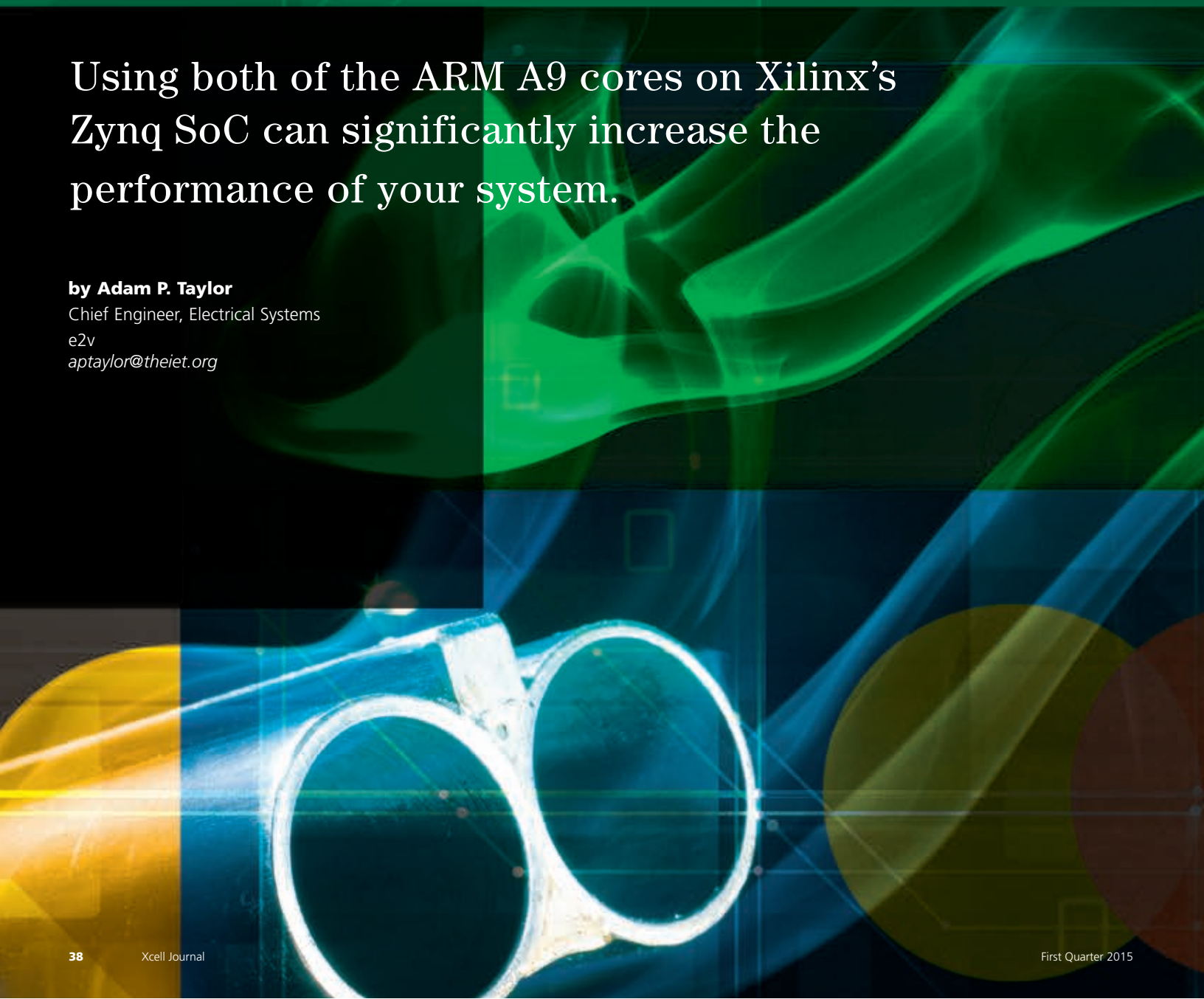
A Double-Barreled Way to Get the Most from Your Zynq SoC



Using both of the ARM A9 cores on Xilinx's Zynq SoC can significantly increase the performance of your system.

by Adam P. Taylor
Chief Engineer, Electrical Systems
e2v
aptaylor@theiet.org

One of the many benefits of Xilinx®'s Zynq®-7000 All Programmable SoC is that it is has two ARM® Cortex™-A9 processors onboard. However, many bare-metal applications and simpler operating systems use only one of the two ARM cores in the Zynq SoC's processing system (PS), a design choice that can potentially limit system performance.

Depending upon the application in development, there could, however, be a need to have both processors running bare-metal applications, or to run different operating systems on each of the processors. For instance, one side could be performing critical calculations and hence running a bare-metal/RTOS application while the second processor is providing HMI and communications using Linux.

WHAT IS MULTIPROCESSING?

Either of those scenarios is an example of multiprocessing. Briefly defined, multiprocessing is the use of more than one processor within a system. A multiprocessing architecture can allow the execution of multiple instructions at a time, though it does not necessarily have to.

There are two kinds of multicore processing: symmetric and asymmetric.

Symmetric multiprocessing makes it possible to run a number of software tasks concurrently by distributing the load across a number of cores. Asymmetric multiprocessing (AMP) uses specialized processors or applications execution on identical processors for specific applications or tasks.

Using both of the cores on the Zynq SoC with bare metal or different operating systems is, by definition, an example of asymmetric multiprocessing. AMP on the Zynq SoC can involve any of the following combinations:

- Different operating systems on Core 0 and Core 1
- Operating system on Core 0, bare metal on Core 1 (or vice versa)
- Bare metal on both cores executing different programs

When you decide upon the need to create an AMP system on the Zynq SoC, you must consider the fact that the ARM processor cores contain a mixture of both private and shared resources that must be correctly addressed. Both processors have private L1 instruction and data caches, timers, watchdogs and interrupt controllers (with both shared and private interrupts). A number of shared resources also exist, of which common examples include I/O peripherals, on-chip memory, the interrupt controller distributor, L2 cache and system memory located within the DDR memory (see Figure 1). These private and shared resources require careful management.

Each PS core has its own interrupt controller and is capable of interrupting itself, with one or both cores using software interrupts. These interrupts are distributed by means of ARM's Distributed Interrupt Controller technology.

As the program being executed for each core will be located within the DDR memory, you must take great care to ensure that you have correctly segmented these applications.

GETTING AMPED UP

The key aspect required to get AMP up and running on the Zynq SoC is a boot loader that will look for a second executable file after loading the first application into memory. Xilinx helpfully provides an application note and source code in XAPP1079. This document comes with a modified first-stage boot loader (FSBL) and modified standalone OS, which you can use to create an AMP system.

The first thing to do is to download the ZIP file that comes with this application note before extracting the two elements—FSBL and OS—to your desired working directory. Then, you must rename the folder called SRC "design." Now, it's important to make sure the software development kit (SDK) is aware of the existence of these new files containing both a modified FSBL and a

Using software interrupts is not too different from using hardware interrupts except, of course, in how you trigger them.

modified standalone OS. Therefore, the next step is to update your SDK repository such that it is aware of their existence.

This is straightforward to achieve. Within SDK under the Xilinx tools menu, select "repositories" and then "new," navigating to the directory location <your working directory>\app1079\design\work\sdk_repo as shown in Figure 2.

COMMUNICATING BETWEEN PROCESSORS

Before creating the applications for your AMP design, you will need to consider how the applications will communicate (if they need to). The simplest method is to use the on-chip memory. The Zynq SoC has 256 kbytes of on-chip SRAM that can be accessed from one of four sources:

- From either core via the snoop control unit (SCU)
- From the programmable logic using the AXI Accelerator Coherency Port (ACP) via the SCU
- From the programmable logic using the High-Performance AXI port via the on-chip memory (OCM) interconnect
- From the central interconnect, again via the OCM

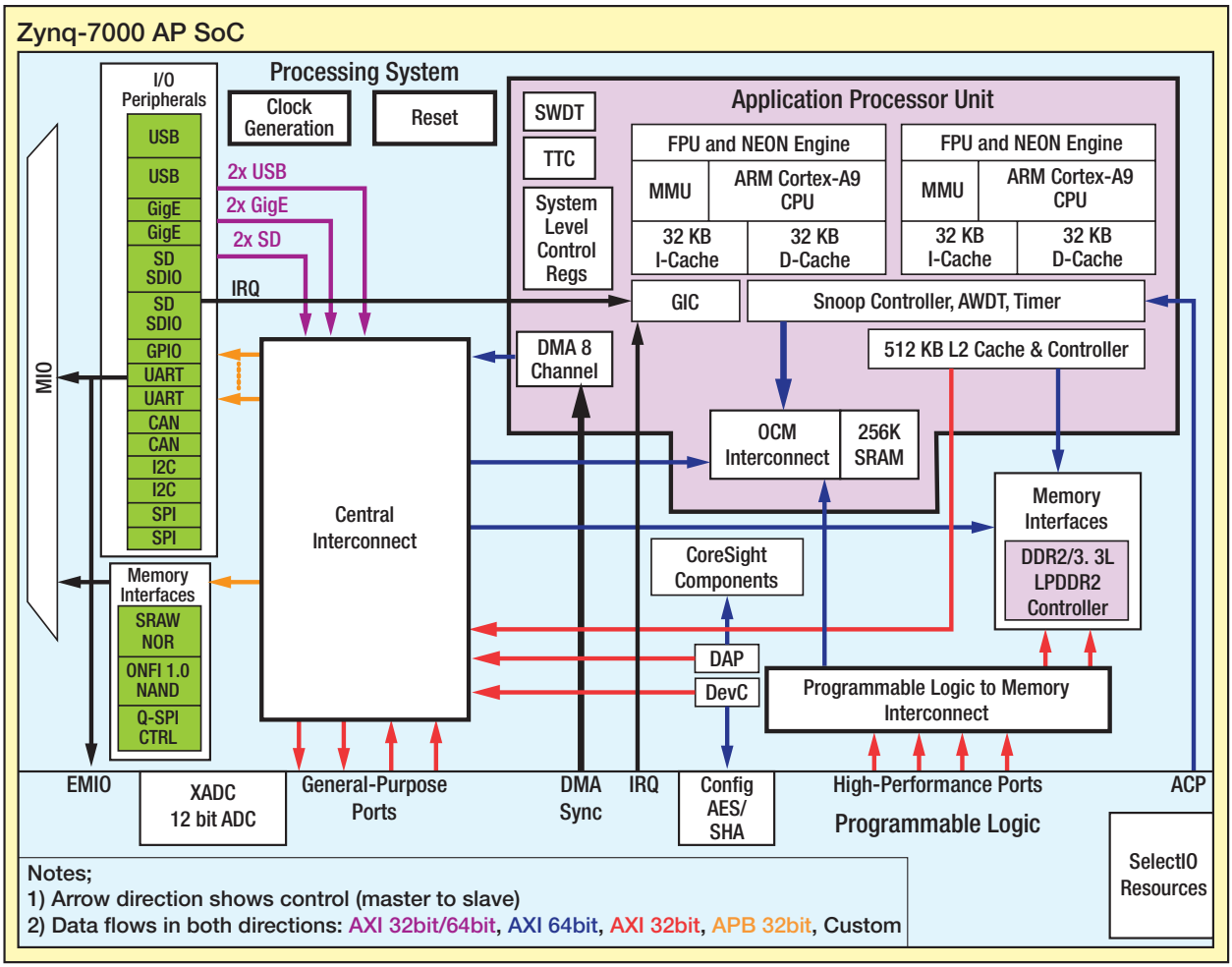


Figure 1 – The Zynq SoC processing system, showing private and shared resources

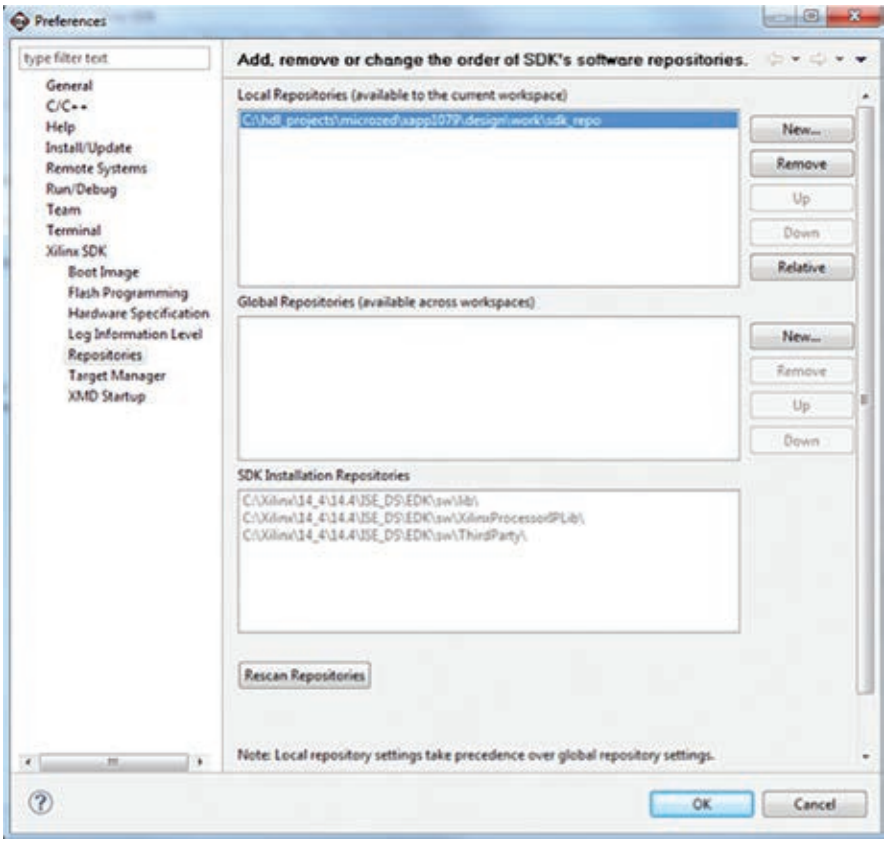


Figure 2 — Adding your new files to the repository

With these different sources that can read and write the on-chip memory, it is especially important to understand the operation of the OCM in detail before using it.

Since there are multiple sources accessing the OCM, it is only sensible that you define a form of arbitration and priority. As the lowest latency is required by the snoop control unit, which is either a processor core or an AXI ACP interface, an SCU read from these sources has the highest priority followed by the SCU write and then the OCM interconnect read and write. The user can invert the priority between the SCU write and the OCM interconnect access by setting the SCU write priority low in the on-chip memory control register.

The OCM itself is organized as 128-bit words, split into four 64-kbyte regions at different locations within the PS address space. The initial configuration has the first three 64-kbyte blocks arranged at the start of the address space and the last 64-kbyte block toward the end of the address space (Figure 5).

SIMPLE ON-CHIP MEMORY EXAMPLE

You can access the OCM using Xilinx I/O functions to read and write to and from the selected memory address. These functions, which are contained within Xil_IO.h, allow for storing and accessing 8-, 16- or 32-bit char, short or int within the CPU address space. Using these functions just requires the address you wish to access and the value you wish to store there. If it is a write, for example,

```
Xil_Out8(0xFFFF0000,0x55);
read_char = Xil_In8(0xFFFF0000);
```

A better way to use this technique to ensure the addresses are both targeting the same location within the on-chip memory, especially if different people are working on the different core programs, is to have a common header file. This file will contain macro definitions of the address of interest for that particular transfer, for instance:

```
#define LED_PAT 0xFFFF0000
```

An alternative approach is for both programs to access the memory location using a pointer. You can do this by defining the pointer, which points to a constant address, normally in C, using a macro:

```
#define LED_OP (*(volatile
  unsigned int *)(0xFFFF0000))
```

Again, you could also use another macro definition for the address to ensure that the address is common to both application programs. This approach does not then require the use of the Xilinx I/O libraries and instead allows simple access via the pointer.

INTERPROCESSOR INTERRUPTS

The Zynq SoC has 16 software-generated interrupts for each core. As noted above, each can interrupt itself, the other core or both cores. Using software interrupts is not too different from using hardware interrupts except, of course, in how you trigger them. The use of software interrupts frees the receiving application from having to poll an expected memory location for an updated value.

Within both cores, you need to configure the Generic Interrupt Controller just as you would for any hardware interrupt. See Xcell Journal issue 87, "How to Use Interrupts on the Zynq SoC," for further information.

You can then trigger a software interrupt in the updating core using the XScuGic_SoftwareIntr function provided within xscugic.h. This command will issue a software interrupt to the identified core, which can then take the appropriate action:

```
XScuGic_SoftwareIntr(<GIC
Instance Ptr>, <SW Interrupt
ID>, <CPU Mask>)
```

You must correctly segment the DDR memory for Core 0 and Core 1 applications or run the risk of one corrupting the other.

CREATING THE APPLICATIONS

Having added in the repositories, the next stage is to generate three crucial pieces of the AMP solution: the AMP first-stage boot loader, the Core 0 application and the Core 1 application. For each of these items, you will have to generate a different board support package.

The first thing you need to do is to create a new FSBL with the SDK. Selecting "file new application project" enables you to create a FSBL project that supports AMP. This is no different than the process you would follow in creating a normal FSBL. However, this time you will be selecting the "Zynq FSBL for AMP" template as shown in Figure 3.

Following the creation of the AMP FSBL, you will next create the application for the first core. Be sure to select Core 0 and your preferred operating system, and allow it to create its own BSP, as shown in Figure 4.

Having created the application, you need to correctly define the location, with DDR memory, from which it will execute. To do this, edit the linker script as in Figure 5 to show the DDR base address and size. This is important, because if you do not correctly segment the DDR memory for Core 0 and Core 1 applications, you run the risk of one inadvertently corrupting the other.

Having done this segmentation, you can now write the application you wish to execute on Core 0, as this is the core that is in charge within the AMP system. Core 0 must start the execution of the Core 1 application. You need to include the section of code seen in Figure 6 within the application. This code dis-

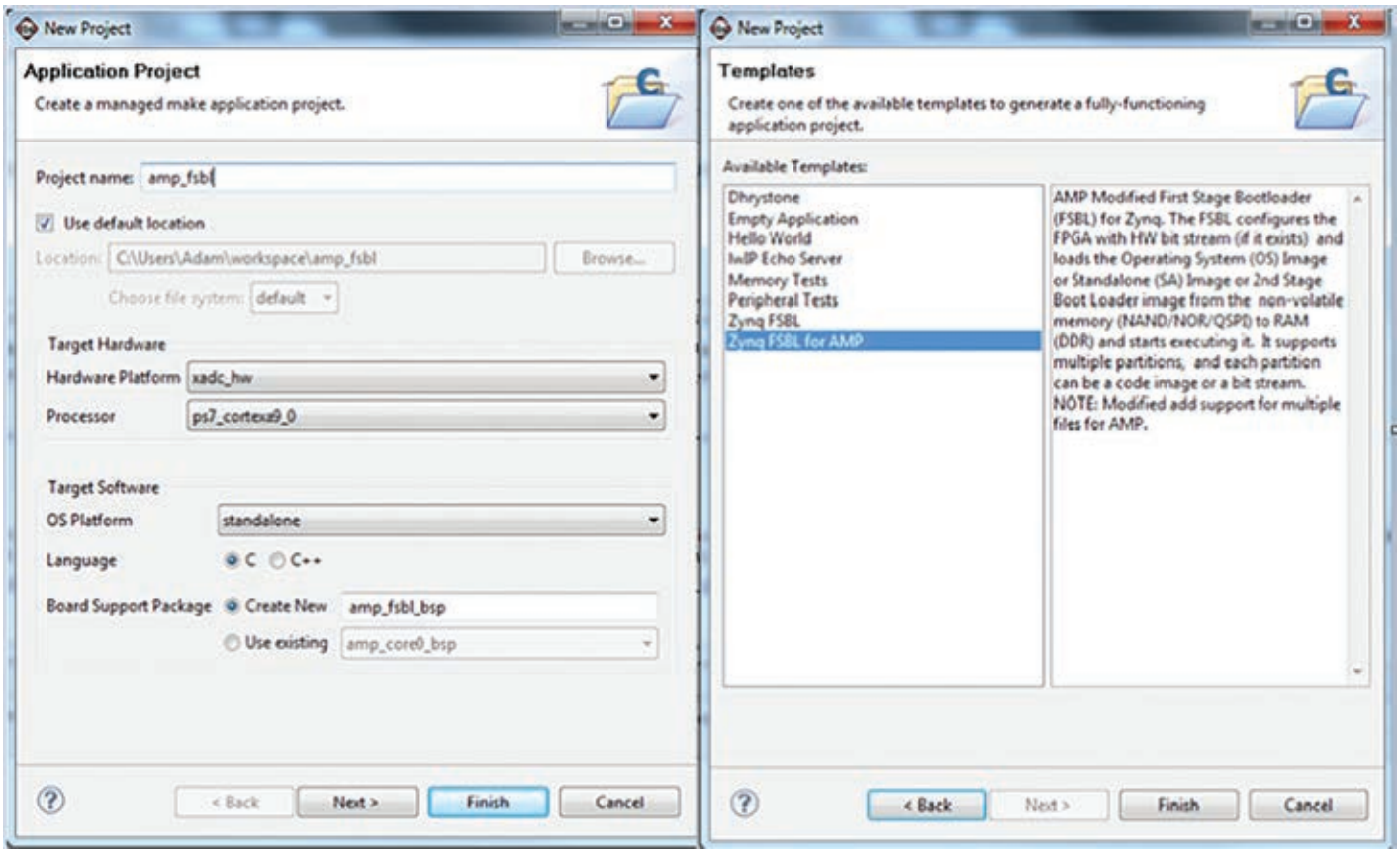


Figure 3 – Selecting the first-stage boot loader for the AMP design

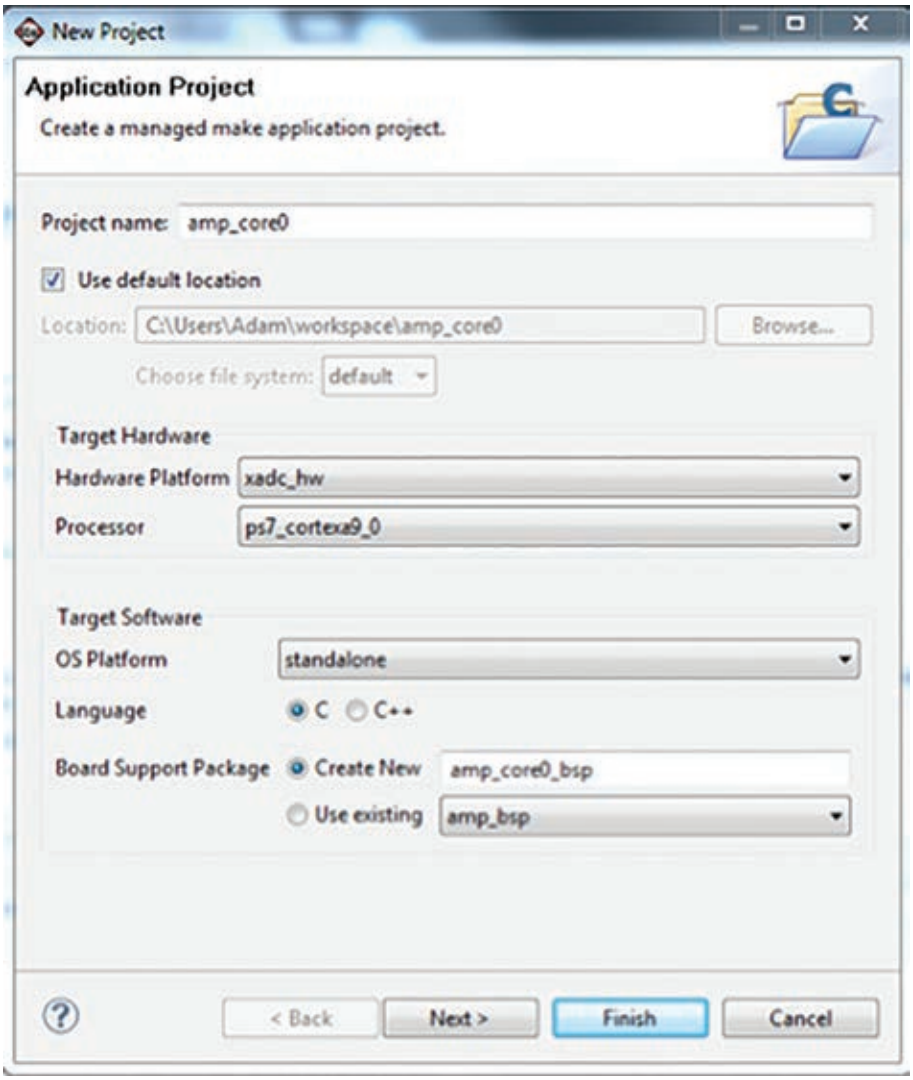


Figure 4 – Creating the application and BSP for Core 0

ables the cache on the on-chip memory and writes the start address of the Core 1 program to an address that Core 1 will access. Once Core 0 executes the Set Event (SEV) command, Core 1 will start executing its program.

The next step is to create a BSP for Core 1. It's important to use the modified standalone OS (standalone_amp, as shown in Figure 7), which prevents reinitialization of the PS snoop control unit. As such, do not allow automatic generation of the BSP while you create the project, as you did for Core 0. Be sure to select Core 1 in the CPU selection options.

Now that you have created the BSP for Core 1, you need to modify the settings of the BSP before you can progress to creating the application you want to run upon Core 1. Doing so is very simple and requires the addition of an extra compiler flag of –DUSE_AMP=1 to the configuration for the drivers section of the BSP.

With this step completed, you are free to create the application for Core 1. Be sure to select Core 1 as the processor and use the BSP you just created. Again, having created the new application, you need to once more define the correct memory locations within the DDR memory from which the Core 1 program will execute. This is achieved by editing the linker script for the application for Core 1 as you did previously. As with the first core, within this application you must likewise disable the cache on the on-chip memory, which you can use to communicate between the two processors.

PUTTING IT ALL TOGETHER

Once you have completed creation of your applications and built the projects, you should now be in possession of the following components:

- AMP FSBL ELF
- Core 0 ELF
- CORE 1 ELF
- BIT file defining the configuration of the Zynq device upon which you wish to implement AMP

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00100000	0x00100000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Figure 5 – Core 0 DDR location and size

Creating an asymmetric multiprocessing application on the Zynq SoC can be a very simple matter using the tools available.

```
#include <stdio.h>
#include "xil_io.h"
#include "xil_mmu.h"
#include "xil_exception.h"
#include "xpseudo_asm.h"
#include "xscugic.h"

#define sev() __asm__("sev")
#define CPU1STARTADR 0xffffffff0
#define COMM_VAL  (*(volatile unsigned long *)(0xFFFF0000))


int main()
{

     //Disable cache on OCM
     Xil_SetTlbAttributes(0FFFF0000,0x14de2);      // s=b1 TEX=b100 AP=bll, Domain=blll1, C=b0, B=b0
     Xil_Out32(CPU1STARTADR, 0x00200000);
     dmb(); //waits until write has finished
     sev();
```

Figure 6 – Coding to disable cache on the on-chip memory

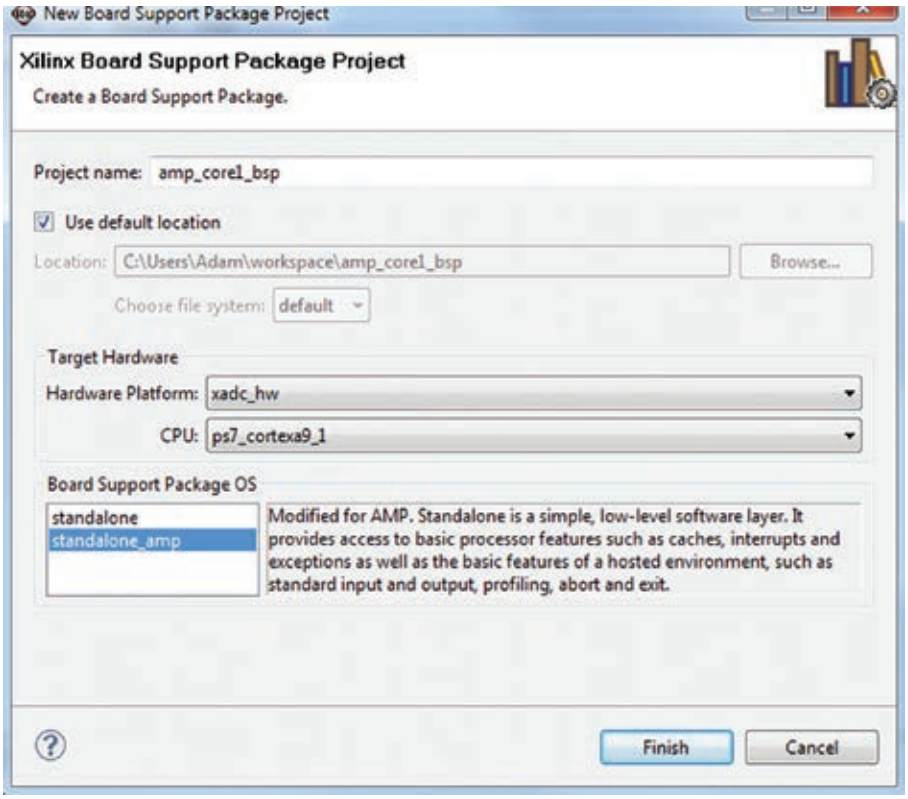


Figure 7 – Creating the BSP for Core 1

To enable the Zynq SoC to boot from your selected configuration memory, you will need a .bin file. To create it, you will also need a BIF file, which defines the files to be used to create this BIN file and the order in which they go. Rather than use the "create Zynq" boot image within the SDK, you will be using an ISE® Design Suite command prompt and BAT file provided as part of XAPP1079 under the downloaded directory\design\work\bootgen. This directory contains a BIF file and a cpu1_bootvec.bin, which is used as part of the modified FSBL to stop it looking for more applications to load.

To generate the BIN file, you need to copy the three generated ELF files to the bootgen directory and edit the BIF file to ensure the ELF names within it are correct, as shown in Figure 8.

Now you can open an ISE command prompt and navigate to the bootgen directory. There, you should run the createboot.bat. This step will create the boot.bin file as shown in Figure 9.

You can then download this file into the nonvolatile memory on your Zynq SoC. Booting the device will result in both cores starting and executing their respective programs.

Creating an asymmetric multiprocessing application on the Zynq SoC can be a very simple matter using the tools available. It's easy to achieve communication between the two cores using the on-chip memory or even a segmented DDR area.

```
the_ROM_image
{

     [bootloader] amp_fsbl.elf
                  download.bit
                    amp_cpu0.elf
                    app_cpu1.elf

  //write start vector address 0xFFFFFFF0 with 0xFFFFFFF00
  //This load address triggers fsbl to continue
  [load = 0xFFFFFFF0] cpu1_bootvec.bin
}
```

Figure 8 – Modifying the BIF file

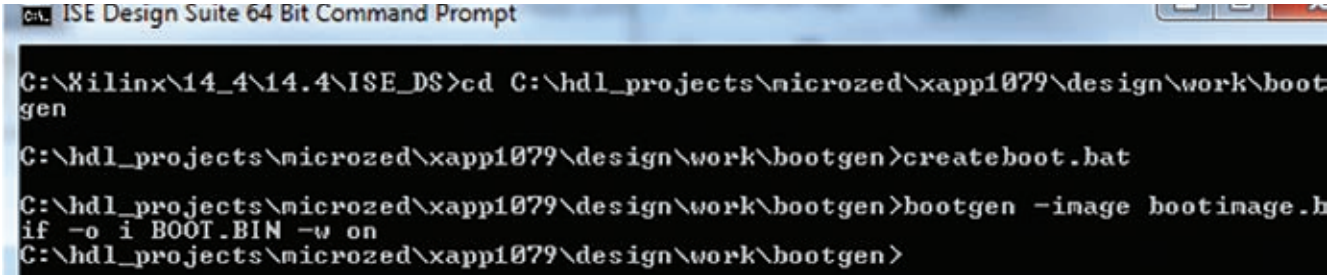


Figure 9 –The creation of the boot.bin file that will run on the Zynq SoC