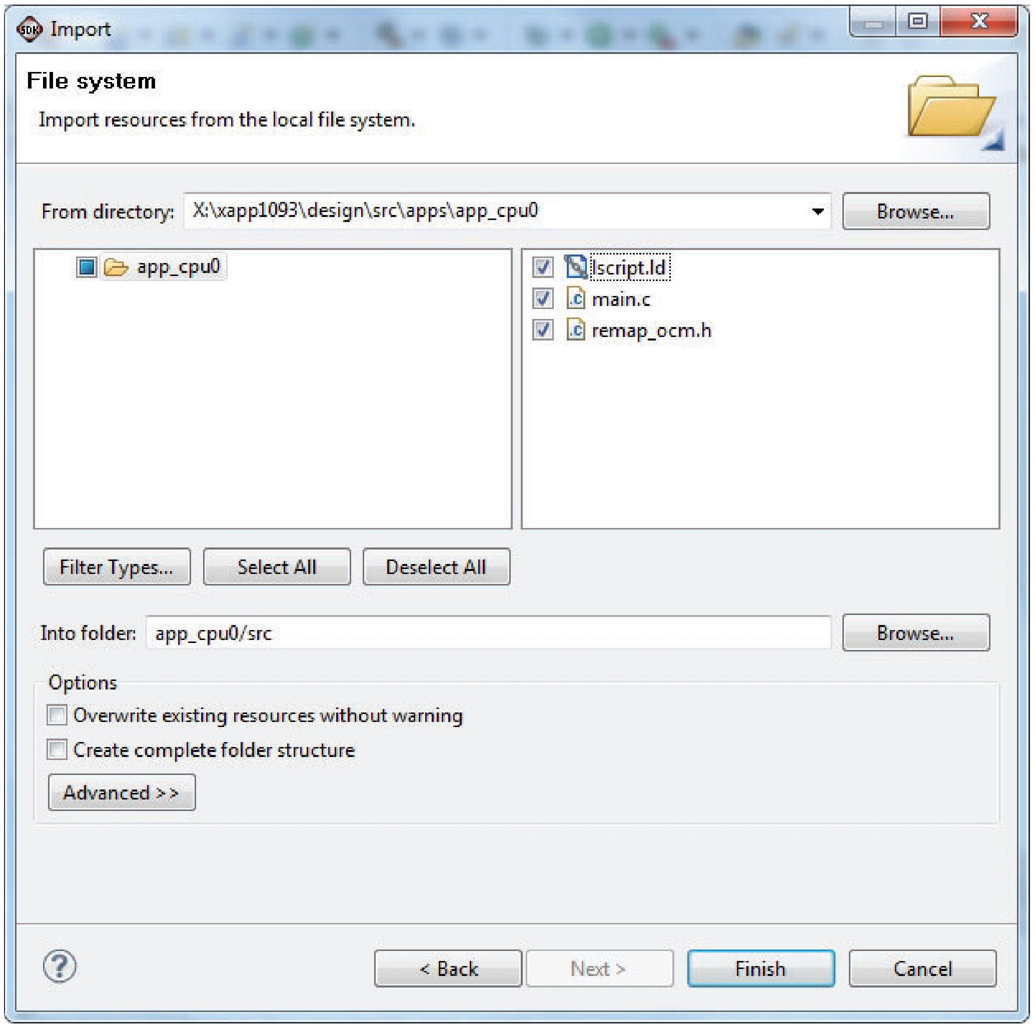


Figure 9: **CPU0 Select Source Directory For Import**

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 10.


Import

File system

Import resources from the local file system.

From directory: X:\xapp1093\design\src\apps\app_cpu0

☐ app_cpu0

☑ lscript.ld
☑ main.c
☑ remap_ocm.h

Into folder: app_cpu0/src

Options

☐ Overwrite existing resources without warning
☐ Create complete folder structure

Figure 10: **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 MB0

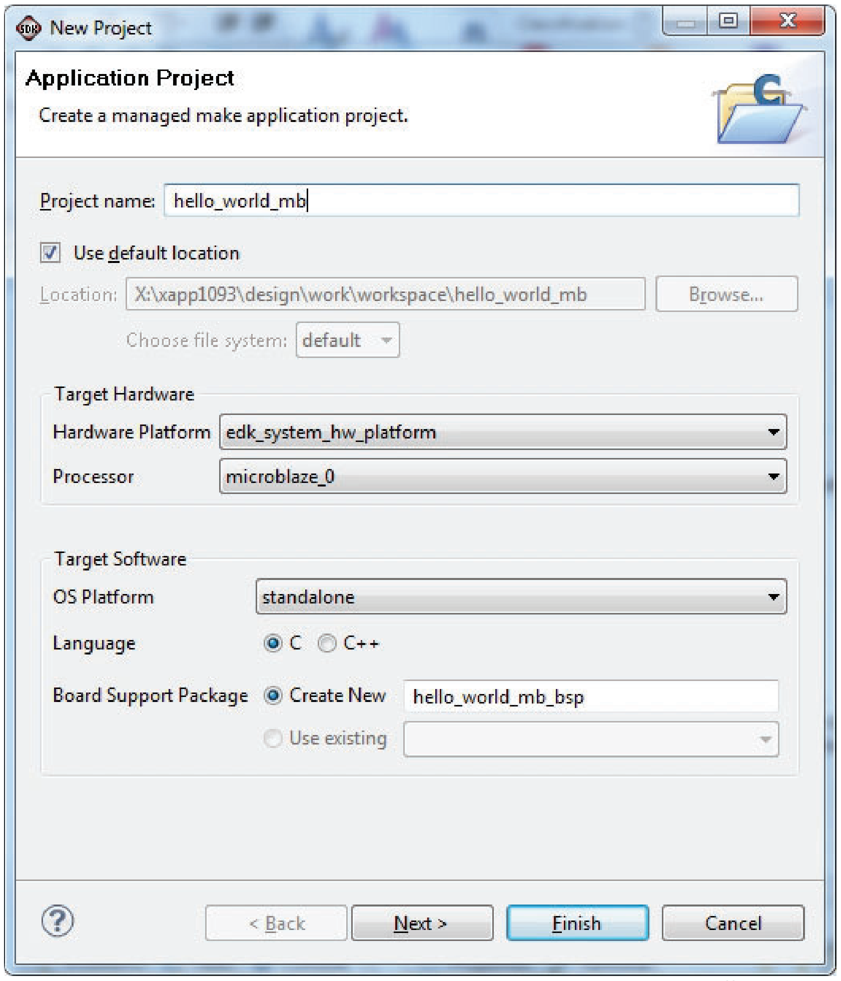
The instructions in this section describe how to create the application ELF that runs on MB0 after the FSBL loads the applications to DDR memory and CPU0 releases PL reset.

Note: This application has already been compiled and is available at `design\generated_files\hello_world_mb.elf`.

To create the bare-metal application that runs on MB0 and to import the included software:

1. Select **File > New > application_project**.
2. Enter the project name as **hello_world_mb**.

3. Make sure Processor is set to **microblaze_0** as shown in Figure 11.



Project name: hello_world_mb

[✓] Use default location

Location: X:\xapp1093\design\work\workspace\hello_world_mb

Choose file system: default

Target Hardware

Hardware Platform: edk_system_hw_platform

Processor: microblaze_0

Target Software

OS Platform: standalone

Language: [● C] C++

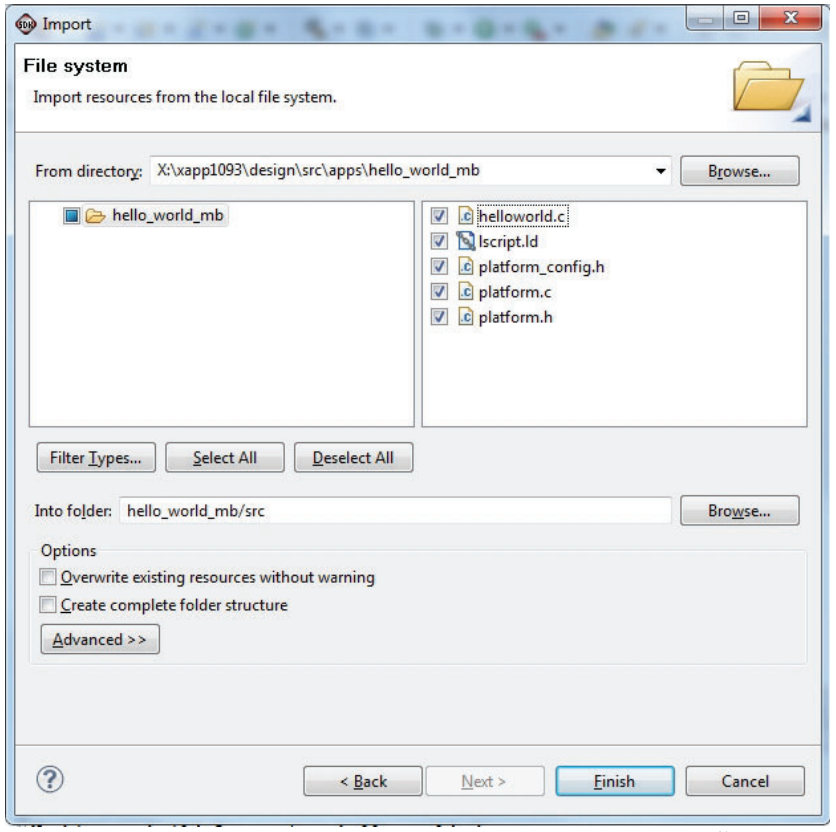
Board Support Package: [● Create New] hello_world_mb_bsp

Use existing

Figure 11: **MB0 Create Application**

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

13. In the right pane, select all files as shown in Figure 12.



From directory: X:\xapp1093\design\src\apps\hello_world_mb

hello_world_mb

- [✓] helloworld.c
- [✓] lscript.ld
- [✓] platform_config.h
- [✓] platform.c
- [✓] platform.h

Into folder: hello_world_mb/src

- [ ] Overwrite existing resources without warning
- [ ] Create complete folder structure

Figure 12: **MB0 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\hello_world_mb\Debug\hello_world_mb.elf`.

Generating the Boot File

The boot file (`BOOT.BIN`) contains the FSBL, FPGA bit file, the ELF for the application that runs on CPU0, and the ELF for the application that runs on CPU1. The design files contain a batch file and BootGen configuration file. 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. MB0 application

A precompiled version of `BOOT.BIN` is available at `design\generated_files\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`) and the `.bif` file (`bootimage.bif`).
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\amp_fsbl.elf`.
3. Copy the bit file from `design\work\edk_system\implementation\system.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\system.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\app_cpu0.elf`.
5. Copy the generated bare-metal application for MB0 from `design\work\edk_system\SDK\Workspace\hello_world_mb\Debug\hello_world_mb.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\hello_world_mb.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.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]

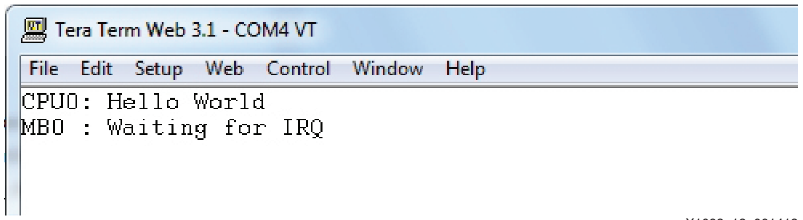
Hardware Setup

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 14.5) 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 14.5) Getting Started Guide* [Ref 2].

If the files were created correctly, CPU0 displays the message “CPU0: Hello World” followed by MB0 displaying “MB0 : Waiting for IRQ,” as shown Figure 13.


Tera Term Web 3.1 - COM4 VT
File Edit Setup Web Control Window Help
X1093_13_061413

```
CPU0: Hello World
MB0 : Waiting for IRQ
```

Figure 13: **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, and the CPU0 and MB0 ELFs. 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 releases PL reset and the MB0 starts running its application. Because MB0 has the parameter C_BASE_VECTORS=0x30000000, MB0 jumps to the reset vector located at `0x30000000` and starts running the application previously downloaded by the FSBL. The starting address of MB0’s application was defined in the `lscript.ld` linker script for the `hello_world_mb.elf` application.

The ChipScope analyzer VIO core is used to generate interrupts towards MB0. A ChipScope 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.

   ChipScope analyzer displays the two devices (ARM_DAP and XC7Z020) that are in the 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\cs.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 **gen_irq** button as shown in Figure 14. The figure also shows the waveform for the interrupt generated after the **gen_irq** button is pressed.

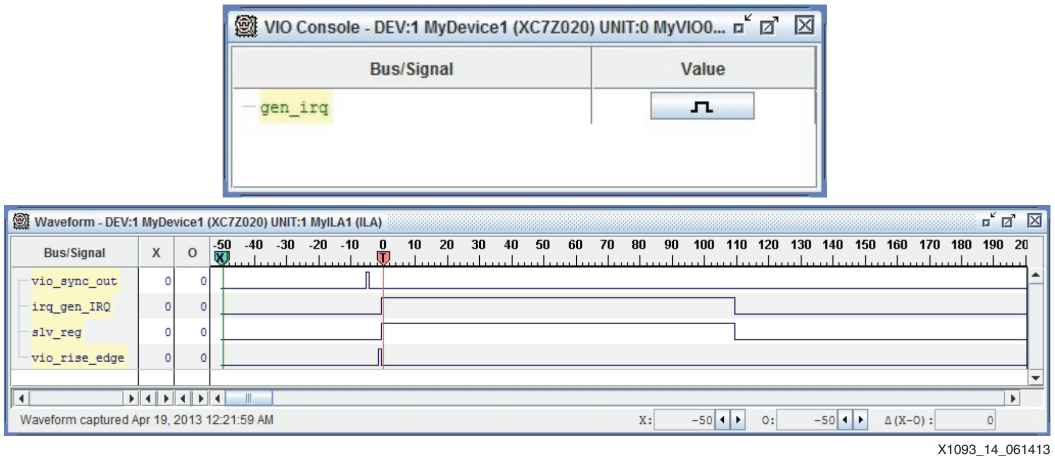
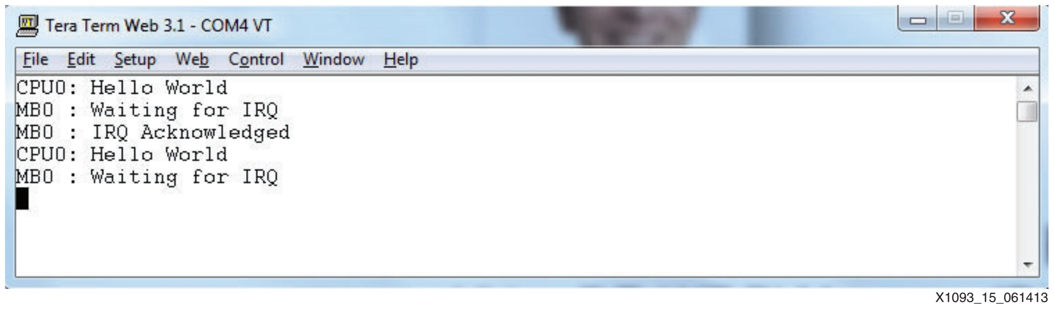

VIO Console - DEV:1 MyDevice1 (XC7Z020) UNIT:0 MyVIO0...
Bus/Signal
Value
gen_irq
Waveform - DEV:1 MyDevice1 (XC7Z020) UNIT:1 MyILA1 (ILA)
Bus/Signal
X
O
vio_sync_out
irq_gen_IRQ
slv_reg
vio_rise_edge
Waveform captured Apr 19, 2013 12:21:59 AM
X: -50
O: -50
Δ(X-O): 0
X1093_14_061413

Figure 14: **ChipScope Analyzer Capture of First IRQ**

MB0 services the interrupt and increments the global flag irq_count. MB0's main() detects the serviced interrupt by waiting for irq_count to be set, prints "MB0 : IRQ Acknowledged," and then clears the OCM memory location used to communicate to CPU0 that MB0 now owns the UART. CPU0 responds by printing "CPU0: Hello World" and passes control back to MB0 by setting the OCM memory location. MB0 detects the OCM memory location being set, prints "MB0 : Waiting for IRQ" and then waits in a while() loop until another interrupt occurs, as shown in Figure 15.


Tera Term Web 3.1 - COM4 VT
File Edit Setup Web Control Window Help
X1093_15_061413

```
CPU0: Hello World
MB0 : Waiting for IRQ
MB0 : IRQ Acknowledged
CPU0: Hello World
MB0 : Waiting for IRQ
```

Figure 15: **Console Output after Chipscope Analyzer Trigger**

Every time the virtual button is clicked, an interrupt is created and the MB0 IRQ service routine sets a global variable that the MB0 main() function uses to print to the console. In Figure 16, it can be seen how subsequent IRQs have much lower interrupt latency due to the IRQ service routine being cached.

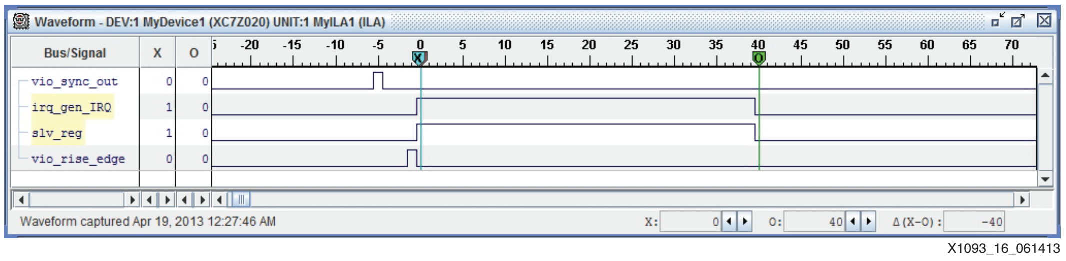

Waveform - DEV:1 MyDevice1 (XC7Z020) UNIT:1 MyILA1 (ILA)
Bus/Signal
X
O
vio_sync_out
irq_gen_IRQ
slv_reg
vio_rise_edge
Waveform captured Apr 19, 2013 12:27:46 AM
X: 0
O: 40
Δ(X-O): -40
X1093_16_061413

Figure 16: **Chipscope Subsequent Capture**

The bare-metal application that services the interrupt is located in the PS DDR memory. When the first interrupt occurs, MB0 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 14, there is a delay of 110 clocks between the interrupt being asserted and the service routine 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 MB0 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 16, the interrupt service completed after 40 clocks. This delay is approximately one third of the delay for the first, non-cached, interrupt service.

The time difference between the first and following interrupt services could be reduced by moving the service routine into local memory bus (LMB) memory on the MicroBlaze processor.

Debugging the Design

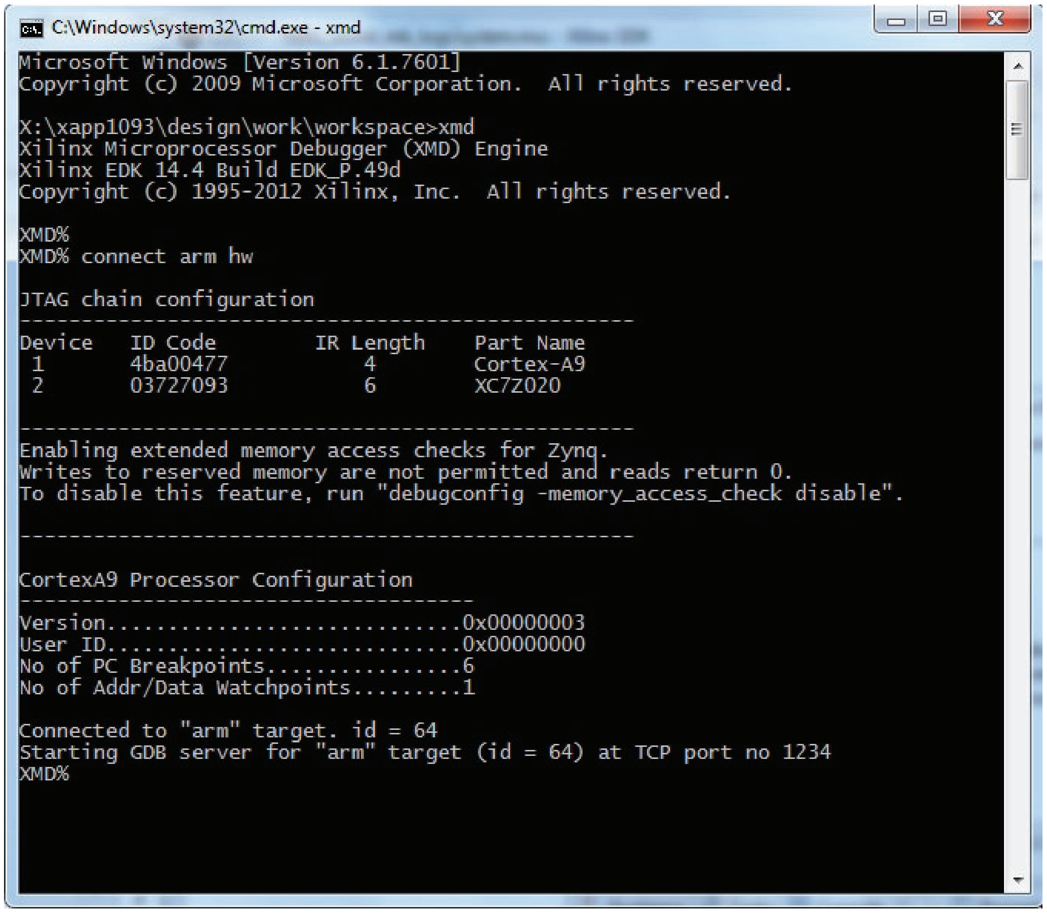
SDK can be used to connect and debug the applications running on both CPU0 and MB0 in tandem. Xilinx Microprocessor Debug (XMD) provides a command shell and GDB server that connects to each processor by way of the JTAG cable. Normally, SDK automatically starts XMD in the background when starting to debug an application. For this example design, XMD is manually started outside of SDK to connect to both CPU0 and MB0. 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, which in turn resets the PL, in order to debug the processors simultaneously.

To prepare to debug the design:

1. Connect the mini USB 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 17.



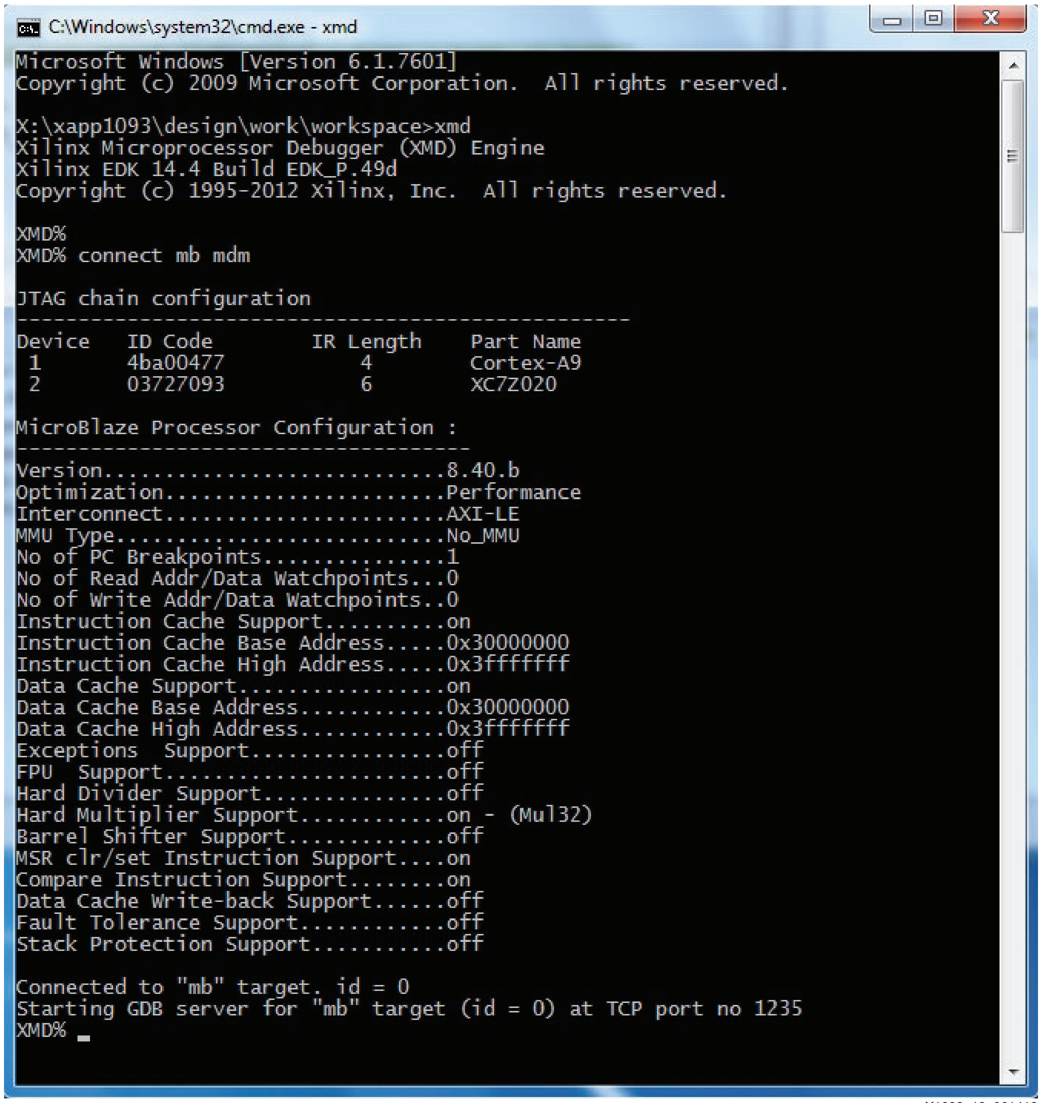
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

X:\xapp1093\design\work\workspace>xmd
Xilinx Microprocessor Debugger (XMD) Engine
Xilinx EDK 14.4 Build EDK_P.49d
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

--------------------------------------------------
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".

--------------------------------------------------

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%
```

X1093_17_061413

Figure 17: **Connect XMD to CPU0**

5. In SDK, select **Xilinx_Tools > Launch_shell** to open another Xilinx command shell.
6. At the command shell prompt enter **xmd**.
7. Enter the command **`connect mb mdm`**.

8. XMD should respond with the TCP port number 1235, as shown in Figure 18.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

X:\xapp1093\design\work\workspace>xmd
Xilinx Microprocessor Debugger (XMD) Engine
Xilinx EDK 14.4 Build EDK_P.49d
Copyright (c) 1995-2012 Xilinx, Inc.  All rights reserved.

XMD%
XMD% connect mb mdm

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

MicroBlaze Processor Configuration :
-------------------------------------
Version............................8.40.b
Optimization.......................Performance
Interconnect.......................AXI-LE
MMU Type...........................No_MMU
No of PC Breakpoints...............1
No of Read Addr/Data Watchpoints...0
No of Write Addr/Data Watchpoints..0
Instruction Cache Support..........on
Instruction Cache Base Address.....0x30000000
Instruction Cache High Address.....0x3fffffff
Data Cache Support.................on
Data Cache Base Address............0x30000000
Data Cache High Address............0x3fffffff
Exceptions  Support................off
FPU  Support.......................off
Hard Divider Support...............off
Hard Multiplier Support............on - (Mul32)
Barrel Shifter Support.............off
MSR clr/set Instruction Support....on
Compare Instruction Support........on
Data Cache Write-back Support......off
Fault Tolerance Support............off
Stack Protection Support...........off

Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1235
XMD% _
```

X1093_18_061413

Figure 18: **Connect XMD to MB0**

Two GDB servers are now running and listening to TCP ports 1234 and 1235. As XMD connects to each processor, the processor is stopped.

To start debugging CPU0:

1. In the SDK Project Explorer window, right-click **app_cpu0** and select **Debug As > debug_configurations**, as shown in Figure 19.

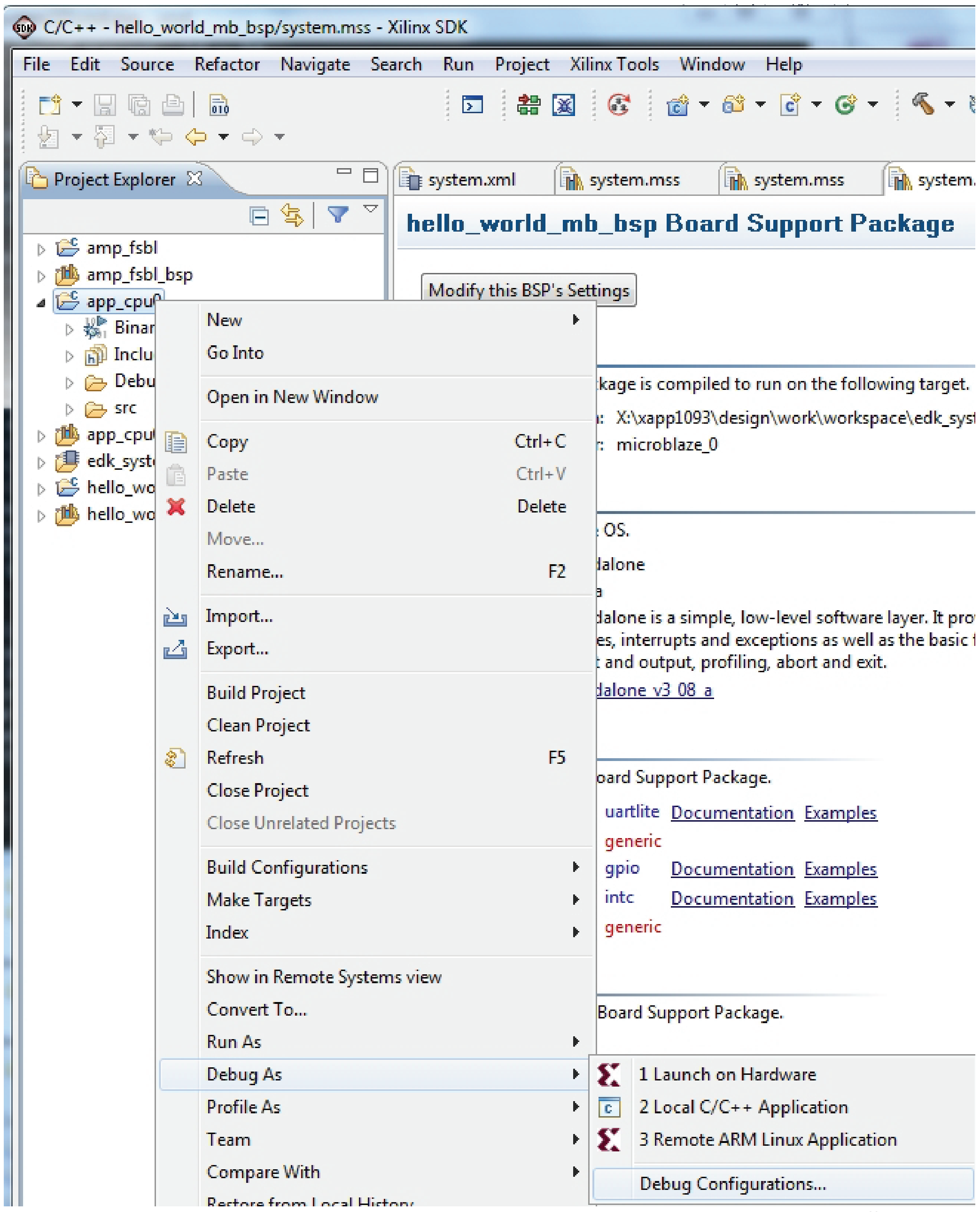


Figure 19: **CPU0 Debug Configuration**

2. Select **Xilinx C/C++ ELF**, then click the **New launch configuration** icon at the top left, as shown in Figure 20.

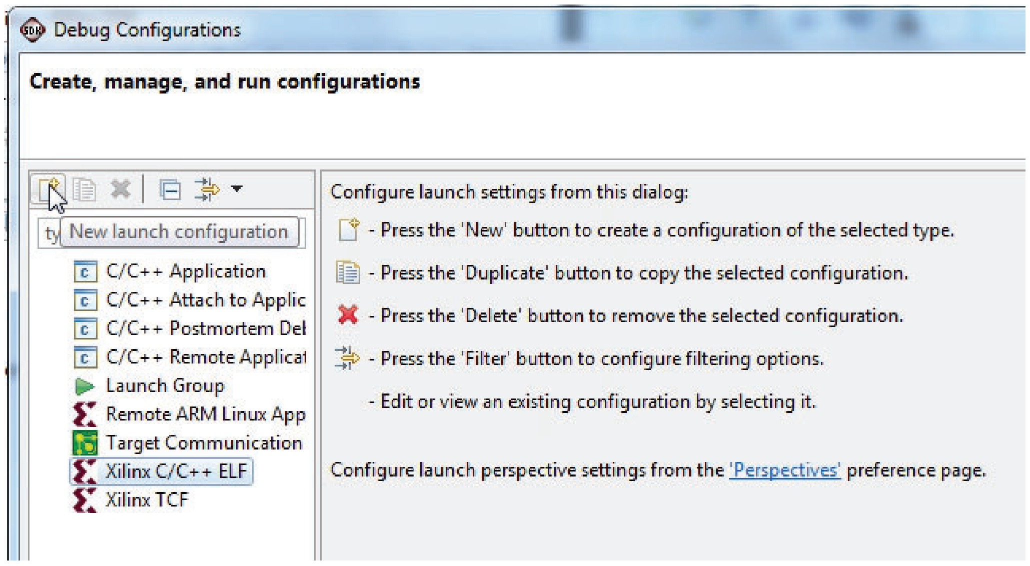
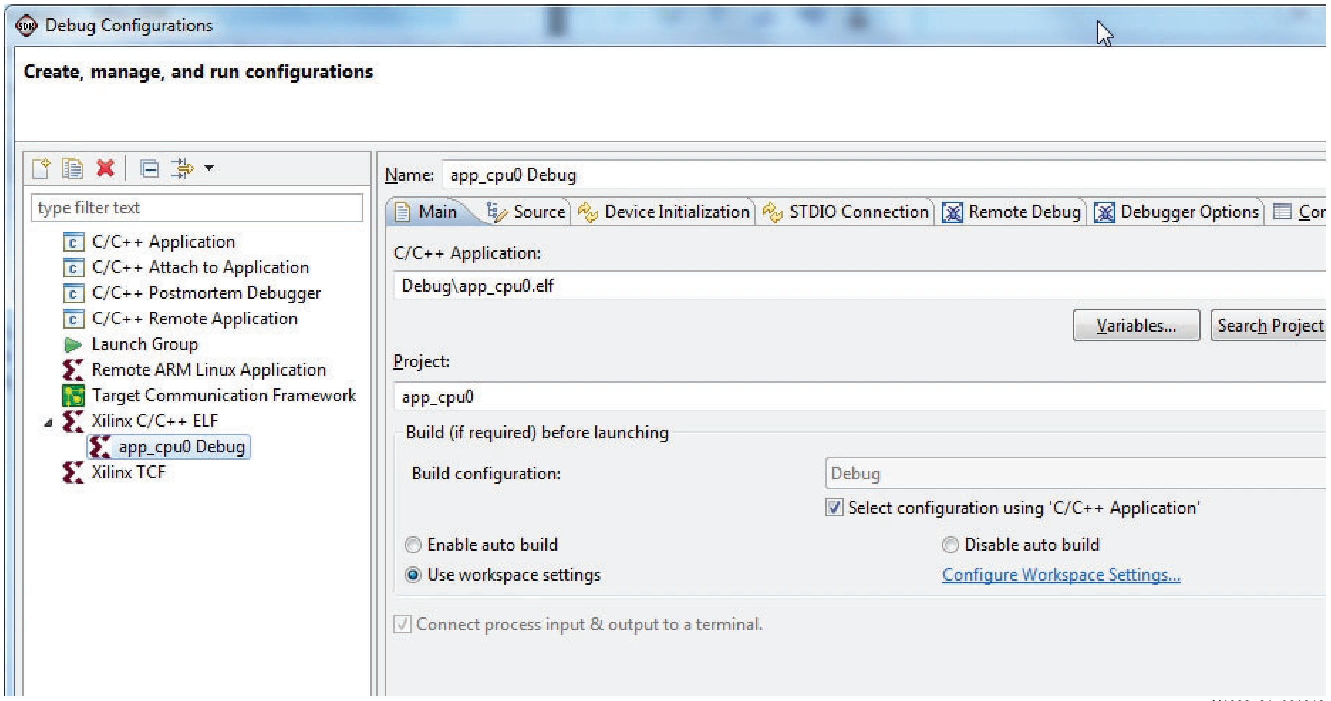

Debug Configurations
Create, manage, and run configurations
New launch configuration
C/C++ Application
C/C++ Attach to Applic
C/C++ Postmortem Del
C/C++ Remote Applicat
Launch Group
Remote ARM Linux App
Target Communication
Xilinx C/C++ ELF
Xilinx TCF
Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.
Configure launch perspective settings from the 'Perspectives' preference page.
X1093_20_061313

Figure 20: **CPU0 Debug Configurations**

The name is automatically set to **app_cpu0 Debug**, as shown in Figure 21.



Name: app_cpu0 Debug

C/C++ Application: Debug\app_cpu0.elf

Project: app_cpu0

Build (if required) before launching

Build configuration: Debug

[✓] Select configuration using 'C/C++ Application'

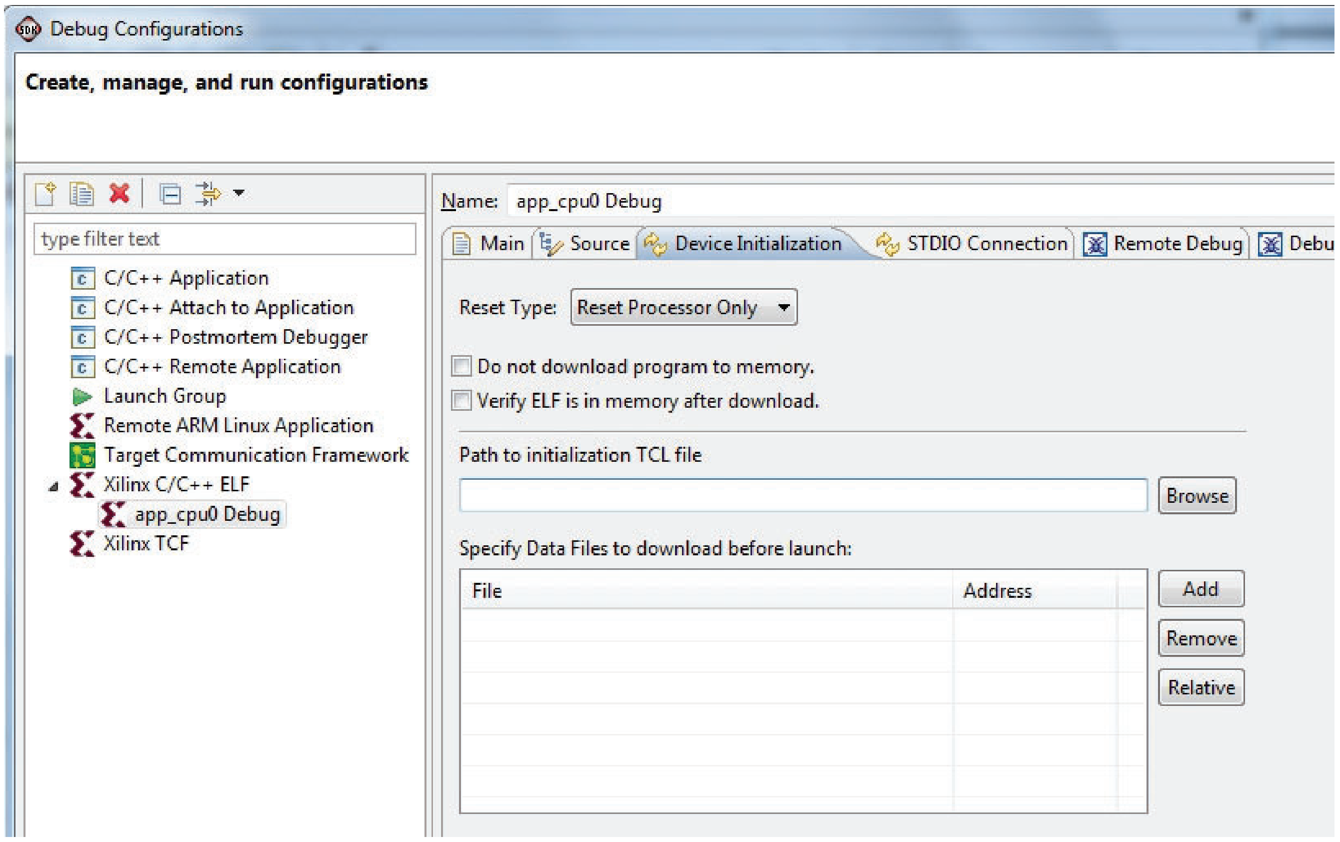
Enable auto build [✓ Use workspace settings] Disable auto build

[✓] Connect process input & output to a terminal.

Figure 21: **CPU0 Debug Configuration Name**

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

4. Clear the **Path to initialization TCL file** field as shown in Figure 22. (Initialization of the PS has already been done by the FSBL.)



Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
- Launch Group
- Remote ARM Linux Application
- Target Communication Framework
- Xilinx C/C++ ELF
  - app_cpu0 Debug
- Xilinx TCF

Name: app_cpu0 Debug

Main | Source | Device Initialization | STDIO Connection | Remote Debug | Debu

Reset Type: Reset Processor Only

☐ Do not download program to memory.

☐ Verify ELF is in memory after download.

Path to initialization TCL file

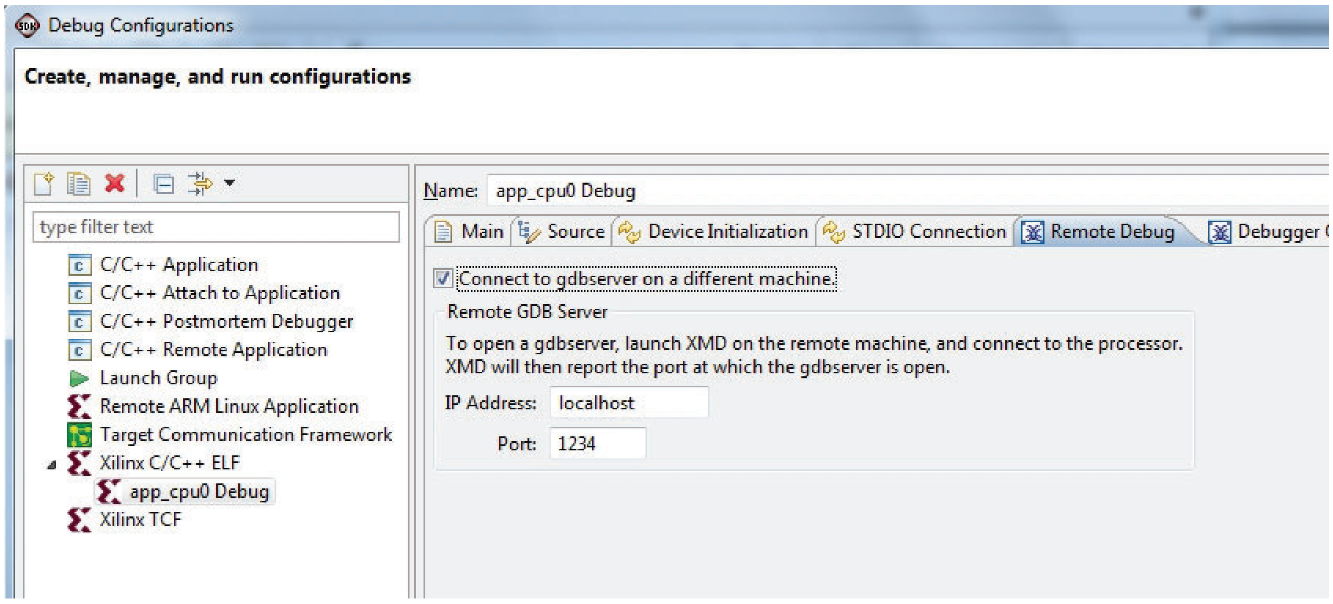
Specify Data Files to download before launch:

File	Address

Figure 22: **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 defaults to **localhost** and the port should be **1234**, as shown in Figure 23.


Debug Configurations

Create, manage, and run configurations

Name: app_cpu0 Debug

Tabs: Main | Source | Device Initialization | STDIO Connection | Remote Debug | Debugger

☑ Connect to gdbserver on a different machine.

Remote GDB Server

To open a gdbserver, launch XMD on the remote machine, and connect to the processor. XMD will then report the port at which the gdbserver is open.

IP Address: localhost

Port: 1234

Figure 23: **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 because the FSBL had already downloaded the ELF.) 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.

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

While debugging CPU0, start debugging MB0 by following these steps:

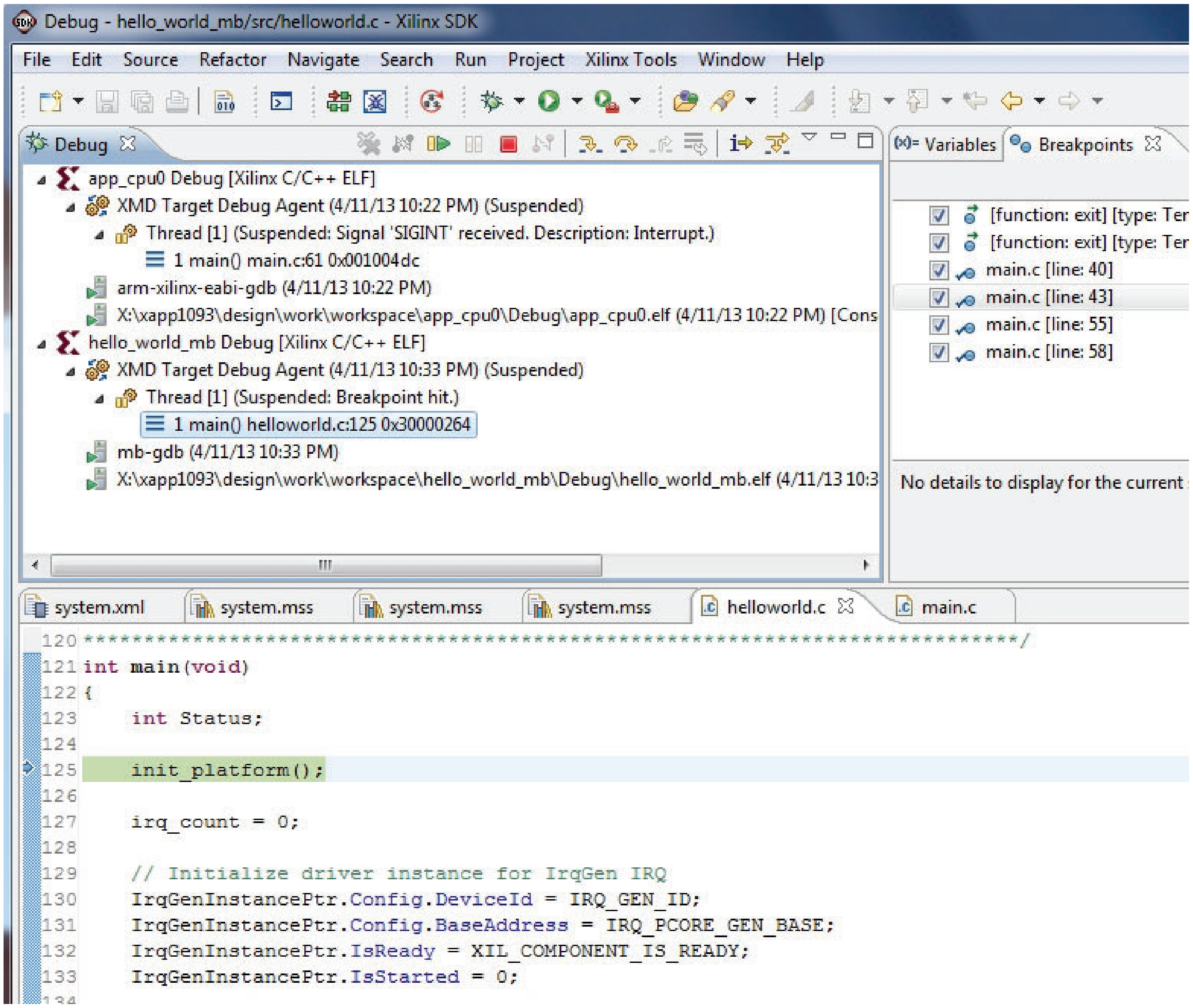
1. In the SDK window, select **C/C+ View**.
2. In the Project Explorer window, right-click **hello_world_mb** and select **Debug As > debug_configurations**.
3. Select **Xilinx C/C++ ELF**, then click the **New launch configuration** icon at the top left. The name is automatically set to **hello_world_mb Debug**.
4. Click the **Device Initialization** tab.
5. Clear the **Path to initialization TCL file** checkbox (initialization has already been done by CPU0 when it ran 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 defaults 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 MB0 debug by selecting the listed function under Thread. As each function is selected, the visible source changes, as shown in Figure 24.


Debug - hello_world_mb/src/helloworld.c - Xilinx SDK
File Edit Source Refactor Navigate Search Run Project Xilinx Tools Window Help
Debug
(x)= Variables
Breakpoints

```
app_cpu0 Debug [Xilinx C/C++ ELF]
  XMD Target Debug Agent (4/11/13 10:22 PM) (Suspended)
    Thread [1] (Suspended: Signal 'SIGINT' received. Description: Interrupt.)
      1 main() main.c:61 0x001004dc
  arm-xilinx-eabi-gdb (4/11/13 10:22 PM)
  X:\xapp1093\design\work\workspace\app_cpu0\Debug\app_cpu0.elf (4/11/13 10:22 PM) [Cons
hello_world_mb Debug [Xilinx C/C++ ELF]
  XMD Target Debug Agent (4/11/13 10:33 PM) (Suspended)
    Thread [1] (Suspended: Breakpoint hit.)
      1 main() helloworld.c:125 0x30000264
  mb-gdb (4/11/13 10:33 PM)
  X:\xapp1093\design\work\workspace\hello_world_mb\Debug\hello_world_mb.elf (4/11/13 10:3
```

```
[function: exit] [type: Ter
[function: exit] [type: Ter
main.c [line: 40]
main.c [line: 43]
main.c [line: 55]
main.c [line: 58]
No details to display for the current
```

```
*****************************************************************************/
int main(void)
{
    int Status;

    init_platform();

    irq_count = 0;

    // Initialize driver instance for IrqGen IRQ
    IrqGenInstancePtr.Config.DeviceId = IRQ_GEN_ID;
    IrqGenInstancePtr.Config.BaseAddress = IRQ_PCORE_GEN_BASE;
    IrqGenInstancePtr.IsReady = XIL_COMPONENT_IS_READY;
    IrqGenInstancePtr.IsStarted = 0;
```

X1093_24_061413

Figure 24: **Debug View**

Debugging the Design Without FSBL

In Debugging the Design, page 22, a debug method was provided that debugs the system after the FSBL has initialized the PS, written both applications to DDR memory, and run CPU0's application that removes reset from the PL.

These steps describe another method of debug that does not rely on the FSBL:

1. If SDK is currently connected to debug sessions:
   a. Terminate each debug agent.
   b. Remove all terminated sessions.

c. Exit from all XMD sessions.

2. If the ChipScope analyzer is connected to the cable, select **JTAG Chain > Close Cable**.
3. Remove the SD card from the ZC702 board.
4. Power cycle the ZC702 board.
5. Download the FPGA using **SDK Xilinx Tools > Program FPGA**.
6. Modify `app_cpu0/src/main.c` by removing the comments on lines 60 and 61.
7. Create a new debug configuration for app_cpu0 and leave all settings at their defaults. (That is, do not change Reset Type or Path to initialization TCL file, and do not select Remote Debug.)
8. Start the new debug configuration.

   SDK starts an XMD session, connects to the PS, runs PS7_init and init_user, and then downloads the ELF and starts running the application until it gets to the first line in main().
9. Run or single step through CPU0's application at least until it gets past line 68 where the PL is released from reset, otherwise XMD is unable to connect to MB0.
10. Launch a new shell, start XMD, and connect mb mdm.

    Note: The TCP port is 1235 because SDK already started its own XMD when it connected to CPU0.
11. In SDK, start the previously created debug session for hello_world_mb.

    At this point, SDK downloads the MB0 ELF and overwrites the bootloop that CPU0 had placed at `0x30000000`. SDK then starts running the application and stops at the first executable line within main().
12. If the ChipScope analyzer is currently open, re-connect to the JTAG cable. Otherwise, start ChipScope analyzer, connect to the JTAG cable, and then open the `cs.cpj` Chipscope project file.

If at any point the MicroBlaze processor becomes unresponsive, enter the following commands in MB0's XMD command window:

- **`stop`**
- **`rst`**

Conclusion

The example design demonstrates how to boot the Zynq-7000 AP SoC and start a Cortex-A9 processor and MicroBlaze processor, each running their own bare-metal application. Leveraging the low overhead of a bare-metal application on MB0, 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. *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585)
2. *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.5) Getting Started Guide* (UG926)
3. *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT)* (UG873)
4. AMBA AXI4 Protocol Specification
   http://www.arm.com/products/system-ip/amba/amba-open-specifications.php
5. *EDK Concepts, Tools, and Techniques* (UG683)
6. *LogiCORE IP AXI Interconnect* (DS768)
7. *Embedded System Tools Reference Manual* (UG111)
8. *Xilinx AXI Reference Guide* (UG761)
9. *Platform Specification Format Reference Manual* (UG642)

10. *ChipScope Pro Software and Cores User Guide* (UG029)
11. Zynq-7000 Base Targeted Reference Design 14.5
http://www.wiki.xilinx.com/Zynq+Base+TRD+14.5
12. *MicroBlaze Processor Reference Guide* (UG081)

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
07/08/2013	1.0	Initial Xilinx release.
01/24/2014	1.0.1	Minor typographical corrections.

Notice of Disclaimer

Automotive Applications Disclaimer