UltraFAST™

UltraFast Embedded Design Methodology Guide

UG1046 (v2.1) April 22, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/22/2015	2.1	• Added Embedded Design Methodology Checklist, page 9. • Added Accessing Documentation and Training, page 10.
03/26/2015	2.0	• Added Chapter 8, SDSoC Environment. • Added Related Design Hubs, page 225.
10/20/2014	1.1	• Removed outdated information. • In Chapter 2, System Level Considerations, added information to the following sections: ◦ Performance ◦ Clocking and Reset
10/08/2014	1.0	Initial Release of document.

Table of Contents

Chapter 5: Hardware Design Flow

Chapter 6: Software Design Flow

Chapter 7: Debug

Chapter 8: SDSoC Environment

Appendix A: Additional Resources and Legal Notices

Introduction

With the introduction of an All Programmable SoC product, Xilinx provides designers a powerful way to build smarter systems quickly, effectively, and reliably. Smarter systems are typically associated with increased complexity. This is both a benefit and a challenge. It is a benefit because customers can create products that were previously impossible or incredibly difficult to build. It is a challenge because product complexity increases the importance of making good design decisions, particularly early in the product life cycle. The interplay of system software, applications, and hardware requires new ways of thinking about and solving system level problems. Xilinx has addressed the challenge by providing customers with a comprehensive tool box, including software tools, user guides, reference manuals and reference designs, to help accelerate product development with All Programmable SoCs.

A typical embedded development team consists of system architects, software engineers, and hardware engineers. Each team member often starts a design with familiar tools, and this approach has typically worked for embedded development projects in the past. However, the broad capabilities of the All Programmable SoC can cause problems for a development team that does not consider the development approach up front. To make teams more effective, Xilinx has created this methodology guide for embedded system developers. This guide complements the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 16], targeted primarily at FPGA designers.

The term *methodology* can mean different things to different people. Flow charts, methods, principles, rules, and policies are among several possible themes. This methodology guide does not illustrate a step-by-step process for success. Instead, the goal is to equip designers with information and guidance on designing embedded systems so that they can make informed decisions when using the tool box. Some content applies generally to embedded systems, while other content is specific to the Xilinx® All-Programmable SoC products. The content is a reflection of user experiences and learning gained from system development inside and outside of Xilinx. The content covers key principles, specific do's and don'ts, best practices, and avoiding pitfalls. In some topics, use cases are provided to illustrate concepts.

This guide is organized around important functional areas that map to specific skill sets within development teams. The sections are:

- Chapter 2, System Level Considerations
- Chapter 3, Hardware Design Considerations
- Chapter 4, Software Design Considerations
- Chapter 5, Hardware Design Flow
- Chapter 6, Software Design Flow
- Chapter 7, Debug
- Chapter 8, SDSoC Environment

A typical mapping of chapter relevance to team members is shown in Table 1-1. However, it is recommended that the entire team read the complete methodology guide before beginning development. It is beneficial to understand the challenges described and guidance provided in other chapters, even if the content is outside the engineer's direct area of responsibility. The line between software engineers and hardware engineers is continuing to blur, and engineers should reach beyond their primary responsibilities to effectively work with the entire team. For example, a software engineer needs to understand how the underlying hardware works, while a hardware engineer should understand the software implications of hardware decisions that are made.

Table 1-1: Chapters Relevant to Design Team Members

Book Chapters	System Architects	Hardware Designers	Software Designers
Chapter 2, System Level Considerations	X	X	X
Chapter 3, Hardware Design Considerations	X	X	
Chapter 4, Software Design Considerations	X		X
Chapter 5, Hardware Design Flow		X	
Chapter 6, Software Design Flow			X
Chapter 7, Debug		X	X
Chapter 8, SDSoC Environment			X

Typically, embedded design is done in the order listed in this guide–starting with system-level design and ending with test and debug. The chapters are written such that they can be read in any particular order. This is demonstrated in Figure 1-1, showing the interdependencies between the chapters in this guide.

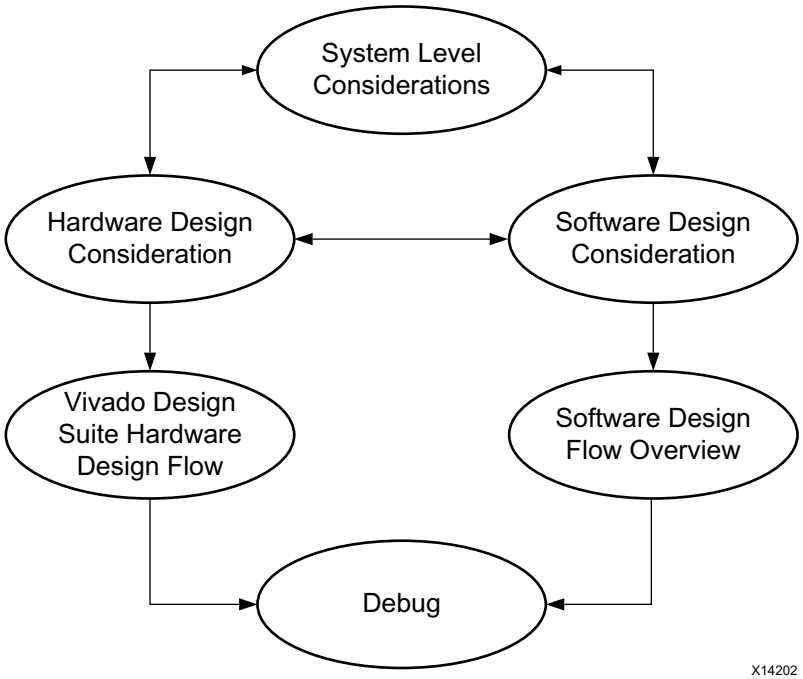

Figure 1-1: Interdependence of Methodology Guide Chapters

Note: Chapter 8, SDSoC Environment is not shown in Figure 1-1 because it is a tool that aids the user in building a system using all of the principles and methodologies described in Chapters 2-7. Additionally, Chapter 8 describes recommendations specific to the SDSoC flow.

After reading this guide, designers will be able to navigate the various tools and collateral provided by Xilinx in a more informed and effective manner. The key takeaways in each topic area should enable a designer to read the All-Programmable SoC detailed documentation with greater understanding.

Users who are familiar with the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] and the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7] will benefit from reading this guide first.

Readers unfamiliar with the Zynq®-7000 AP SoC architecture should refer to the *Zynq-7000 All Programmable SoC Overview* (DS190) [Ref 29] for more information. Overall, this guide enables an Embedded Design team to quickly assess trade-offs and avoid bottlenecks and problems, thereby enabling the team to make the right decisions for successful embedded system development.

Embedded Design Methodology Checklist

To take full advantage of the UltraFast Embedded Design Methodology, Xilinx recommends that you use this guide along with the Embedded Design Methodology Checklist. The checklist includes common questions and recommended actions to consider during the design process, starting with planning and continuing through all subsequent stages of design. The checklist questions highlight typical areas in which design decisions are likely to have downstream ramifications and draw attention to issues that are often unknown or ignored.

Xilinx recommends reading the guide first before proceeding to the checklist. Most links provide cross references to this guide and links to other Xilinx documentation. These references offer guidance on addressing the design concerns raised by the questions.

The checklist is part of the Xilinx Documentation Navigator, a free tool that you can use to access documentation while using Xilinx products. You can download Documentation Navigator as a standalone product or as part of your SDK or Vivado installation (see Using the Documentation Navigator).

To access the checklist feature, use Documentation Navigator version 2015.1 or later. From within Documentation Navigator, use these steps to begin using the Design Methodology Checklist:

1. Click the **Design Hub View** tab.
2. At the top of the menu on the left side, click **Create Design Checklist**.
3. Fill out the information in the New Design Checklist Dialog and click **OK**.
4. The new checklist opens. Tabs across the top of the checklist (see Figure 1-2) provide navigation. The Title Page tab provides some basic information on using the checklist. Click the other tabs to see the checklist questions and guidance.

Figure 1-2: **Embedded Design Methodology Checklist Tabs in Documentation Navigator**

A spreadsheet version of the Design Methodology Checklist is also available at:

http://www.xilinx.com/support/documentation/sw_manuals/xtp397-embedded-design-methodology-checklist.xlsx

Accessing Documentation and Training

Access to the right information at the right time is critical for timely design closure and overall design success. Reference guides, user guides, tutorials, and videos get you up to speed as quickly as possible with Xilinx tools. This section lists some of the sources for documentation and training.

Using the Documentation Navigator

The Xilinx embedded tools ship with the Xilinx Documentation Navigator, which provides an environment to access and manage the entire set of Xilinx software and hardware documentation, training, and support materials. Documentation Navigator allows you to view current and past Xilinx documentation. The documentation display can be filtered based on release, document type, or design task. When coupled with a search capability, you can quickly find the right information.

Documentation Navigator scans the Xilinx website to detect and provide documentation updates. The Update Catalog feature alerts you to available updates, and gives details about the documents that are involved. Xilinx recommends that you always update the catalog when alerted to keep it current. You can establish and manage local documentation catalogs with specified documents.

The Documentation Navigator has a tab called the *Design Hub View*. Design hubs are collections of documentation related by design activity, such as Zynq-7000 Design Overview, PetaLinux Tools, and SDK. Documents and videos are organized in each hub in order to simplify the learning curve for that area. Each hub contains an Embedded Processor Design section, a Design Resources section, and a list of support resources. For new users, the Embedded Processor Design section provides a good place to start.

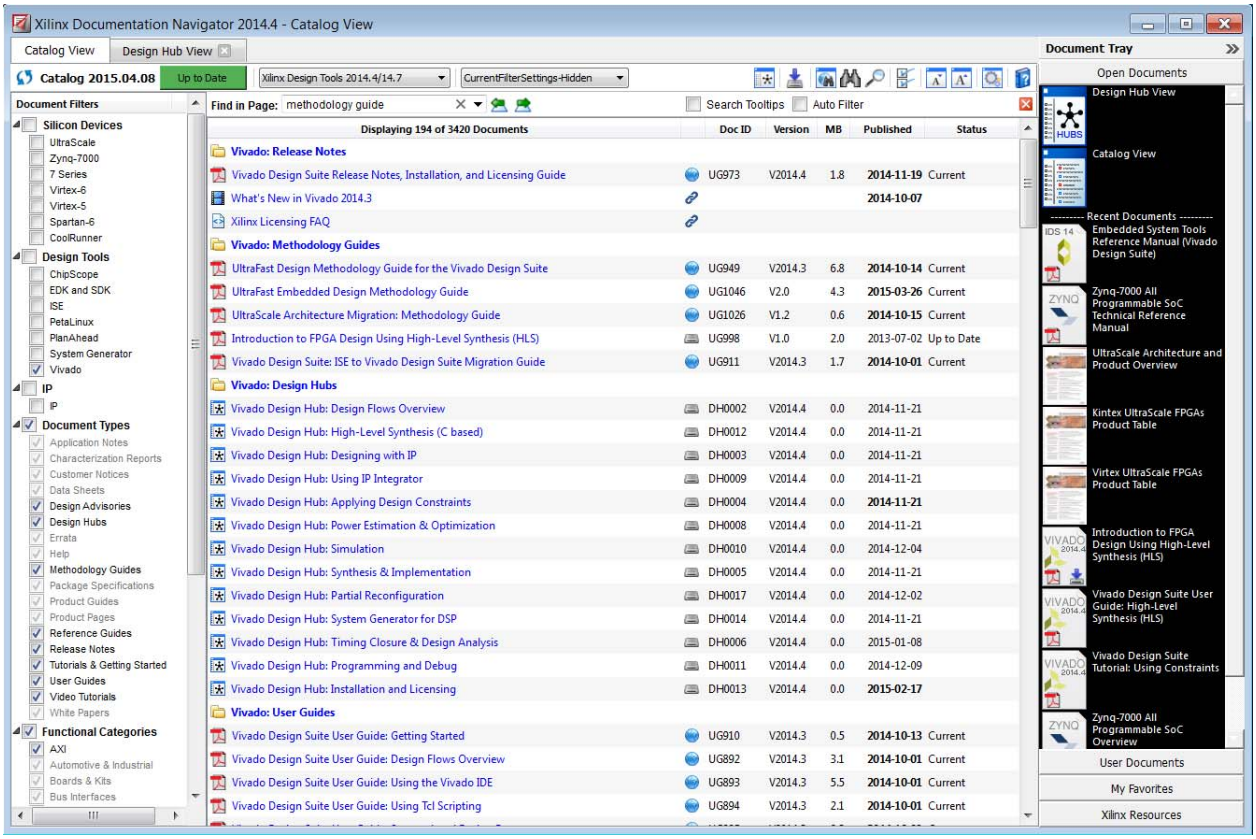

Figure 1-3: **Xilinx Documentation Navigator, Catalog Viewer**

Accessing the QuickTake Video Tutorials

Xilinx QuickTake video tutorials provide guidance on using the features of Xilinx tools, including SDK, SDSoC, and PetaLinux Tools. These tutorials are short and succinct training tools. They can be viewed from the Video Tutorials page on xilinx.com or the Xilinx YouTube channel and can be downloaded locally.

TIP: *Download the clips locally if connection speed interferes with viewing quality. The QuickTake video tutorials are also available through Documentation Navigator.*

System Level Considerations

This chapter covers the following system-level design issues to be considered when designing with a Zynq®-7000 AP SoC:

- Performance, page 13: The target application generally determines the overall system performance. System performance objectives are allocated between hardware and software, and further allocated to sub-components.
- Power Consumption, page 18: System performance is a primary driver of power consumption. The design team is often faced with difficult trade-off choices to make between the two issues. This section describes design considerations for optimizing power consumption on a Zynq-7000 AP SoC.
- Clocking and Reset, page 36: It is important to understand what clocking resources are available so that you can plan how to best use those resources. Similarly, the reset system has many different sources and it is important that you understand how they affect the different reset destinations.
- Interrupts, page 41: The system-level interrupt environment provides a comprehensive set of capabilities for you to prioritize hardware and software resources in your application.
- Embedded Device Security, page 45: The security level required from application to application can vary greatly. Understanding the various security features of the Zynq-7000 AP SoC will help you implement the right level of security for your application.
- Profiling and Partitioning, page 51: An important system-level decision you make is how to partition the functions of an application between hardware and software. Profiling tools help in making those decisions.

Performance

With a Zynq-7000 AP SoC device, system performance depends on the final application goals. For example, a real-time application's system performance might depend on the interrupt service routine latency, while a video application's system performance might depend on maintaining 60 frames per second over an off-chip interface. In this section, the division of performance goals across different design team members is described, followed by considerations for implementing those performance goals through data movement and computation design choices. The section concludes with a discussion of Zynq-7000 AP SoC device monitoring options, allowing designers to build custom performance monitoring capabilities in software and programmable logic (PL).

System Performance Design Goals

System performance goals are divided across the different engineering disciplines on a Zynq-7000 AP SoC device design team. The three engineering disciplines discussed in this document are hardware, software, and system.

Hardware Engineer

Hardware engineers working with Zynq devices implement designs using a mixture of PL components, AXI-connected IP, high speed off-chip interfaces, and custom logic. PL choices are driven by performance requirements, such as throughput and latency, but can also be driven by the performance constraints of system software and hardware interactions. Hardware engineers must consider interactions with the processing system (PS), because data movement and synchronization can have a large effect on PL throughput and latencies. PL data movement and monitoring points can be used to guide design decisions and are discussed later in this section. Traditional PL metrics, such as maximum frequency and resource utilization, are not covered. Refer to the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 16] for more information.

Software Engineer

Software engineers focus on system software running within the PS and its interaction with memory, I/O, and PL. For example, user software that communicates with PL has multiple communication options, each with advantages and disadvantages. The software performance monitoring capabilities unique to Zynq devices can be used by software engineers to tune performance and are described later in this section. Also, the rich ecosystem of existing ARM performance profiling and monitoring tools can be used with the dual-core Cortex-A9 processors for optimizing performance. For more information on the performance monitoring capabilities, refer to the ARM DS-5 Development Studio Streamline Performance Analyzer documentation [Ref 72].

System Engineer

Performance goals for designs implemented using Zynq devices can be partitioned across hardware and software. This partitioning should be considered throughout all design stages. During the initial stages, hardware and software engineers can work relatively independently, but early performance estimates are needed to set realistic performance goals and make initial partition choices. The system engineer must consider all hardware and software performance bottlenecks and make trade-offs. The system architect can make early performance estimates of data and communication paths, and fine tune them later using the system performance-monitoring points and tools.

All design team members generally consider the impact on performance of data movement and data computation. The ability to monitor system events is useful in designing and optimizing performance of a Zynq device. These considerations are described in the following sections, including suggested design methodologies to help guide Zynq device design flows.

System Data Movement

Moving data through a system is a common system-level performance problem. A Zynq device has several AXI masters that can drive transactions either directly or with assistance from DMAs. This section describes the various options and trade-offs for addressing data movement in a Zynq device.

The ARM CPUs can move data using direct memory transfers, such as memcpy. Such transfers are useful for small transfers of 4 KB or less, while larger transfers benefit from DMA assistance. The data's source and destination buffer locations should also be considered. For example, a PS DMA data transfer to PL will typically go through the 32-bit master GP ports. The *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] contains techniques for using the PS DMA controller. The PS DMA controller runs its own microcode. An alternative method would use a DMA in PL to move data across the 64-bit ACP or HP ports, which is a higher-performing option and uses PL resources.

Other AXI masters within the PS should be considered when determining system data movement performance. IOP DMAs exist in several IP blocks: the GigE controller, SDIO controller, USB controller, and device configuration interface (DevC). The IP block functions are described in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4], and drivers are provided for their use in operating systems such as Linux and stand-alone designs. The GigE controller performance characteristics are further described in *PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 AP SoC* (XAPP1082) [Ref 38]. The remaining cores have driver layers that provide additional function, but their performance characteristics are not described here.

From the PL, AXI masters can drive data movement on the ACP, HP, or slave GP ports. These ports and associated performance are described further in the following sections:

- ACP and Cache Coherency, page 116
- PL High-Performance Port Access, page 120
- GPs and Direct PL Access from APU, page 133

DMAs attached to these ports are best used to translate between AXI4 Memory-Mapped and AXI4-Streaming interfaces, providing a clean separation of memory-mapped accesses to data-centric processing (such as video pipelines). DMA is not always the preferred method, however. If the streaming interface performance is low and not a concern, then the AXI4-Stream FIFO core is a simple way to source and synchronize an AXI-Stream interface from a processor. User applications can have custom AXI masters that do direct data movement using standard AXI Memory-Mapped transactions. The throughput and latency performance characteristics of such custom data transfers may end up being very similar to standard PL DMA data transfers. Performance comparisons can be made using the counters and timers described in System Monitoring, page 17.

DMA moves data from a source to a destination location. Memory is often used as data buffers to match the differences in rate of data source, processing, or data sink, therefore ensuring that the processing stage can achieve maximum throughput.

Off-chip data buffer location can be implemented using off-chip memory attached to customizable MIO or EMIO pins. The memory characteristics affect the performance of moving large buffers, such as file systems on SD cards or network-attached storage over the GigE controller.

For on-chip buffering, the OCM, L2 cache, and DDR controller are the three main sources of sharable buffer space within the PS. The L2 cache and DDR controller provide excellent buffer-access latency for sharing data between the processor and ACP port. Only the ACP can access the L2 cache from PL. For high-bandwidth accesses to DDR, the HP ports are better suited than ACP. The OCM can be used by software applications as a 256 KB scratchpad accessible by all masters in the PL. A benefit to using OCM is its excellent random-access latency, whereas the L2 cache and DDR memory benefit from memory-access locality.

System Computation

Computation can be done in either the ARM cores or within the PL. Typical Zynq devices have the control plane written in software executing on the ARM cores. Data-centric computation is done in the PL using a mix of existing and custom IP. In high-performance situations such as line-rate packet processing, computation and control can be moved closer to the data. This typically requires custom PL IP to manage data flow.

Moving computations from software into the PL is an important benefit of the Zynq-7000 AP SoC platform. Not all software can be moved to the PL, in particular pre-compiled code, complex library routines, and OS services. If it is possible to move software components to PL, performance metrics such as the full application runtime should consider the trade-off between data movement costs and acceleration benefits. This cost and benefit analysis can be done using software profiling tools to determine potential speedups according to Amdahl's law:

$$S = \frac{1}{(1-\alpha) + \alpha/p}$$

In the equation above, *S* is the overall performance improvement and α is the percentage of the algorithm that can be sped up with hardware acceleration. Therefore 1-α is the percentage of the algorithm that cannot be improved. The variable *p* is the speedup due to acceleration. For example, to accelerate a computational algorithm implemented entirely in software, you can use profiling tools such as TCF profiler to help identify the particular function that is frequently used. The percentage of time with which this function is used corresponds to a high α number. The function can then be sped up by implementing it in hardware. The speed up of the function maps to *p*. Dividing a frequently used function by a large multiplier will result in the most improvement in performance in the computation of the algorithm due to hardware acceleration.

The speed up of a particular function has two components: the transfer of data to and from the acceleration block and the actual computation performed by the accelerator. When a DMA is used to perform the data transfer, the overhead for setting up and managing the DMA must be taken into account when measuring the multiplier factor, *p*. If the overhead is large, it can bound the value of *p* and may result in a low *p* factor, and therefore low improvement in performance. For more information, refer to Profiling and Partitioning, page 51.

High-level synthesis provides an excellent way of moving software components to PL for exploring acceleration options. If candidate software is not a good fit for high-level synthesis, Xilinx also provides a rich library of programmable logic IP with corresponding drivers which can replace functions implemented in software to improve overall system performance.

System Monitoring

A rich tool ecosystem exists for monitoring the ARM processors. In a Zynq device, full system-level performance monitoring also uses blocks available in the PS and PL. These are:

- **SCU Global Timer (PS).** The SCU global timer can be used to timestamp system events in a single clock domain. Alternatively, operating systems often provide high accuracy timers for software event tracing, such as Linux clock_nanosleep.
- **ARM Performance Monitoring Units (PS).** Each ARM core has a performance monitoring unit (PMU) that is used to count micro-architectural events. These counters can be accessed directly by software, through operating system utilities, or with chip debuggers such as Linux Perf or ARM Streamline. The counters are viewable in the Performance View of SDK 2014.2. See the Performance Monitoring Unit section in the ARM *Cortex-A9 Technical Reference Manual* [Ref 75] for more information.
- **L2 Cache Event Counters (PS).** The L2 cache has event counters that can be accessed to measure cache performance. The counters are viewable in the Performance Counters View of SDK. See the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.
- **GigE Controller (PS).** The gigabit Ethernet controller has statistical counters to track bytes received and transmitted on its interface. See the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.
- **AXI Performance Monitor (PL).** This core can be added in PL to monitor AXI performance metrics such as throughput and latency. Trace functions enable time-stamped AXI traces, such as time-stamped start and end of AXI transactions to observe per-transaction latency. See the AXI Performance Monitor web page [Ref 43] for more information.
- **AXI Timer (PL).** This core can be added in PL to provide a free-running timer in PL. This timer is useful for time-stamping events in PL clock domains. See the AXI Timer/Counter web page [Ref 44] for more information.
- **AXI Traffic Generator (PL).** This core can generate a variety of traffic patterns to the PS interfaces. When used with an AXI performance monitor, the traffic generator can help provide early system-level performance estimates. The core can be used to estimate data-movement costs and validate design partition choices. For more information, refer to the LogiCORE™ AXI Traffic Generator web page [Ref 45].

These monitor blocks provide visibility into the full system performance behavior. Event counters and PL monitoring blocks can be customized for specific design performance goals or used to get a high-level view of system performance. Early use of built-in performance monitoring allows performance feedback throughout the design cycle, resulting in data-driven system architecture decisions.

There are different ways of exercising or acquiring monitoring data from the various blocks. Most peripherals come with an example application packaged with the corresponding bare-metal driver shipped with Xilinx SDK. A list of available Linux drivers is provided in the Xilinx Linux Drivers wiki page [Ref 50].

Xilinx SDK provides built-in System Performance Modeling and Analysis functionality. Performance monitors gather data from the system, either live data or data created by traffic generators which are used to model real world transactions. This data is displayed using the Xilinx SDK visualization tools with statistics for number of transactions, bandwidth, and latency from an APM (profile mode) connected to the HP and ACP ports. For more information, refer to Xilinx SDK System Debugger, page 53.

Power Consumption

The power consumption of Zynq-7000 AP SoCs is an important consideration for system architects and board designers. Power consumption is a critical concern for most applications, and some applications specify maximum power per card or per system. Thus, designers must consider power consumption early in the design process, often starting with device selection.

Reducing SoC power consumption can improve board design by lowering supply-rail power, simplifying power supply design and thermal management, and easing the requirements on the power distribution planes. Low power also contributes to longer battery life and higher reliability, because cooler-running systems last longer.

Reducing system power consumption requires a comprehensive and focused approach to achieve optimal results. This section covers several aspects of Zynq-7000 AP SoC power consumption, including PS and PL architecture and features, the power components associated with PL, and the process technology. It also covers power dissipation, and traditional methods of estimating and measuring power.

Power Challenges

According to Moore's Law, transistor size decreases with each process technology generation. As size decreases the amount of current each transistor leaks increases, causing an increase in static power consumption, which is the amount of current the device draws when not switching. Increasing SoC performance requires higher-frequency clocks, resulting in increased dynamic power consumption. Thus, static power is driven by transistor leakage current and dynamic power is driven by the transistor switching frequency. Compounding the problem, decreasing transistor sizes allow more transistors to be packed on FPGAs with each product generation. More transistors results in more leakage current and more transistors switching at higher clock frequencies on each FPGA device.

These issues require designers to address power supply and thermal-management issues earlier in the design cycle. Using a heat sink on a device may not adequately resolve these issues. Instead, designers must look for opportunities to reduce the impact of the design logic.

Figure 2-1 illustrates actions that can be taken at various points in the design cycle to reduce power consumption. Addressing power issues early in the design process yields the greatest benefits.

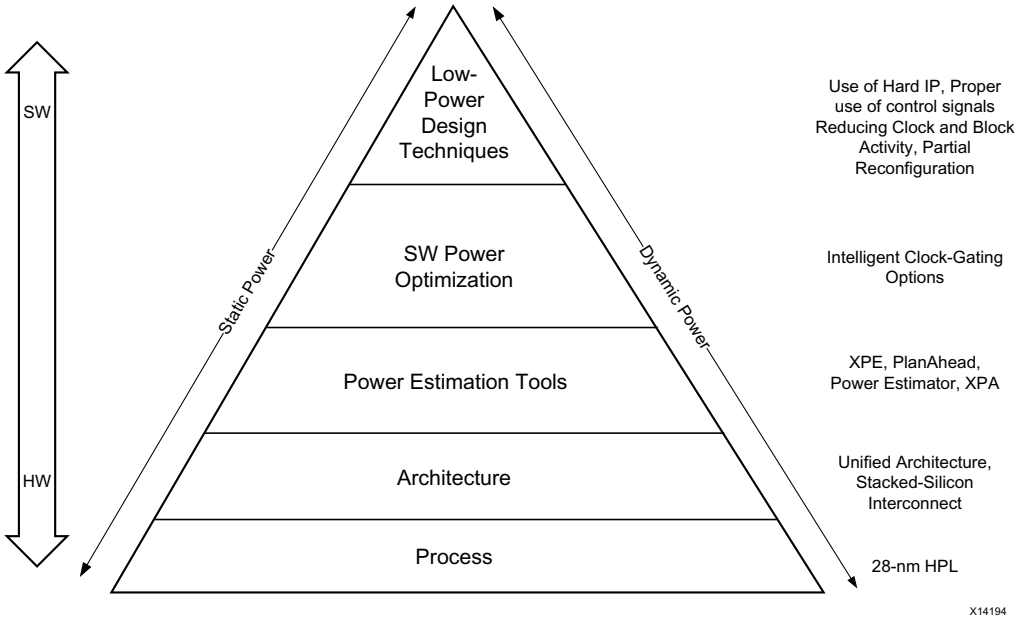

Figure 2-1: Managing Power Issues Throughout the Design Cycle

Power and Signaling

Zynq-7000 AP SoC devices are divided into several power domains, as illustrated in Figure 2-2.

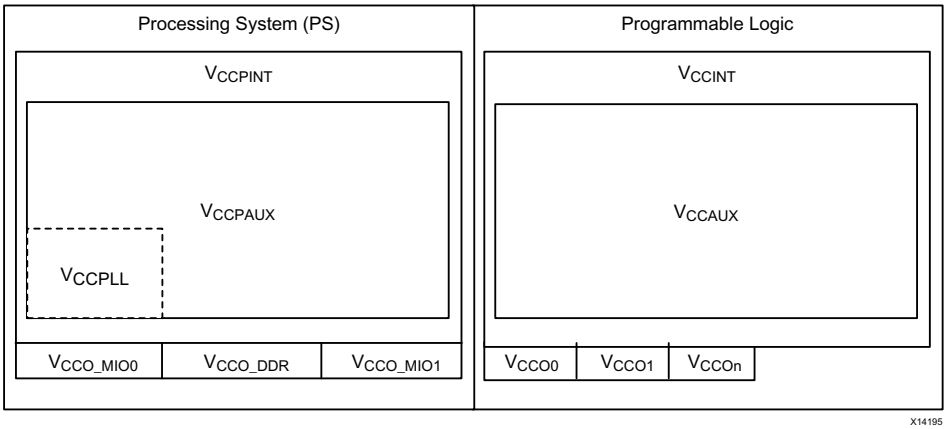

Figure 2-2: Zynq-7000 AP SoC Power Domains

The PS and PL power supplies are independent; however, the PS power supply must be present when the PL power supply is active. The PL can be powered off in applications that do not require the PL. The PS and PL power pins are summarized in Table 2-1. The voltage sequencing and electrical specifications are described in *Zynq-7000 All Programmable SoC (Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics* (DS187) [Ref 28].

Table 2-1: **Power Pins**

Type	Pin Name	Nominal Voltage	Power Pin Description
PS Power	V_{CCPINT}	1.0V	Internal logic
	V_{CCPAUX}	1.8V	I/O buffer pre-driver
	V_{CCO_DDR}	1.2V to 1.8V	DDR memory interface
	V_{CCO_MIO0}	1.8V to 3.3V	MIO bank 0, pins 0:15
	V_{CCO_MIO1}	1.8V to 3.3V	MIO bank 1, pins 16:53
	V_{CCPLL}	1.8V	Three PLL clocks, analog
PL Power	V_{CCINT}	1.0V	Internal core logic
	V_{CCAUX}	1.8V	I/O buffer pre-driver
	$V_{CCO\_\#}$	1.8V to 3.3V	I/O buffers drivers (per bank)
	V_{CC_BATT}	1.5V	PL decryption key memory backup
	V_{CCBRAM}	1.0V	PL block RAM
	$V_{CCAUX\_IO\_G\#}$	1.8V to 2.0V	PL auxiliary I/O circuits
XADC	VCCADC	1.8V	Analog power and ground.
Ground	GND	Ground	Digital and analog grounds

PS Power Domains

For more information on Zynq-7000 AP SoC PS power domains, refer to Chapter 5: Processing System (PS) Power and signaling of *Zynq-7000 All Programmable SoC PCB Design Guide* (UG933) [Ref 14].

PL Power Domains

Multiple power supplies are required to power the different PL resources in a Zynq-7000 AP SoC. The different resources operate at different voltage levels for increased performance or signal strength while preserving improved immunity to noise and parasitic effects.

Table 2-2, page 21 lists the power sources typically used by PL resources available in Zynq-7000 AP SoC. This table is provided only as a guide because the specifics can vary across Zynq-7000 AP SoC families.

Table 2-2: PL Power Sources

Power Supply	Resources Powered
V_{CCINT} & V_{CCBRAM}	• All CLB resources • All routing resources • Entire clock tree, including all clock buffers • Block RAM/FIFO • DSP slices • All input buffers • Logic elements in the IOB (ILOGIC/OLOGIC) • ISERDES/OSERDES • Tri-Mode Ethernet MAC • Clock Managers (DCM, PLL, etc.)(minor) • PCIE and PCS portion of MGTs
V_{CCAUX} & V_{CCAUX_IO}	• Clock Managers (MMCM, PLL, DCM, etc.) • IODELAY/IDELAYCTRL • All output buffers • Differential Input buffers • V_{REF}-based, single-ended I/O standards, such as HSTL18_I • Phaser
V_{CCO}	• All output buffers • Some input buffers • Digitally Controlled Impedance (DCI) circuits, also referred to as On-Chip Termination (OCT)
MGT	• PMA circuits of transceivers

Board-Level Power-Distribution System

On a printed circuit board, the power distribution system (PDS) distributes power from the source to the various chips and devices requiring power. Although the PDS design can vary from simple to complex, there are three primary requirements it must satisfy:

- The PDS must deliver a well-regulated voltage. Power regulation is done at the regulated power-supply circuit, supported by one or more bulk capacitors and LC filtering circuits.
- The PDS must be stable at all points on the board under all current loading conditions. Stability under all loading conditions has two sub-requirements:
  - The distribution system must have low resistance and inductance. This usually requires a system of power and return planes for distribution, and low inductance pads and vias at the various device connections.
  - The charge must be available where and when needed. The required charge is typically stored in bypass capacitors that are placed around the board, and to a lesser extent from distributed planar capacitance.
- The PDS must be quiet, meaning that when devices switch, noise is not generated that interferes with other devices or generates EMI.

For more information on power distribution systems for Zynq devices, refer to "Chapter 3: Power Distribution System" in the *Zynq-7000 All Programmable SoC PCB Design Guide* (UG933) [Ref 14].

Power Management

Using a Zynq-7000 AP SoC can help reduce a system's static power consumption. The Zynq-7000 AP SoC PS is an optimized silicon element consisting of dual-core ARM Cortex-A9 CPUs plus integrated peripherals. The PL is based on the Xilinx® 7-Series architecture built on the 28nm high-performance, low-power (HPL) process, providing high-performance operation while enabling significant power reduction. Choosing a device built on the HPL process eliminates the need for complex and expensive static-power-management schemes.

There are different ways to reduce system power. The following sections provide tips that can be used to optimize a design to meet system power requirements.

PS Power Management

This section describes the design considerations needed to optimize power consumption of a Zynq-7000 AP SoC PS. This includes power management of the APU unit, PS peripherals, clocks and PLLs, caches, SCU, and OCM. It is assumed that the designer understands the impact of power management trade-offs on the overall system.

System Design Considerations

The PS components can be power managed as follows:

- **Application Processing Unit (APU).** The Zynq-7000 AP SoC APU supports dynamic clock gating. This feature can be enabled using the CP15 power control register. If enabled, the clocks to several CPU internal blocks are dynamically disabled during idle periods. The gated blocks are:
  - Integer core
  - System control block
  - Data engine

  By reducing the processor core voltages and operating frequency, as much as a two-fold decrease in power consumption can be realized. Refer to the power-management section of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

- **PS Peripherals.** The PS supports several clock domains, each with independent clock-gating control. When the system is in run mode, the user can shut down unused clock domains to reduce dynamic power dissipation. Clocks for PS peripherals such as timers, DMA, SPI, QSPI, SDIO, and the DDR controller can be independently gated to save power. Refer to Chapter 25, Clocks, in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information on the system clocks and how they can be controlled using dividers, gates, and multiplexers.
- **Caches.** The L2 cache controller supports the following dynamic-power reduction features. These features are controlled by a corresponding enable bit in the l2cpl310.reg15_power_ctrl register.
  - **Dynamic Clock Gating.** When the dynamic high-level clock-gating feature is enabled, the cache controller clock stops when the controller is idle. The clock-gating feature waits several cycles after the controller is idle before it stops the clock.
  - **Standby Mode.** The L2 cache controller standby mode can be used with the processor's wait-for-interrupt (WFI) mode that drives the L2 cache controller. When a processor is in WFI mode and standby mode is enabled, the L2 cache controller internal clocks are stopped. Refer to Chapter 3, Application Processing Unit, in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information on using WFI.

  The dynamic clock gating feature is a superset of the standby mode. In standby mode, clock gating is limited to WFI states, thus making L2 cache accesses more predictable under normal run conditions.
- **On-Chip Memory (OCM).** In general, OCM can be used to reduce overall power during low-power modes such as Linux standby mode. For example, OCM can be used to store executable code when the DDR is in low power mode.
- **Snoop Control Unit (SCU).** The SCU has a standby mode that is enabled by setting the corresponding bit in the mpcore.SCU_CONTROL_REGISTER. When enabled, the internal SCU clocks are stopped when the following conditions are met:
  - The CPUs are in WFI mode.
  - There are no pending requests on the ACP.
  - There is no remaining activity in the SCU.

  The SCU resumes normal operation when a CPU leaves WFI mode or a request on the ACP occurs.

- **PLL.** PLL power consumption is dependent on the PLL output frequency, thus power consumption can be reduced by using a lower PLL output frequency. Power can also be reduced by powering down unused PLLs. For example, if all clock generators can be driven by the DDR PLL, then the ARM core and I/O PLLs can be disabled to reduce power consumption. The DDR PLL is the only unit that can drive all of the clock generators. Each clock can be individually disabled when not in use. In some cases, individual subsystems contain additional clock disable capabilities and other power management features.
- **Physical Memory.** Zynq-7000 AP SoCs support different types of physical memory, such as DDR2, DDR3, and LPDDR2. The supported DDR memory types can operate with both 16-bit and 32-bit data. DDR power consumption can be a significant component of total power, so minimizing DDR power consumption is an important way of reducing system power. Items to consider when reducing DDR power consumption include:
  - The DDR controller operating speed.
  - The choice of DDR width and whether ECC is enabled or disabled.
  - The number of DDR chips used.
  - The DDR type, such as using LPDDR for significant voltage reductions.
  - The use of different DDR modes during low power operation, such as DDR self-refresh mode. Refer to the appropriate DDR standards for more information about power consumption during DDR low-power operating modes.

  Refer to the "Clocks" chapter in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information on implementing DDR controller clock gating.
- **I/O.** I/O devices such as MIOs and DDR IOs also contribute to overall power. Refer to the "SelectIO Signaling" chapter of the *Zynq-7000 All Programmable SoC PCB Design Guide* (UG933) [Ref 14] for more information on I/O buffer control power management.

Software Support

The Linux kernel supports the following power management states:

- S0: Freeze or low-power idle. This is a generic, pure software, light-weight, low-power state.
- S1: Standby or power-on suspend. All processor caches are flushed and instruction execution stops. Power to the processor and RAM is maintained.
- S3: Suspend-to-RAM. System and device state is saved to memory. All devices are suspended and powered off. RAM remains powered.

Refer to the Zynq Power Management wiki page [Ref 64] for more information about Linux power management support for Zynq-7000 AP SoCs.

The link also provides information about the CPU-scaling framework implemented for Zynq-7000 AP SoCs. The CPU-scaling framework is used to scale CPU frequency at run time. For applications that do not require high processing performance, the CPU frequency can be reduced to meet application needs. A lower clock frequency can significantly reduce the operating power when compared to operating at a higher frequency.

PL Power Management

The PL can be powered off in applications that do not require the PL. To do this, independently-connected power supplies are needed for the PS and PL. The PL supplies that can be powered off include VCCINT, VCCAUX, VCCBRAM, and VCCO. Refer to the appropriate data sheet to determine the correct power sequencing.

The configuration is lost when the PL is powered down and must be reconfigured when it is powered on again. Software should determine when it is safe to power down the PL.

This section describes design considerations for optimizing PL power consumption on a Zynq-7000 AP SoC.

Logic Resource Utilization

PL resource utilization is an important contributor to the total power consumed by Zynq-7000 AP SoCs. The amount of CLB resources, dedicated hardware, and routing used is design dependent and adds to both static and dynamic power consumed by the PL. A thorough understanding of the PL architecture enables the designer to leverage silicon resources.

To reduce power, designers must look for opportunities to reduce the logic in a design. This allows use of smaller devices and reduces static power consumption. One option is to use dedicated hardware blocks rather than implementing the function in CLBs. This can help lower both static and dynamic power consumption and make it easier to meet timing requirements. Blocks lower static power consumption because the total transistor count is less than an equivalent component built using CLB logic.

Designers can use the IP catalog to customize the dedicated hardware for instantiating a specific resource. Unused PS IP can be re-purposed for other tasks that may not be obvious. For example, DSP48 slices have many logic functions, such as multipliers, adders and accumulators, wide logic comparators, shifters, pattern matchers, and counters. Block RAMs can be used as state machines, math functions, and ROMs.

Most of the coding techniques needed are described in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 16].

Managing Control Sets

Control signals (signals that control synchronous elements such as a clock, set, reset, and clock enable) can affect device density, utilization, and performance. Some guidelines follow for keeping the power impact of these signals to a minimum.

Avoid using both a set and reset on a register or latch. The flip-flops in Xilinx FPGAs can support both asynchronous and synchronous reset and set controls. However, the underlying flip-flop can natively implement only one set, reset, preset, or clear at a time. Specifying more than one of these functions in the RTL code results in the implementation of one condition using the SR port of the flip-flop and the other condition implemented in the PL, thus using more PL resources.

If one of the conditions is synchronous and the other is asynchronous, the asynchronous condition is the one implemented using the SR port, and the synchronous condition is implemented in the PL. In general, avoid more than one set, reset, preset, or clear condition. Also, only one attribute for each group of four flip-flops in a slice determines whether the SR ports of flip-flops are synchronous or asynchronous.

Use active-high control signals, because the control ports on registers are active high. Active-low signals use more lookup tables because they require an inversion before they drive the register control port. The LUT may already have other inputs such that the inversion could require another LUT. Using active-low control signals can lead to longer implementation runtimes and result in poor device utilization, affecting timing and power. Therefore, active-low resets in an FPGA design are not recommended.

Use active-high control signals where possible in the HDL code or instantiated components. When it's impossible to specify control signal polarity within the design, invert the signal in the top-level code hierarchy. The I/O logic can absorb the inferred inverter without using additional FPGA logic or routing, resulting in better utilization, performance, and power.

Managing Sets and Resets

Coding unnecessary sets and resets can prevent the inference of shift-register LUTs (SRLs), LUT RAMs, block RAMs, and other logic structures. Although coding can be awkward, many circuits can be made to self-reset, or simply do not need a reset. For example, considering flip-flops within a data pipeline, there is little point in having a reset at all. After a few cycles, the entire pipeline is operational, and any incorrect data is flushed out of the system.

Reducing the use of sets and resets improves device utilization, resulting in better placement, improved performance, and reduced power.

Refer to the *Get Smart about Reset: Think Local, Not Global Whitepaper* (WP272) [Ref 32] for more information on designing resets.

Clock Gating

The PL dynamic power consumption is determined by the operating clock frequency (fclk), node capacitance (C), FPGA operating voltage (V), and the switching activity (α) on various design nodes. The equation for dynamic power is:

$$\text{Dynamic Power} = \alpha \times \text{fclk} \times C \times V2 \qquad \textit{Equation 2-1}$$

For most designs, some parameters are determined either by the FPGA technology (for example, operating voltage) or by design requirements (for example, operating frequency).

Gating the clock or data paths is a common technique used to stop switching activity when the results of those paths are not used. Clock gating stops all synchronous activity, preventing data path signals and glitches from propagating.

The Vivado® tool analyzes the description and netlist to detect unwanted conditions. However, designer knowledge of the application, data flow, and dependencies are not available to the tool, but should be specified by the designer to further remove unwanted conditions.

There are several design nodes that do not affect the PL output, but continue to toggle. Every flip-flop, block RAM and DSP48 has a local clock-enable. Designers can gate the local clock-enable to eliminate unnecessary flip-flop, block RAM, and DSP toggling, as shown in Figure 2-3.

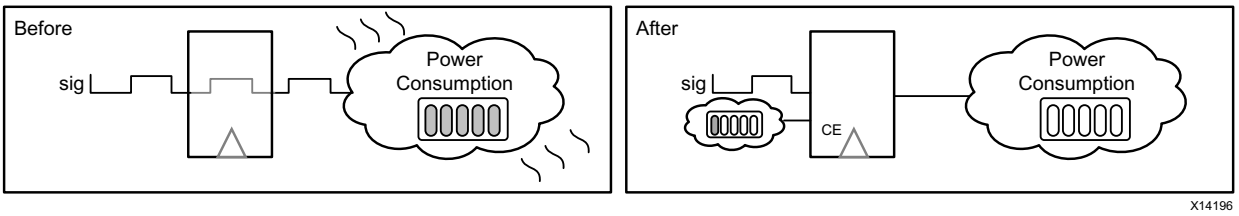

Figure 2-3: **Gating the Local Clock-Enable to Eliminate Unnecessary Toggling**

Designers should ensure the maximum number of elements are controlled by the gating signal when possible. For example, it is more power efficient to gate a clock domain at its source rather than gate each load with a clock-enable signal. Designers can use the different clock buffer primitives in the PL to gate clocks, depending on the application.

The BUFGCE primitive is a global clock buffer with a clock enable. BUFGCE can be used to dynamically gate a global clock or clock domain, glitch free. Using this resource also reduces high clock enable fanout and saves PL fabric routing resources.

If gating is needed for a specific logic function or a clock domain in the PL, the BUFHCE or BUFRCE primitive can be used. These primitives reduce loading and capacitance, lowering the PL dynamic power consumption.

The BUFGMUX_CTRL primitive can be used to distribute clocks to specific regions and avoid unwanted PL fabric dynamic switching. It can also be used to switch between fast and slow clocks in order to reduce power.

There are often several design nodes that do not affect the PL output but continue to toggle, resulting in unwanted dynamic power consumption. The FPGA clock enables can be used to gate those nodes.

Making the best use of timing constraints is also important in low-power design. If an application operates in a temperature-controlled environment, the application can be derated to meet timing. The design should be constrained to use the maximum specified clock rate. Using a faster clock rate than necessary typically has the following negative effects:

- More PL resources are used due to reduced resource sharing.
- Logic and registers are often duplicated to meet tight timing constraints.
- The amount of routing increases.
- There are fewer inferences of PL dedicated features.

All of these negative effects can significantly impact dynamic power consumption.

Block RAM

The power block RAM consumes is directly proportional to the time it is enabled. To save power, the block RAM enable can be driven low when the block RAM is not used. Both the block RAM enable rate and the clock rate are important to consider when optimizing power.

The block RAM should be enabled during an active read or write cycle. Synthesis tools might not infer these primitives, so if they are needed their inference should be verified with a schematic viewer and instantiated if necessary to save power.

Floorplanning

Designs that span multiple clock regions use more clocking resources and consume more power. When possible, place any intermittently used logic in a single clock region to help reduce power, as shown in Figure 2-4. While the tools attempt to do this automatically, some designs may require manual effort like applying an area constraint to achieve this.

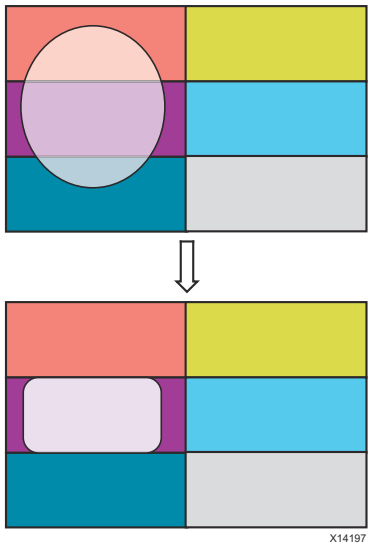

Figure 2-4: Constraining Intermittently Used Logic to a Single Clock Region

Limiting data motion is another power-reduction technique. Instead of moving operands around the PL, move only the results, as shown in Figure 2-5. Using fewer and shorter buses lowers capacitance, improves performance, and consumes less power. The pinout placement and corresponding logic design must be considered during floorplanning.

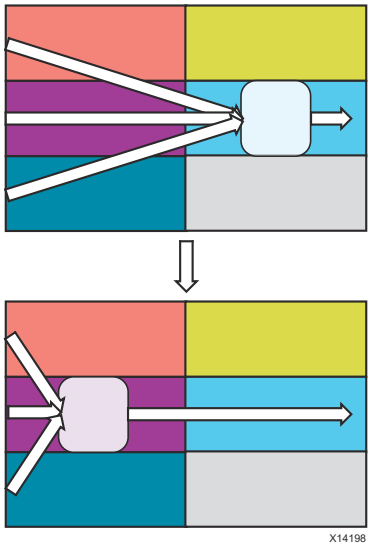

Figure 2-5: Limiting Data Motion

IO Power

I/O power can be a major contributor to total power consumption. In some designs, as much as half of the total power consumption comes from the I/Os, particularly in memory-intensive systems.

Some interfaces do not require fast, differential I/O capabilities. I/O standards such as HSLVDCI can save considerable power in FPGA-to-FPGA communications and in lower-speed memory interfaces.

All Zynq-7000 AP SoCs offer programmable slew rate and drive strength that can be used to reduce I/O dynamic power. The devices support digitally controlled impedance (DCI) technology, and can be tri-stated. DCI eliminates termination power when the I/O's output is enabled, so that the device consumes termination power only during ingress cycles.

Zynq-7000 AP SoCs incorporate a user-programmable receiver power mode for HSTL and SSTL. By controlling the programmable power modes on each I/O, DC power can be reduced by making trade-offs between power and performance.

Zynq-7000 AP SoCs have transceivers optimized for high performance and low jitter. The transceivers offer several low-power operating features, enabling designers to customize the flexibility of operation and granularity to make power and performance trade-offs.

In the transceivers, you can use the shared LC PLL to save power. In four-lane designs with an identical line rate (XAUI, for example), you can use a quad PLL instead of an individual channel PLL. Similarly, because a PLL can run at higher and lower rates within the range, a lower operating range can be selected to save power.

The RXPOWERDOWN and TXPOWERDOWN options can be enabled. PLL power down can be enabled in the lowest-power mode, such as in a system D3 state, which is mostly used in PCIe systems.

You can also save I/O power in the following ways:

- Using time-multiplexing techniques to reduce I/O count.
- Using minimal I/O count design partitioning that can help switch off an I/O bank when not used.
- Reducing the number of I/O standards used within a bank.

Partial Reconfiguration

One way to reduce static power is to simply use a smaller device. With partial reconfiguration, designers can essentially time-slice a block of logic in the PL and run parts of their design independently. The design then requires a much smaller device because not every part of the design is needed 100 percent of the time.

Partial reconfiguration has the potential to reduce dynamic power as well as static power. For example, many designs must run very fast, but that maximum performance might only be needed a small percentage of the time. To save power, designers can use partial reconfiguration to swap out a high-performance design with a low-power version of the same design-instead of designing for maximum performance 100 percent of the time. You can switch back to the high-performance design when the system needs it.

This principle can also apply to I/O standards, specifically when a design does not need a high-power interface all the time. LVDS is a high-power interface, regardless of activity, due to the high DC currents required to power it. You can use partial reconfiguration to change the I/O from LVDS to a low-power interface, such as LVCMOS, at times when the design does not need the highest performance, and then switch back to LVDS when the system requires high-speed transmissions.

Power Estimation

Power calculations can be performed at three distinct phases of the design cycle.

- **Concept phase:** In this phase a rough estimate of power can be calculated based on estimates of logic capacity and activity rates.
- **Design phase:** Power can be calculated more accurately based on detailed information about how the design is implemented in a Zynq-7000 AP SoC.
- **System integration phase:** Power is measured in a lab environment.

Xilinx provides a suite of tools and documentation to help evaluate the thermal and power supply requirements of the system throughout the design cycle. Figure 2-6 shows the tools available at each stage of the design cycle.

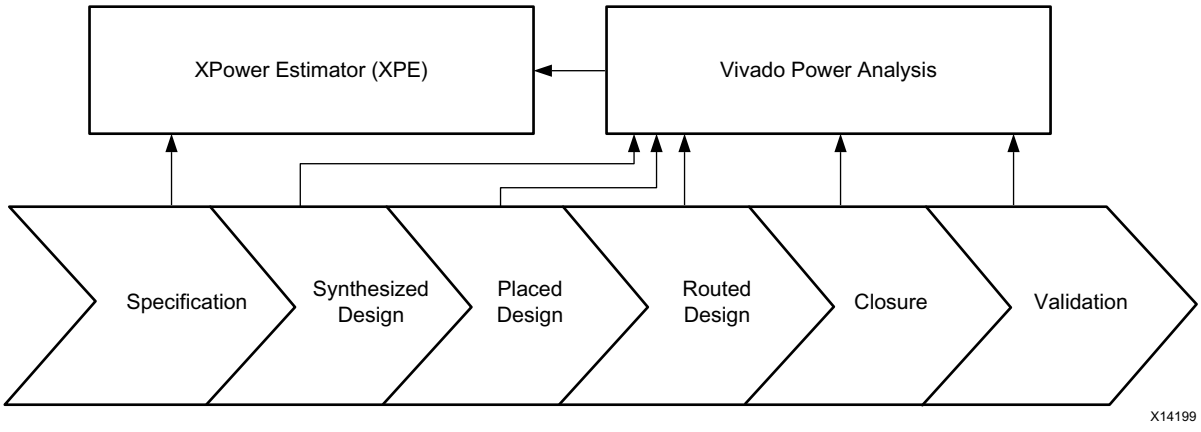

Figure 2-6: Vivado Power Estimation and Analysis Tools in the FPGA Design Process

Some tools are stand-alone, while others are integrated into the implementation process, using information available at each stage of the design process. All tools can exchange information back and forth for efficient analysis.

The final power consumption is determined after the system has been implemented and measured in the lab. This is done by manually probing the development-board power lines, or by providing a mechanism to read the voltage and current from external programmable voltage regulators. Accurate power calculations in early design stages result in fewer problems later.

Xilinx Power Estimator (XPE)

The Xilinx Power Estimator (XPE) spreadsheet is a power estimation tool typically used during a project's concept phase. XPE assists with architecture evaluation and device selection, and helps with selecting the appropriate power supply and thermal management components for the application. The XPE interface for Zynq-7000 AP SoCs is shown in Figure 2-7. Designers can use the tool to specify design resource use, activity rates, I/O loading, CPU clock frequency, and many other design parameters. XPE combines the parameters with the device models to calculate an estimated power distribution.

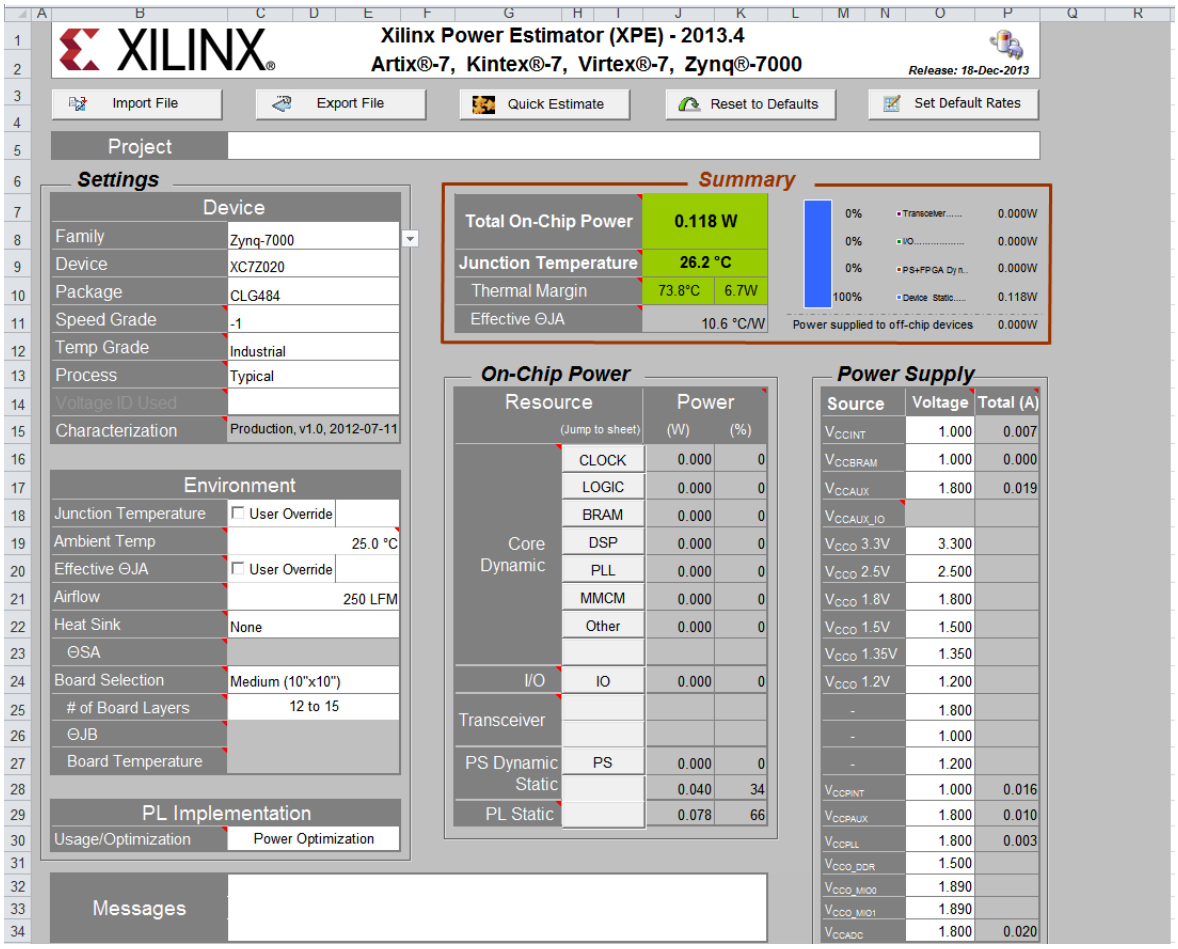

Figure 2-7: Xilinx Power Estimator for Zynq-7000 AP SoCs

XPE is also commonly used later in the design cycle during implementation and power closure, such as evaluating the power implications of engineering change orders. For large designs implemented by multiple teams, the project leader can use XPE to import the utilization and activity of each team's module, then monitor the total power and reallocate the power budget to ensure constraints are met.

System-Level Power Analysis

The final system power consumption is determined after the design has been implemented and measured in the lab. This is done by manually probing the development-board power lines, or by providing a mechanism to read the voltage and current of voltage regulators.

The Zynq-7000 AP SoC Low Power Techniques part 1 - Installing and Running the Power Demo Tech Tip wiki page [Ref 61] provides a reference design demonstrating system-level power consumption while executing different application scenarios on Zynq-7000 AP SoC PS and PL sections. You can refer to this technical article when performing system-level power measurements on your design.

Vivado Power Analysis

Vivado Power Analysis is a tool used to analyze power consumption of placed and routed designs during the design phase. It provides a comprehensive GUI that allows a detailed analysis of the power consumed as well as thermal information for the specified operating conditions. Figure 2-8 shows an example report from Vivado Power Analysis.

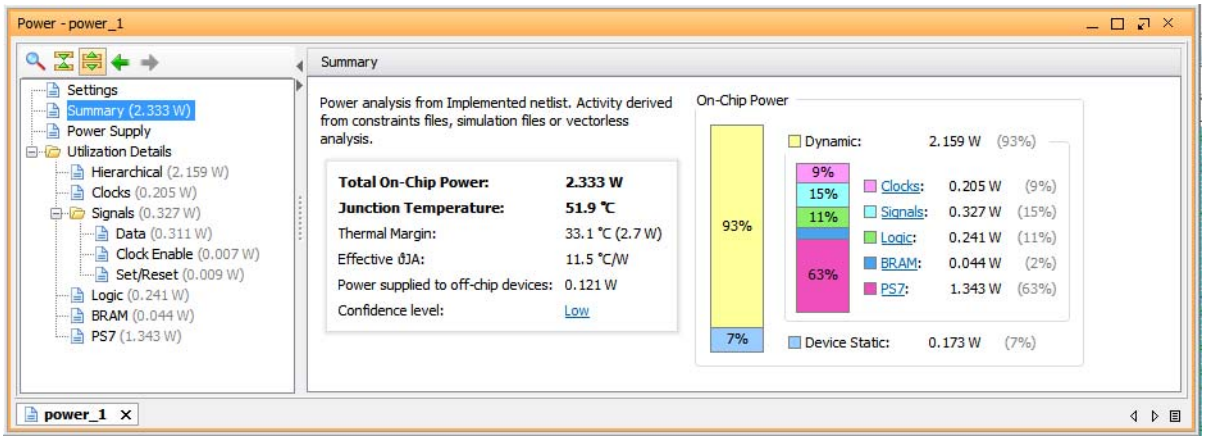

Figure 2-8: Vivado Power Analysis Example Report

The tool provides two different views of power consumption:

- Power consumed by type of blocks found in the design, including clock trees, logic, signals, I/Os, and PS IP such as block RAMs and DSP blocks.
- Power consumed throughout the design hierarchy.

Designers can toggle between the two views to perform a detailed power analysis. The views provide an efficient method for locating the blocks or design parts that consume the most power, thus identifying places to focus power optimization efforts.

Switching activity information from Value Change Dump (VCD) and Switching Activity Interchange format (SAIF) files can be entered into the Vivado Power Analysis tool for more accurate power estimation. VCD files contain header information, variable definitions, and value change details for each step of a simulation. SAIF files contain signal toggle counts and timing attributes specifying the time duration of signals at level 0, 1, X, or Z.

PL Power Optimization in Vivado

Although designers can use clock gating to optimize power as described previously, it is rarely done. This is either because the design contains intellectual property from other sources or because of the effort involved in doing such fine-grained clock gating. The Vivado tools automate such power optimizations to maximize power savings while minimizing effort.

The Vivado design tools offer a variety of power optimizations enabling you to minimize dynamic power consumption by up to 30%.

Vivado performs an analysis on the entire PL design, including legacy and third-party IP blocks, for potential power savings. It examines the output logic of sourcing registers that do not contribute to the result during a given clock cycle, and then creates fine-grained clock gating and logic-gating signals that eliminate unnecessary switching activity.

Power optimizations are also applied to dedicated block RAMs. Most of the power savings is realized by disabling the dedicated block RAM's enable when no data is written and when the output is not being used.

Refer to the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 10] for more information on using the power optimization options and extracting the power consumption information from the design.

Correcting Over-Budget PL Power Consumption

Late in the design cycle there is pressure to get the system to market, and most system parameters are well defined, such as the board environment and cooling solution. Even though this limits the engineering rework you can do, further power reduction might be possible in the PL. The following process can help focus your efforts on the areas in the PL with the highest potential for power reduction.

1. **Determine which Power Budget is Exceeded**

   Vivado GUI users can review the summary view in the Vivado Power Analysis report, and command line users can use the summary section of the power report file. The on-chip and supply power tables provide a high-level view of the power distribution. Use the summary view to determine the type and amount of power that exceeds your budget.

2. **Identify the Focus Areas**

   Review the different detailed views in the Vivado Power Analysis report or Xilinx Power Estimator. Analyze the environment parameters and the power distribution across the different resources used, the design hierarchy, and clock domains. When an area of the design is found where power seems high, the information presented should help in determining the likely contributing factors.

3. **Experiment**

   After developing a list of focus areas for power optimization, sort the list from easiest to most difficult and decide which optimization or experiment to perform next. The power tools allow what-if analysis so that design changes can be made quickly and power estimates produced without requiring edits to code or constrains, and without rerunning the implementation tools.

   While power optimization techniques that can be used are listed in this document, more information can be found in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 10].

Experiment Using the Vivado Power Optimizer Feature

To maximize power savings when running the power optimizer in the Vivado tools, run power optimization on the entire design and do not exclude portions of the design. If power savings are not realized after enabling power optimization, there are three areas that should be examined more closely:

- Global set and reset signals
- Block RAM enable generation
- Register clock gating

A low number of enables generated by power optimization in any of those areas could indicate a need to review coding practices, or the options and properties set for design.

Experiment with the Vivado Power Analysis Feature

In the Vivado Report Power dialog box, adjustments can be made before rerunning the analysis to review the power implications. Refer to the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 10] for more information on how to use the different options in the Vivado Power Analyzer tool to produce optimal results.

Experiment with Xilinx Power Estimator (XPE)

The Vivado power analysis results from modules developed by multiple sources can be imported into XPE. This permits a review of total power once the separate IP blocks are implemented in the device. You can estimate the implications of design changes on power consumption without requiring code or netlist changes. The estimation done by XPE is not as accurate as that done by the Vivado Power Analysis tool because adjustments cannot be made at the individual logic element or signal level in XPE.

Conclusion

Understanding and implementing power-sensitive design techniques before coding is the single-largest method for reducing system power. Using the various Xilinx tools at the appropriate design cycle stages also helps in meeting power specifications, and provides the board designer with information on selecting the number, type, and size of the power supplies.

Clocking and Reset

External and internal clocking resources are available for use as source clocks to various IP blocks in the Zynq-7000 AP SoC. Those resources are described below. Also described in this section are the various hardware, software, and debug resets, particularly when and how you should use them.

External Clocks

PS_CLK

On the PS side, a fixed-frequency oscillator in the range of 30–60 MHz is typically used to provide the processor clock PS_CLK. The clock must be a single-ended LVCMOS signal, using the same voltage level as the I/O voltage for MIO bank 0. From this clock, all other PS internal clocks are generated based on three PLLs: ARM, DRM, and IO PLL.

The default PS_CLK frequency used on Xilinx evaluation boards is 33.3 MHz. Other clock frequencies can be used, but the following items are dependent on the PS clock frequency and must be adjusted accordingly:

- The LogiCORE IP Processing System 7 configuration wizard calculates the derived clock dividers and multipliers of each of the PLLs as well as the I/O peripheral clocks such as SPI or UART based on the selected PS_CLK frequency. These values are later used by the first stage boot loader (FSBL) during initialization of the PS.
- The U-Boot board configuration include file.
- The Linux design-specific device tree.

Others

On the PL side, single-ended or differential fixed-frequency oscillators can be used as additional clock sources for greater flexibility. They should be connected to multi-region clock capable (MRCC) input pins and adhere to the I/O standard and voltage of the corresponding PL bank. PL input clock jitter can be much smaller than that of PS-derived clocks. For certain applications (such as to generate a video resolution dependent, accurate pixel clock in a video system), external low-jitter programmable clock synthesizers are used.

Certain board peripherals require an external crystal to operate the PHY (for example, a 25 MHz crystal is required for the Ethernet PHY). Also, some peripheral I/O interfaces provide input and/or output clocks to communicate with the corresponding PHY (such as Ethernet RX and TX clocks).

Internal Clocks

PS

All clocks generated by the PS clock subsystem are derived from one of three programmable PLLs: CPU, DDR, and I/O. Each of these PLLs is loosely associated with the clocks in the CPU, DDR, and peripheral subsystems. During normal operation, the PLLs are enabled, driven by the PS_CLK clock pin. In bypass mode, the clock signal on the PS_CLK pin provides the source for the various clock generators instead of the PLLs. After the boot process, the bypass mode and output frequency of each PLL can be individually controlled by software.

The CPU clock domain is composed of four separate clocks: CPU_6x4x, CPU_3x2x, CPU_2x, and CPU_1x. These four clocks are named according to their frequencies, which are related by one of two ratios: 6:3:2:1 or 4:2:2:1 (abbreviated 6:2:1 and 4:2:1, respectively). All of the CPU clocks are synchronous to each other. There are two independent DDR clock domains: DDR_3x and DDR_2x. These clocks are asynchronous to each other and the CPU clocks. Most I/O peripherals clocks have dedicated dividers. Each peripheral clock is completely asynchronous to all other clocks.

PS - PL Interface

PL AXI channels (AXI_HP, AXI_ACP, and AXI_GP) have asynchronous interfaces between the PS and the PL. The synchronization, where the clock domain crossing occurs, is located inside the PS. Therefore, the PL provides the interface clock to the PS. Each of the aforementioned interfaces could use unique clocks in the PL.

The PS provides four frequency-programmable fabric clocks (FCLK [3:0]) to the PL that are physically spread out along the PS-PL boundary. The clocks can be controlled individually by setting the clock's source (ARM, DDR, or I/O PLL) and the clock's output frequency. There is no guaranteed phase relationship between any of the four FCLK clocks, even when sharing the same clock source. Make sure to use appropriate design constraints when interfacing between multiple FCLK regions. The FCLK clocks are disabled until the PS - PL level shifters are enabled.

RECOMMENDED: *It is good practice to route a single FCLK into a clocking wizard IP core instantiated inside the PL to generate more than one phase-aligned output clocks.*

The following are pros and cons of using FCLK:

- FCLK is the preferred PL clock under the following circumstances:
  - The processor controls the PL clock frequency.
  - An on-board clock generator is not available.
- FCLK is not the preferred PL clock under the following circumstances:
  - The PL clock frequency is outside the frequency range supported by FCLK.
  - The PL uses a clock provided by the FPGA pins. This is common in source-synchronous protocols that use the input clock to sample receive data.
  - Low clock jitter is required.
  - Some IP blocks that require specific clocking cannot use FCLK:
    - The memory-interface generator (MIG) requires a differential clock, Therefore, FCLK cannot be used for the MIG except at reduced frequencies due to jitter.
    - GTs should use a differential clock from the board as a reference clock.

For more information on the PS clock system, refer to the Clocks chapter of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

PL

The PL provides clock primitives that are commonly found on FPGA devices, such as global or regional clock buffers (BUFG, BUFR), phased-locked loops (PLL), or mixed-mode clock managers (MMCM). For convenience, the clocking wizard IP core implements a wrapper around the MMCM/PLL primitives with up to eight configurable output clocks. The user can

optionally enable dynamic reconfiguration of clock frequencies using the dynamic reconfiguration port (DRP) or an AXI-Lite interface.

TIP: *In the simplest case, a complete Zynq-7000 system can be built with a single input clock based on PS_CLK, and all PL clocks are generated from the provided FCLKs and FPGA clocking resources.*

PS Power-On Reset

The PS power-on reset (PS_POR_B) is an active-low signal used to hold the PS in reset until all PS power supplies are stable and at their required voltage levels. This signal should be generated from the power supply power-good signal or from a voltage supervisor chip. At the time PS_POR_B is released, the system clock (PS_CLK) must have been stable for 2,000 clock cycles. PS_POR_B should be pulled high to VCCO_MIO0. When asserting PS_POR_B, the pulse length must be longer than 100 µs.

The power-on reset is the chip master reset. It resets every register in the device that can be reset, resets all PS RAM (including OCM, Fifos, buffers, etc.) and starts BootROM execution, clearing the PL configuration. When PS_POR_B is held low, all PS I/Os are held in 3-state.

PS System Reset

The PS system reset (PS_SRST_B) is an active-low signal that is used primarily for debugging proposes. PS_SRST_B must be high to begin the boot process. If PS_SRST_B is not used it can be pulled high to VCCO_MIO1. When asserting PS_SRST_B, the pulse length must be longer than 1 µs.

A PS system reset (PS_SRST_B) resets all functional logic without affecting the debug environment. This differs from a power-on reset (PS_POR_B), which erases the debug configuration. The PS_SRST_B erases all PS RAM, starts BootROM execution, and clears the PL configuration. It does not re-sample the boot-mode strapping pins, unlike the PS_POR_B. The boot mode remains the same as the previous power-on reset, and the security level of the previous boot is retained.

The PS_SRST_B signal must be de-asserted before the PS_POR_B signal is de-asserted. If the BootROM execution caused by a PS_POR_B reset is interrupted by the assertion of the PS_SRST_B reset signal, the system will lock down. If both PS_SRST_B and PS_POR_B are used, PS_POR_B must be the last signal that is de-asserted. For more information, refer to Xilinx Answer Record 52847 [Ref 68].

Contact Xilinx Technical Support if you need a PS reset solution that does not clear the PL.

System Software Reset

The System Software Reset, also called SLCR Soft Reset, is asserted by writing to PSS_RST_CTRL[SOFT_RST] and has the same effect as asserting the PS_SRTS_B pin. All of the PS RAMs are cleared and the PL is reset as well.

Watchdog Timer Resets

There are two sources of watchdog timer resets: the System Watchdog, SWDT, and the two ARM Watchdog Timers, AWDT0 and AWDT1. The SWDT always resets the entire system, while each of the AWDTs can be used to reset either the associated ARM core (same effect as CPU Reset) or the entire system (same effect as System Software Reset).

CPU Resets

There are two CPU Resets, one for each ARM core asserted by writing to A9_CPU_RST_CTRL[A9_RSTx]. A CPU Reset to a single processor must be applied from the other CPU, through JTAG or the PL.

Debug Resets

There are two debug resets, Debug System Reset and Debug Reset, that originate from the ARM DAP and are controlled by JTAG. The Debug System Reset has the same effect as a System Software Reset, whereas the Debug Reset only resets the debug logic.

Peripheral Resets

Individual peripheral resets can be asserted under software control, using programmable bits within the SLCR. However, asserting reset at the peripheral block level is not recommended. Resetting a peripheral without completing all in-flight or pending transactions will cause the system to hang, because AXI transactions do not support a timeout mechanism. When asserting reset to a peripheral, all pending and in-flight transactions must be completed and no future transactions can be issued prior to the reset.

PL Resets

The PS provides four programmable reset signals to the PL (FCLK_RESET [3:0]). The resets are individually programmable and independent of the PL clocks. After a POR or system-wide reset, the reset signals are not de-asserted until the BootROM execution finishes and the PS to PL level shifters are enabled

The FCLK_RESET is loosely associated with the FCLK of the same number. That is, the FCLK needs to be toggling for the FCLK_RESET to propagate out of the PS. The reset is an asynchronous reset to the PL and you must synchronize the reset inside the PL if required.

RECOMMENDED: *To synchronize an FCLK_RESET, connect the signal to the* ***external reset*** *input port of a* ***proc_sys_reset*** *IP core. By doing so, the reset output signals connected to other PL IP cores can be synchronized to the slowest clock.*

If the FCLK_RESET signal drives the reset of AXI-based IP in PL and is reasserted after the initial de-assertion described above, resetting the IP later without completing all in-flight or pending transactions to and from the process block will cause the system to hang. The system must idle all AXI masters and finish all AXI transactions prior to asserting reset.

Interrupts

Generic Interrupt Controller

Because a Zynq-7000 AP SoC has many interrupt sources, the MPCore multicore processor includes an implementation of the generic interrupt controller (GIC) architecture to help funnel, prioritize, and arbitrate those interrupt sources. The GIC maps interrupts to specific processor nIRQ and nFIQ lines. Multiple, concurrent interrupts are presented serially to one or more processor interrupt lines.

Conceptually, the GIC contains an interrupt distribution block (distributor) and two processor-interrupt blocks. Each block contains a set of registers. Registers in the processor-interrupt blocks can only be accessed by the processor they are attached to using its private bus and cannot be accessed by the other CPU or other AXI masters in the system. Distributor registers can be accessed by either processor, and some registers are also banked for secure and non-secure access.

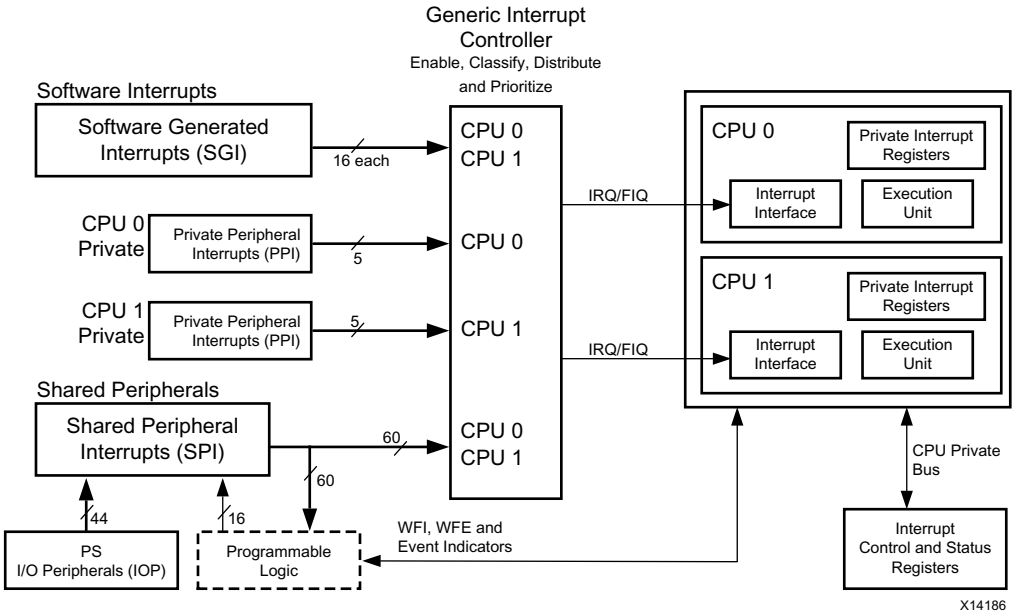

Figure 2-9: **System-Level Block Design**

Interrupts sent to the GIC can be unmanaged (legacy) or managed. Legacy interrupts are not controlled by the GIC and the interrupt handler should not interact with GIC registers.

Managed interrupts are controlled by the GIC and the interrupt handler must interact with GIC registers.

The distributor can control managed interrupt sources as follows:

- Define the interrupt as edge-sensitive or level-sensitive, subject to hardware configuration limitations.
- Assign a 5-bit priority to the interrupt, with a programmable binary point.
- Assign a TrustZone technology security state to the interrupt. Interrupts labeled as secure are called Group 0 interrupts, and non-secure interrupts are labeled as Group 1 interrupts.
- Route shared peripheral interrupts to one or both processors.

  Note: Private peripheral interrupts (PPIs) are dedicated to each CPU and cannot be routed elsewhere other than through the GIC.
- Route a software-generated interrupt to one or both processors.
- Save and restore the pending state of each interrupt. This is useful in low-power applications.

Interrupt Architecture

The PS and PL interrupt sources, PS interrupt hierarchy, and Cortex-A9 processor considerations are described in this section.

PS Interrupt Sources

Each Cortex-A9 processor can be interrupted by the following sources:

- Sixteen software-generated interrupts (SGIs) are available for software to interrupt either processor.
- Five private peripheral interrupts (PPIs) are available. There are two interrupts from PL (FIQ and IRQ) and one each from the global timer, the private timer, and AWDT.
- Sixty shared peripheral interrupts are available. There are 44 PS I/O peripheral interrupts and 16 PL interrupts.
- Four PL interrupts can bypass the GIC and directly interrupt the processors, reducing interrupt latency.
- Although it is not an interrupt, a CPU may use the WFE instruction to deliberately stall and wait for an event on a dedicated input line from the other CPU or the PL.

PL Interrupt Sources

Each device in the PL that responds to interrupts must arrange for its own interrupt controller, if needed. The following interrupts can be sent from the PS to the PL:

- Twenty-nine shared peripheral interrupts from the PS. These correspond to many (but not all) of the PS peripherals.
- A processor can also use software-generated interrupts to interrupt the PL by using EMIO GPIOs, an AXI_GPIO output channel in the PL, or by asserting a per-processor hardware event line using the SEV instruction.

Interrupt IDs

A Zynq-7000 AP SoC supports IRQ IDs #0 through #95, as follows:

- **IRQ ID #0–#15.** These are assigned to software generated interrupts, allowing software to interrupt a processor, including the processor hosting the software.
- **IRQ ID #16–#31.** These are assigned to private peripheral interrupts. IRQ ID #16–#26 are unused. Zynq-7000 AP SoCs implement IRQ ID #27–#31, as follows:
  a. IRQ ID #27 is used by the processor global timer.
  b. IRQ ID #28 is used by the managed FIQ and is also an unmanaged input.
  c. IRQ ID #29 is used by the processor private timer.
  d. IRQ ID #30 is used by the processor AWDT.
  e. IRQ ID #31 is used by the managed IRQ and is also an unmanaged input.
- **IRQ ID #32–#95.** These are assigned to shared peripheral interrupts. IRQ ID #36 is unused. Zynq-7000 AP SoCs implement IRQ ID #32–#35 and #37–#95. Thirty-one interrupts (IRQ ID #32–#62) can all be unlocked or all be locked against user change.

Unique Capabilities

Interrupts can originate from the PS or the PL, and a PS processor or dedicated hardware in the PL fabric (such as MicroBlaze™) can respond to interrupts. This enables the unique capabilities of Zynq-7000 AP SoCs, described in the following sections.

Interrupt Processing in Fabric

Custom hardware can be created to offload interrupt processing to the PL. When this is done, interrupt latency can be very low and deterministic (clock cycle accuracy). Also, the custom hardware can perform computations that would take longer on the processor, in parallel with the processor. If this is done, data stitching might be required, and the ACP might be useful in providing cache coherency for PL masters.

Custom hardware can also be used to filter interrupts, reducing PS interrupt frequency and processor loading. This can be done by converting multiple, low-level interrupts into fewer, high-level interrupts.

Processor as an Extension of Fabric

Alternatively, processors in a Zynq-7000 AP SoC can act as an extension of fabric hardware, performing tasks in software that would be difficult to implement in hardware. The PL can send an interrupt to the PS to initiate a task, and interrupts can be sent from the PS to PL indicating task completion. This method can also be used with the PS DMA330 controller to provide additional execution threads.

Using OCM for Handlers

Fetching an interrupt handler from DDR memory might take more time than desired. The Zynq-7000 AP SoC has a large L2 cache, but an interrupt handler may not be cached when an interrupt occurs. In this case, it may be beneficial to place interrupt handlers in on-chip memory (OCM). When this is done, the variability in time spent fetching an interrupt handler is limited to L1 caches misses, leading to reduced jitter

Processor Affinity

Because there are two processors, polling responsibilities and interrupt handlers can be divided between the processors, reducing latency and improving response time. This can be done in either symmetric or asymmetric multi-processing systems.

Using FIQ and IRQ

Each Cortex-A9 in a Zynq-7000 AP SoC has two interrupt lines, nFIQ and nIRQ, driven by the ARM e Generic Interrupt Controller (GIC). The 'n' prefix indicates that they are active low interrupts. FIQ has lower latency than IRQ and is typically used for high-priority interrupts. IRQ is used for interrupts that do not require low-latency response. An FIQ always preempts an IRQ, and a preempted IRQ handler resumes execution after the FIQ handler finishes.

TrustZone

Zynq-7000 AP SoCs use ARM TrustZone technology, allowing system components to be identified as secure or non-secure. For example, at different times each processor can operate in secure and non-secure modes. Similarly, interrupts can be identified as secure and non-secure. This allows isolation of secure components from those that are not secure. In this case, higher-priority FIQs can be used for secure interrupts and lower-priority IRQs for non-secure interrupts. Many secure settings can also be locked to prevent changes.

System Design Considerations

System designs should consider the performance impacts of various interrupt implementation options:

- Latency and associated jitter should be characterized and managed by hardware or the operating system. Jitter occurs if the OS masks interrupts for an unpredictable time period, or because higher priority interrupts are being serviced.
- Policies for peripherals and associated software should be defined for cases when interrupt handling is delayed. Choices include dropping data, throttling or pausing the remote data source, and allowing hardware to run with degraded performance.
- Processor utilization is sufficient for peripheral services and other tasks, assuming average and worst case scenarios.
- The interrupt handler's memory location can affect instruction-fetch performance. Memory closer to the processor will have lower latency. Aligning the interrupt service routine so that it starts on a cache line boundary can improve latency.
- When an interrupted peripheral operates on data that could be cached by the processor, the processor must maintain cache coherency by invalidating or flushing its caches, or by using hardware coherency mechanisms such as the ACP. On Zynq-7000 AP SoCs, PL peripherals that use the ACP port avoid this restriction.
- In systems implementing asymmetric multi-processing (AMP), an ownership policy for interrupts (and other system resources) must be established.

Embedded Device Security

The definition of security in modern embedded devices depends on the application. For mobile phones, security could refer to preventing unauthorized access to personal data, such as bank account numbers and passwords. For data center switches, security could refer to preventing a hacker from maliciously logging in and disrupting network services. Stealing and decompiling binary software logic stored in flash memory or physically breaking open devices for reverse engineering are also possible embedded device security challenges.

Potential threats to embedded devices include:

- Data privacy in the embedded device
- Cloning the embedded device
- Denial of service
- Malware insertion to change the embedded device behavior
- An insider providing keys to an adversary

The above examples describe threats from hackers and competitors. Other system security threats can come from poorly-written programs. Such programs can accidentally corrupt other programs and user data stored in the embedded-device flash, or disrupt normal embedded-device operations by monopolizing the processor or peripheral bandwidth.

Zynq-7000 AP SoCs are designed to improve the security of the embedded systems they power. This section highlights security features of Zynq-7000 AP SoCs that can be used to provide different levels of security, such as:

- Boot image and bitstream encyption and authentication (secure boot).
- Partitioning the system into separate secure zones (TrustZone).
- Deploying asynchronous multiprocessing (AMP) on Zynq-7000 AP SoCs.
- Linux deployment on Zynq-7000 AP SoCs.

Secure Boot

An embedded system using a Zynq-7000 AP SoC typically wants to prevent the device from being used differently than what was originally intended. An embedded system with a Zynq-7000 AP SoC will typically have at least one of two programmable components: software (PS images) and PL bitstreams. The system must be designed to ensure that once it is deployed into the field, untrusted and corrupted programmable components are not used for system boot. Also, security against programmable component theft must be included to prevent a malicious competitor from stealing PS images and PL bitstreams. Authentication is used to prevent booting with unauthorized programmable components, and encryption is used to prevent programmable components theft.

Zynq-7000 AP SoCs provide multiple features in hardware and software (BootROM) that can be configured to ensure that trusted programmable components (software and PL bitstreams) are used to boot a Zynq-7000 AP SoC. Secure boot mode is restricted to NOR, NAND, SDIO, or Quad-SPI flash as the external boot device. A secure boot from JTAG or any other external interface is not allowed.

PS Software Boot Flow

The boot flow for PS is shown in Figure 2-10.

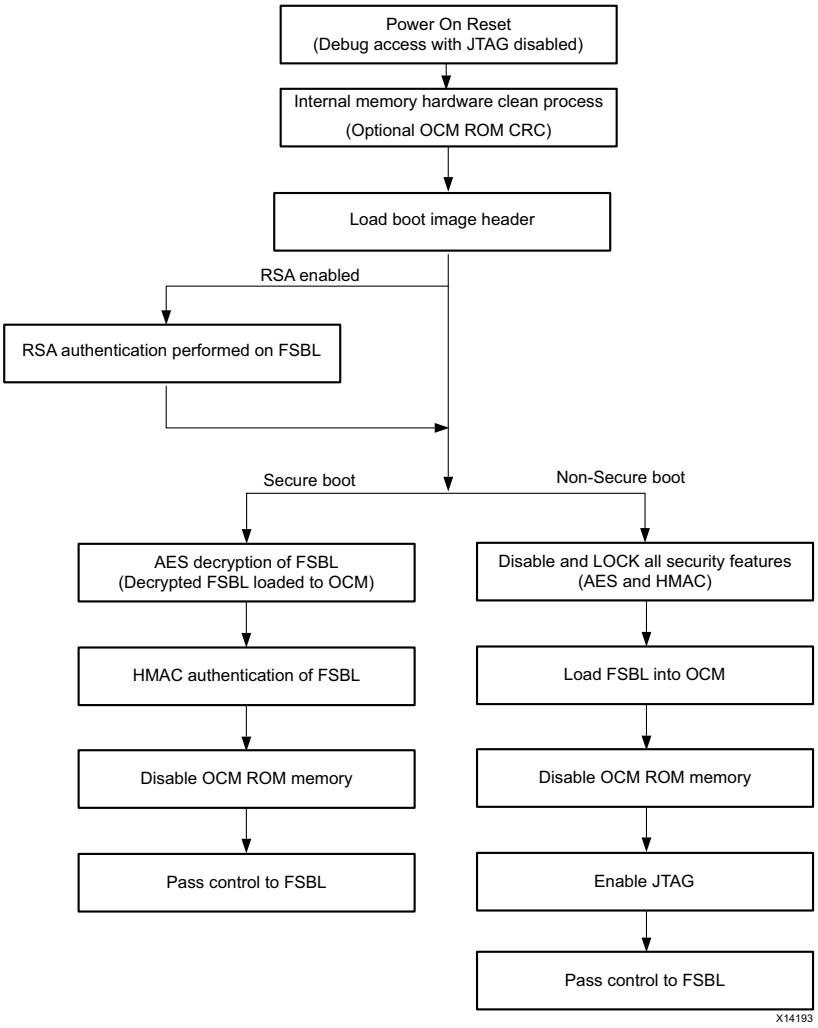

Figure 2-10: **PS Boot Flow**

Power-On BootROM CRC Check

After power-on, the BootROM is the first software component that executes. An optional 128KB CRC check can be done on the BootROM prior to loading the FSBL. The CRC check is controlled by eFuse settings. Refer to the Secure Key Driver section in *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40] for more information on programming eFuses in Zynq-7000 AP SoCs.

During BootROM execution, the boot header is read from the external storage boot device (SDIO, QSPI flash, NAND flash, or NOR flash) as configured by the mode pins. The header contains information indicating whether the device will boot securely or non-securely. Refer to the "Boot and Configuration" chapter in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information on the mode pins.

RSA Authentication of FSBL

During execution, the BootROM can authenticate a secure FSBL prior to decryption or a non-secure FSBL prior to execution using RSA public-key authentication. This feature is enabled by blowing the RSA Authentication Enable fuse in the PS eFuse array. Authenticating the FSBL can ensure that a malicious or corrupted FSBL does not compromise a Zynq-7000 AP SoC by booting from an unauthorized boot source using unauthorized programmable components. Refer to *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40] for more information on RSA authentication.

Secure BOOT Image

The programmable components of a monolithic boot image (BOOT.bin) include:

- PS image components
  - An initialization header that can optionally write values to registers. For example the initialization header can be used to increase the CPU clock speed or boot device speed before the BootROM copies and executes the FSBL.
  - FSBL.
  - Optional secondary-boot loader, such as U-Boot or bare-metal software.
  - Optional data images and multiple ELF.
  - Optional Linux uImage.
- PL bitstream

The boot image can be made secure, if desired. The PS images and PL bitstreams in a secure boot image are authenticated using a hash-based message authentication code (HMAC) and encrypted using the advanced encryption standard (AES). Refer to *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40] and *Zynq-7000 All Programmable SoC Secure Boot Getting Started Guide (*UG1025) [Ref 20] for more information on creating secure boot images.

AES & HMAC Authentication Engines

A Zynq-7000 AP SoC PL contains AES decryption and HMAC authentication engines. Therefore, the PL must be powered on during the secure boot process, even if the secure boot image does not have a PL bitstream. The BootROM checks whether the PL is powered on prior to reading encrypted images from boot devices, so the embedded system must ensure the PL is powered on before decryption. Because the PL is powered by a different power rail, the embedded system must be designed to ensure the rail has appropriate connectivity to the power regulator.

Bootgen

The programmable components of BOOT.bin are assembled using a Xilinx software tool called Bootgen. Bootgen also encrypts the programmable components when the secure boot option is selected. Refer to "Creating a Secure Boot image" in *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40] and to "Using Bootgen" in *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

Generating Keys

The secure boot authentication and decryption process requires the use of cryptographic keys. The keys used by Zynq-7000 AP SoCs are:

- AES 256-bit key
- HMAC key
- RSA Primary Secret Key (PSK)
- RSA Primary Public Key (PPK)
- RSA Secondary Secret Key (SSK)
- RSA Secondary Public Key (SPK)

You can provide a a developer's key for the AES and HMAC engine or generate them using Bootgen. Bootgen creates one key for both key0 and HMAC. If unique values for key0 and HMAC are needed, they can be created by running Bootgen twice, taking the key from each run, and providing it as a user-supplied set of keys in a third run. The AES key is programmed into either eFuse or BBRAM in the PL using Vivado or the Secure Key Driver.

In the Zynq AP SoC, the primary RSA key is used to authenticate the secondary keys. Secondary keys are used to authenticate partitions (such as software, data, and bitstream). OpenSSL is used to create RSA primary and secondary keys. OpenSSL is used because it is readily available, but you can also use other methods to generate keys. The RSA key generated by OpenSSL is a private/public key pair. The public key is a subset of the private key. For security, it is important to protect the private key. In RSA, the private key is used to sign the partitions at the manufacturing site, and a hash of the public key is programmed into the Zynq AP SoC embedded device to verify the signature.

eFuse / BBRAM for Security

For RSA authentication, the hash of the PPK is stored in the PS eFuse array. The PS eFuse is programmed using the secure key driver. For AES decryption, the key is stored in either the eFuse or BBRAM in the PL. The PL eFuse control bits for eFuse Secure Boot, BBRAM Key Disable, and JTAG Chain Disable are programmed using Vivado or the Secure Key Driver. The eFuses are one-time programmable (OTP). A power-on reset (POR) is required after programming the eFuses.

For more information on the Secure Key Driver, as well as many other security features discussed in this section, refer to *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40].

Partitioning a System in Separate Secure Zones (Trust Zone)

Secure boot uses encryption and authentication to prevent a boot from using unauthorized software, and it prevents theft of programmable PS images and PL bitstreams. However, system designers may want to provide additional protection levels, such as allowing access to certain system components only when trusted software executes on the system. This would help prevent dynamically-loaded third-party applications from accessing private data or monopolizing systems resources, degrading system performance. For example, a system designer might not want third-party software to access system flash that is used to store private data, such as bank account details and passwords.

ARM TrustZone technology ensures runtime security by enabling the system designer to divide the system into logical partitions that allow only trusted software to access secure components at the hardware level. For more information, refer to:

- *ARM Security Technology: Building a Secure System using TrustZone® Technology* [Ref 74]
- *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC User Guide* (UG1019) [Ref 19]

Security using Asynchronous Multiprocessing (AMP)

Asynchronous multiprocessing (AMP) can be used in a Zynq-7000 AP SoC for system security by restricting where software can execute. For example, one ARM Cortex-A9 processor can be used to execute untrusted third-party applications and the other ARM Cortex-A9 processor can be used to execute trusted software. The operating system on the ARM Cortex-A9 processor running trusted software is given system resources, such as peripherals, that are mission critical. The operating system on the ARM core running untrusted software is not made aware of the secure peripherals. This approach provides a logical separation where untrusted software cannot intentionally or unintentionally degrade system performance. This approach ensures runtime security. Preventing a malicious boot of a Zynq-7000 AP SoC by an unauthorized programmable component is covered by secure boot.

Linux Deployment

Because Linux is ported to Zynq-7000 AP SoCs, Linux security features are available. Linux security includes file access controls, use of the memory management unit to prevent illegal memory access, such as by programs.

Linux-based platforms such as Android also provide device security by using virtual machines that prevent third-party applications from directly accessing hardware resources.

Many other third-party proprietary security extensions exist for Linux.

Profiling and Partitioning

Profiling tools help you determine how to partition an application's functions between hardware and software for optimal performance.

Software Profiling

Profiling is a form of program analysis that is used to aid the optimization of a software application. It is used to measure a number of application code properties, including:

- Memory usage
- Function call execution time
- Function call frequency
- Instruction usage

Profiling can be done statically (without executing the software program) or dynamically (done while the software application is running on a physical or virtual processor).

Static profiling is generally done by analyzing the source code, or sometimes the object code.

Dynamic profiling is usually an intrusive process whereby program execution on a processor is interrupted to gather information. Trace mechanisms within some processors can be used to non-intrusively gather profiling data.

Profiling allows a designer to identify code-execution bottlenecks that may be caused by:

- Inefficient code
- Poor communication between functions and a module in the PL
- Poor communication between functions in software
- An algorithm or routine implemented in software that may be more suitably implemented in hardware

Once identified, the bottlenecks can be optimized by rewriting the original software function or by moving it to the PL for acceleration. Alternatively, part of the function could remain in software, while the problematic section could be moved to hardware.

Profiling is also useful when analyzing large programs that are too big to be analyzed by reading the source code. Profiling can help designers identify bugs that may otherwise not have been noticed.

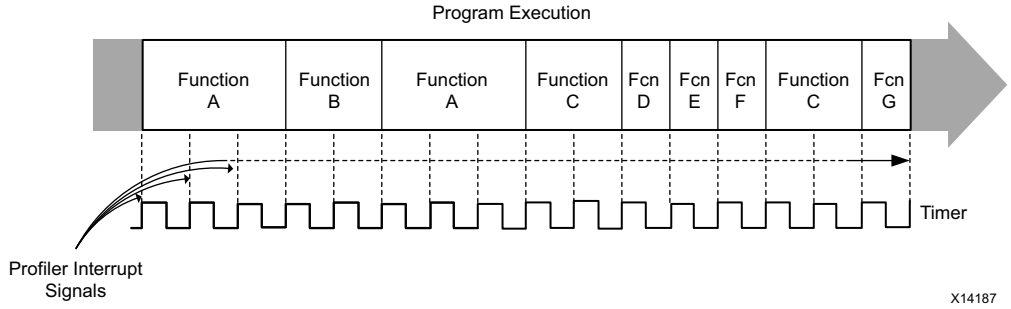

Figure 2-11: **Program Execution Example**

The execution flow of various functions is shown in Figure 2-11 above, with the number of clock cycles required to execute a given function highlighted. By profiling a program, and thus determining the number of clock cycles required to execute each individual function, it can be determined whether or not a function needs to be optimized. During development, a software engineer may have estimated the average time a function should take to execute on a given PS. This estimate can be compared to the profiling results and large discrepancies can be investigated. An example profiling output for the execution flow in Figure 2-11 is provided in Figure 2-12.

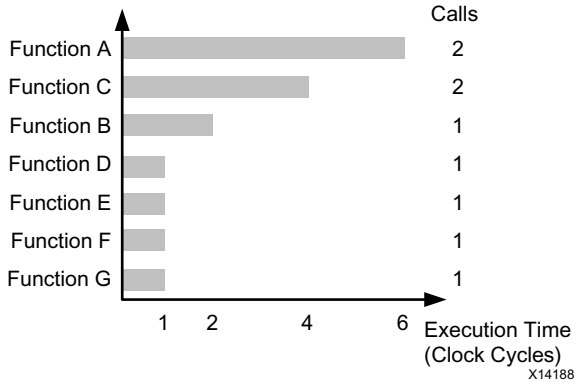

Figure 2-12: **Example Profiling Output**

Xilinx SDK System Debugger

The Xilinx SDK includes profiling tools that help identify bottlenecks in the code that might occur due to the interaction of functions executed within the PL, and functions executed on the processor. Once identified, these bottlenecks can be optimized by migrating the entire function to PL, by optimizing the function code on the processor, or by splitting the function between processor and PL.

SDK supports hierarchical profiling, providing a view into which calling functions and called sub-functions have the largest effect on process performance.

TCF Profiling

Profiling in the Xilinx SDK can be done with the TCF profiler. It works using a statistical sampling method that examines the system at regular intervals, determines what code is running, and updates appropriate counters. The execution profile's accuracy improves the longer a profile is collected, assuming the sample rate is sufficient. Unlike other profilers that make use of interrupts, this method does not cause inaccuracies if the profiled code disables interrupts. The profiled program does not need to be recompiled, as it does with gprof.

Performance Monitoring

The performance monitoring feature in the SDK collects AXI Performance Monitor (APM) event count module data from the PL, ARM Performance Monitor Unit (PMU) data, and L2 cache data from a Zynq-7000 AP SoC PS. The data is collected by the SDK in real-time, over JTAG. The values from these counters are sampled every 10 msec. These values are used to calculate metrics shown in the dedicated view.

The Performance Tab in SDK is shown in Figure 2-13.

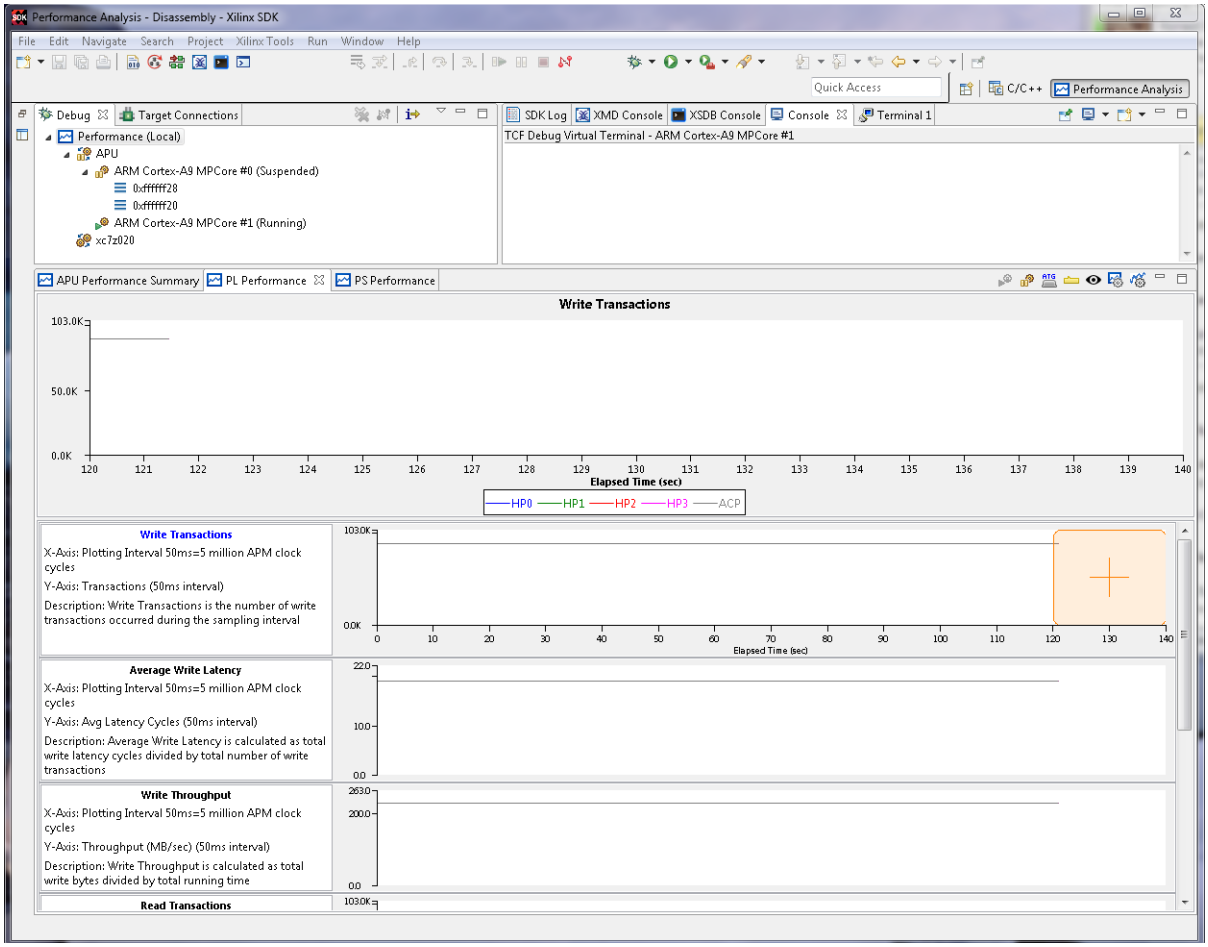

Figure 2-13: **Performance Tab**

For each Cortex-A9 CPU, the following PMU events are monitored:

- Data cache refill
- Data cache access
- Data stall
- Write stall
- Instruction rename
- Branch miss

The following two L2C-PL310 (L2 cache controller) counters are monitored:

- Number of cache hits
- Number of cache accesses

The following APM counters for each of the HP and ACP ports can be monitored when the APM is used in a design:

- Write byte count
- Read byte count
- Write transaction count
- Total write latency
- Read transaction count
- Total read latency

ARM Development Studio 5 (DS-5)

The ARM Development Studio 5 (DS-5) tool chain is a complete suite of software development tools for ARM processor-based systems. DS-5 covers all development stages of ARM processor-based products, from platform bring-up to application profiling, while including a number of ARM Linux and Android specific features.

The Streamline performance analyzer is a component of the ARM Development Studio 5 (DS-5) tool chain. It is a system-level performance analysis tool for Linux and Android systems. It uses sample-based profiling, Linux kernel trace, and software annotation techniques. A Streamline report provides a variety of performance-related information on seven panes: Timeline, Call Paths, Functions, Code, Call Graph, Stack Analysis, and Logs.

By default, the Streamline profiling reports are generated by sampling the program counter every 1ms or 10ms. When event-based sampling is enabled, Streamline takes samples when an event counter reaches the selected threshold value. These samples fall on the instruction that caused the last event, and are used to fill the profiling reports with event-based data instead of time-based data. For example, event-based sampling can be used to determine which parts of the code are causing cache misses or branch mis-predictions.

Practical Applications

Boot Time

The amount of time it takes the application to boot can be a significant concern for some systems. Total boot time includes the execution time for the BootROM, FSBL, 2nd stage boot loader (such as U-Boot), and the operating system. For information about boot time, refer to *Xilinx Answer Record 55572* [Ref 69].

Processor Loading

Processor load is a measure, over time, of the computation actually done as a percent of the total computation that could be done. When the processor load nears 100 percent the system can start to fail or be unresponsive.

Top is a Linux tool that provides an ongoing look at processor load in real time. It displays a list of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes. It can sort the tasks by CPU, CPU usage, memory usage and runtime.

System Latency

System latency is the length of time after a request is made for a system operation to start until the operation actually starts. Identifying the cause of system latency can be challenging because it includes the application and all software layers below it, including the operating system.

Interrupt Latency

Interrupt latency is the length of time for an interrupt to be acted on after it has been generated. Interrupt latency is often measured by the amount of time it takes the CPU to recognize the interrupt and respond by suspending execution of the current processing context. Another measure is the amount of time it takes to begin interrupt processing.

In SMP Linux, interrupt processing can be done by either CPU with the default being to run on CPU0. If processing is left to the default it can result in a large load on one CPU. The CPU affinity of each interrupt can be altered from user space to balance the processing load. There are also applications that help balance interrupt loads, such as irqbalance.

Cyclictest is a Linux tool that measures the amount of time between an interrupt and the start of the interrupt response. The measured time is compared with the expected time, and the difference is the latency. A number of events can delay the actual interrupt response, and cyclictest can be used to identify and characterize those delays.

Hardware Profiling

When profiling, a system can be viewed as a hierarchy of subsystems, each having its own performance monitoring abilities. To completely analyze system performance, all data must be integrated and synchronized into a view that allows the designer to see where time is being consumed in the system.

The PL has flexibility not found in a traditional fixed SOC, giving the designer a range of implementation alternatives. Profiling a design is one method that can be used to fully understand the impacts of a specific implementation.

Program Trace Module (PTM)

The PTM is a module that performs real-time instruction flow tracing based on the Program Flow Trace (PFT) architecture. The PTM generates information that trace tools use to reconstruct the execution of all or part of a program.

The PFT architecture assumes that the trace tools (such as DS-5) can access a copy of the code being traced. For this reason, the PTM generates traces only at certain points in program execution, called waypoints. This reduces the amount of trace data generated by the PTM. Waypoints are changes in the program flow or events, such as an exception. The trace tools use waypoints to follow the flow of program execution.

For full program-flow reconstruction, the PTM traces:

- Indirect branches, with target address and condition code
- Direct branches with only the condition code
- Instruction barrier instructions
- Exceptions, with an indication of where the exception occurred
- Changes in processor instruction set state
- Changes in processor security state
- Context-ID changes
- Entry to and return from debug state when halting debug-mode is enabled.

The PTM can also be configured to trace:

- Cycle count between traced waypoints
- Global system timestamps
- Target addresses for taken direct branches.

Performance Monitor Unit (PMU)

The Cortex-A9 processor PMU provides six counters to gather statistics on the operation of the processor and memory system. Each counter can track any of the 58 events that are significant system performance measurements. Software applications, like the Xilinx System Debugger, can make use of the event counts along with software execution times to help optimize the software. Xilinx drivers provide access to this information so that software applications can retrieve this information from the driver and display it.

Level 2 (L2) Cache Event Counters

The PL310 L2-cache controller incorporates two counters that allow monitoring of cache events. These events can assist in understanding how applications are affecting the L2 cache and how they might be optimized. Xilinx drivers provide access to this information so that software applications can retrieve this information from the driver and display it.

AXI Performance Monitor (APM)

The LogiCORE IP AXI Performance Monitor is a soft IP core that can be built into the PL to measure AMBA AXI system performance metrics in the PL. The performance monitor measures bus latency of masters/slaves (AXI4/AXI3/AXI4-Stream) in a system, the amount of memory traffic over specific durations, and other performance metrics. This core can also be used for real-time profiling of software applications.

The APM monitors the AXI system non-intrusively so that no CPU processing is required. However, software on the ARM CPU can initialize monitoring of specific data and then gather the results from the APM, which is minimally intrusive in system profiling. Xilinx drivers provide access to this information so that software applications can retrieve the information from the driver and display it. The APM can also be configured by the System Debugger in the profile mode.

The APM includes the following capabilities that can be helpful during profiling:

- Studying the latencies of any AXI-based slave, such as a memory controller, and tuning the core.
- Obtaining system-level metrics, such as write-throughput, read-throughput, average interconnect read-latency, and others.
- Analyzing the transaction latencies and identifying the agent causing more idle cycles in the transactions.
- Comparing two similar AXI agents.
- Counting external events (other than AXI), such as FIFO overflow/underflow, interrupts, and others.
- Logging specific events on the monitor slots, and then reconstructing and analyzing the behavior/performance.

Ethernet Statistics Registers

The Gigabit Ethernet Controllers contain statistics registers that are accessible to software through low-level interfaces. The statistics registers record counts of various event types associated with transmit and receive operations. The counts are helpful for profiling and analyzing network performance. Xilinx drivers provide access to this information so that software applications can retrieve it and display it.

Software/Hardware Partitioning

With the help of software profiling, compute-intensive functions within an application can be identified. These functions can then be compiled into hardware and migrated into the PL for higher performance.

At the interface between hardware and software is a communication mechanism that allows data exchange between the two. The parameters of the accelerated function are passed to or made available to the hardware accelerator in PL, and the result of the hardware computation is returned or made available to software.

This communication can be done over one of the PS AXI ports (AXI_ACP, AXI_HP, or slave AXI_GP) using any of the following data movement schemes:

- Memory-mapped registers
- AXI-Stream FIFO
- AXI-DMA

The choice of the port and the data mover employed by the hardware is influenced by the data transfer size, the resources required to implement the communication mechanism, and any latency requirement. Some of the most commonly used solutions are discussed below.

Implementing memory-mapped registers in the accelerator provides the simplest form of data communication. In one method, the accelerator is an AXI-slave. Software writes the required computation parameters into the registers and starts the hardware. When the hardware is done, the accelerator either interrupts the PS or writes to a status register-bit polled by software. In low power applications, using the event inputs may be preferred over interrupts. The software can then obtain the result with a series of register reads. A block diagram of this method is shown in Figure 2-14.

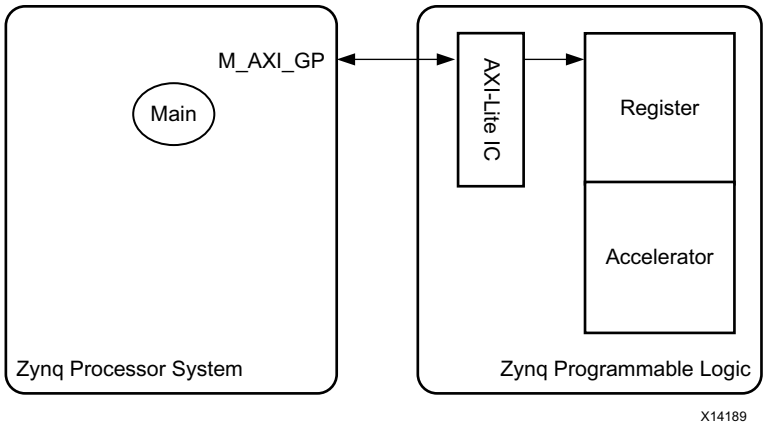

Figure 2-14: **Memory Mapped Register**

Alternatively, the accelerator can act as an AXI-master and pull data from system memory or a PS peripheral as directed by software. A block diagram of this method is shown in Figure 2-15.

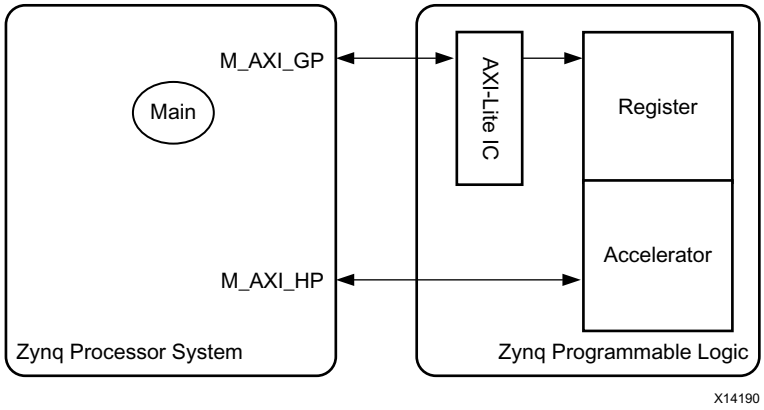

Figure 2-15: **Memory Mapped Register With AXI Master**

While the register-based solution is simple to implement, it works best with accelerators that have a small data set to transfer across the software/hardware boundary. There is a software overhead for accessing individual registers, which makes the data transfer expensive. With larger data sets, an AXI FIFO can be employed for streaming type interfaces. A block diagram of this method is shown in Figure 2-16.

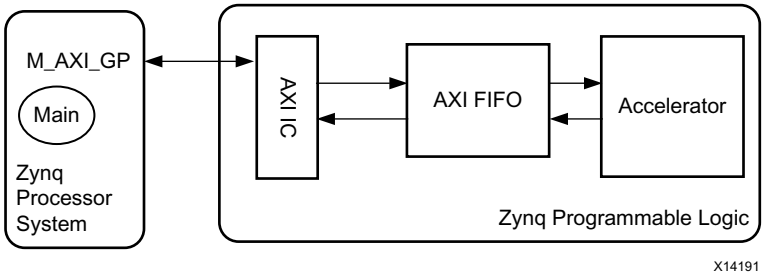

Figure 2-16: **Memory Mapped FIFO**

Very often, the hardware accelerator is used to process a large amount of data, such as in a video based application. The data is stored in memory accessible by both software and hardware in the PL, such as OCM or DDR. A DMA is used to pull data out of the memory and push the data back into memory after accelerator processing. A block diagram of this method is shown in Figure 2-17.

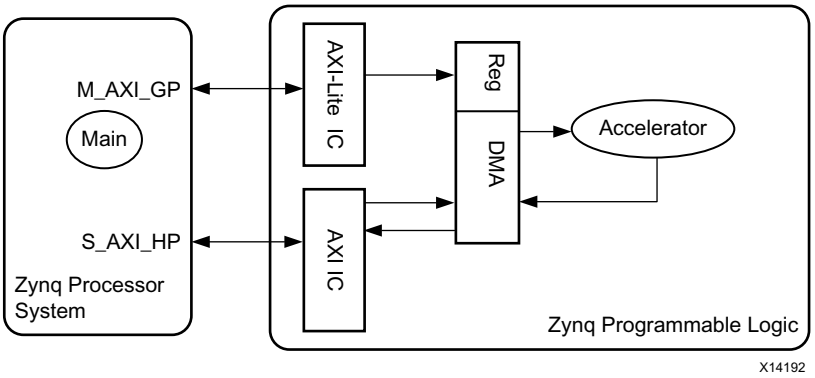

Figure 2-17: **Memory Mapped DMA**

The *Zynq-7000 All Programmable SoC ZC702 Base Targeted Reference Design (Vivado Design Suite 2014.2) User Guide* (UG925) [Ref 13] is a video-processing application that implements an edge-detection filter on a 1080p60 video stream in both software and hardware. In the hardware implementation, a video DMA pulls video data out of the DDR, pipes it through the edge-detection engine, then writes the video stream back into DDR. Software updates the DMA registers to control the flow of video data.

Hardware Design Considerations

This chapter covers the following hardware design issues to be considered when designing with a Zynq®-7000 AP SoC:

- Configuration and Boot Devices, page 63: You can boot from a variety of primary and secondary boot devices. This section describes the boot flow and boot device options.
- Memory Interfaces, page 69: You can connect the DDR memory controller to a variety of DDR memory devices, using either a 16-bit or 32-bit wide data bus. An ECC option is supported.
- Peripherals, page 76: A variety of peripherals are available in the Zynq-7000 AP SoC. This section describes the peripherals and their interaction with the Application Processing Unit (APU).
- Designing IP Blocks, page 94: IP blocks are pre-designed, pre-verified, and reusable functional blocks that can help you reduce design time. This section describes the process of designing solutions containing IP blocks.
- Hardware Performance Considerations, page 102: This section describes hardware performance metrics and methods you can use to tune the performance of AXI masters, AXI slaves, and AXI datapaths.
- Dataflow, page 108: Dataflow within the processing system (PS) and between the PS and programmable logic (PL) is described in this section.
- PL Clocking Methodology, page 112: The PL clocking methodology is covered in this section. The different PL clock sources and their recommended use are described.
- ACP and Cache Coherency, page 116: The ACP's ability to provide low-latency access to PL masters, including optional coherency with the L1 cache, is described in this section.
- PL High-Performance Port Access, page 120: You can use the HP ports to give the PL direct access to the DDR controller and the on-chip memory (OCM). Design-driven optimizations that use the HP ports are described in this section.
- System Management Hardware Assistance, page 124: System management, including the control of system-level parameters based on user-specific inputs, is discussed in this section.
- Managing Hardware Reconfiguration, page 127: You can apply partial reconfiguration to the portion of the FPGA that is not static logic. This section describes the process of partially reconfiguring your design.
- GPs and Direct PL Access from APU, page 133: This section describes how you use the GP interfaces for accessing registers and memory in the PL from the APU.

Configuration and Boot Devices

This section describes Zynq-7000 AP SoC boot devices and how they are configured. It covers the boot flow and the role of BootROM as an agent for performing the boot initialization sequence. Primary boot options like Quad-SPI, SD, NAND, NOR flash, and JTAG boot mode, and secondary boot options like eMMC and PCIe in the Zynq-7000 AP SoC are described. The section focuses on the boot flow and boot device usability in system-level scenarios. The JTAG boot mode is considered a slave boot mode and is always a non-secure boot mode.

Typical Boot Flow

The boot sequence in a Zynq-7000 AP SoC involves reading the first-stage boot loader (FSBL) from external static storage, typically NAND or NOR flash memory. The BootROM contains the code that is executed after power on reset, and that code reads the FSBL from the external static storage.

After the PS_POR_B pin is de-asserted, hardware samples the boot-strap pins and configures the PS and the PLLs. Software in the PS internal BootROM executes, beginning with code that configures the ARM core and peripherals necessary to copy the FSBL image from the chosen boot device to OCM, and then the code switches execution from the OCM. You can copy the FSBL from the primary boot device mentioned above or you can load it through JTAG. Optionally, in non-secure boot mode, you can execute the FSBL from Quad-SPI/NOR flash that supports execute-in-place.

Figure 3-1 depicts the typical boot flow.

Figure 3-1: **Typical Boot Flow**

You can boot a Zynq-7000 AP SoC in both secure and non-secure modes. Secure mode boot is only possible with static memories whereas non-secure mode boot is possible with both JTAG and static memories. JTAG boot is primarily used during the system development phase.

Selecting a Boot Device

The following primary boot options are available for Zynq-7000 AP SoCs:

- Quad-SPI with optional execute-in-place mode
- NAND flash
- NOR flash with optional execute-in-place mode
- SD memory card
- JTAG

The following secondary boot options are available for Zynq-7000 AP SoCs:

- eMMC
- PCIe, Ethernet, USB, UART, or a custom FPGA interface

You can select a BootROM device using any of the following criteria:

- The device works with the Zynq-7000 AP SoC BootROM and is supported by Xilinx® tools (iMPACT, SDK) and higher-level software such as U-Boot and Linux. You can find a list of Xilinx-recommended devices for different boot options in *Xilinx Answer Record 50991* [Ref 67].
- The device meets the application's configuration pin requirements. For example, the QSPI boot option requires fewer pins than the NAND or NOR boot options.
- The device meets the application's size requirements. NAND and SD boot options provide higher memory density than the QSPI and NOR boot options.
- The device meets the application's configuration speed requirements. QSPI is the fastest available boot option.
- Device management can be adequately handled by the application. NAND devices can be more difficult to manage than other boot options. Bad blocks require design decisions on how they will be managed when they occur.

Your choice of boot device affects the number of pins required, the maximum memory size, the boot time, and the device management software complexity. Table 3-1 summarizes the design trade-offs.

Table 3-1: Flash Memory Comparison

Peripheral	Pins	Boot Device	XIP	Limitations	Read/Write	Max Size	Boot Time
QSPI	7 single, 8 dual stacked, 13 dual parallel	Yes. Boot image in 16MB single and stacked, 32MB for parallel.	Yes	Capacity limited to 16MB per device as a boot device. Very slow erase/write – usually used as read only. Often used in with eMMC or SD card.	Byte/Block	16MB in Liner mode, 128MB in IO mode, per QSPI. Each controller can support up to two QSPIs.	Fast
NAND	X8–14 X16–22	Yes Boot image in the first 128MB	No	Requires software management of ECC, wear leveling, bad block. Requires file systems for raw flash such as JFFS2 or UBIFS. Requires ECC. Hardware support for 1-bit ECC only.	Block/Block	1GB	Medium
SD	6	Yes	No	Requires Mechanical connector	Block/Block	Any size	Slow
Parallel NOR	37	Yes	Yes	Support capacity comparable to large QSPI, but with high pin count.	Byte/Block	64MB	Fast
eMMC	6	No	No	Cannot boot from eMMC	Block/Block	Any size	N/A

More information on each boot option is provided in the following sections.

Quad-SPI with Optional Execute-in-Place Mode

The BootROM can detect the Quad-SPI interface's intended I/O width using the width-detection parameter value (`0xAA995566`) and image-identification parameter value (`0x584C4E58`). Quad-SPI is the fastest configuration solution available. Only the SD memory card boot option has a lower pin count. You can access Quad-SPI as linear memory in Zynq-7000 AP SoCs. Device management is simpler because bad blocks are of less concern than they are in other devices, such as NAND flash. In linear mode, Quad-SPI supports up to 16 MB in single mode and 32 MB in dual mode. More than 16 MB is also supported when the QSPI is operated in IO mode. For the execute-in-place option, the BootROM uses the linear addressing feature of the Quad-SPI controller for non-secure boot modes. Multiboot is possible with QSPI.

Micron (N25Q) and Spansion (25FL) are the recommended Quad-SPI boot device families.

NAND Flash

The NAND flash boot option is an inexpensive solution supporting large density devices. The only restriction is the boot image must be located within the first 1 GB of address space in the NAND flash device. Performance of this boot option is lower than that of the Quad-SPI boot option. Typical NAND flash solutions require more pins and have lower memory bandwidth than Quad-SPI solutions. The application design needs to have a mechanism for managing bad clocks, including during boot. Multiboot is possible with NAND.

Micron (on-die ECC) and Spansion (S34) are the recommended NAND boot device families. Micron NAND devices typically require multi-bit ECC, which necessitates using only devices with on-die ECC support.

NOR Flash with Optional Execute-in-Place Mode

You can access NOR flash as linear memory with Zynq-7000 AP SoCs. Also, bad blocks are of less concern than they are in NAND flash devices. NOR flash density is comparable to Quad-SPI. It uses more MIO pins compared to other boot options. A typical NOR flash (byte peripheral interface) uses 40 MIO pins and can support up to 64 MB.

Micron (M29EW) and Spansion (29GL) are the recommended NOR flash boot devices.

SD Memory Card

SD memory cards have higher densities than that of the NAND flash boot option. Devices are generally managed as a file system. Bad blocks do not need to be managed by the application design. SD memory is slower than the Quad-SPI boot option, and the SD cards require an on-board connector. eMMC devices are not primary boot devices but you can use them as a secondary boot source. In SD boot mode, BootROM does not perform header search and a multi-boot option is not supported. You can improve boot time in the SD boot mode by setting the CPU clock divisor to 2 in the ARM_CLK_CTRL register (`0x1F000200`).

JTAG

In JTAG boot mode, you can select the independent JTAG mode to support a debugger connected to the ARM DAP controller and another tool connected to the Xilinx PL TAP controller. The Xilinx PL TAP controller is accessible using the PL JTAG interface connected to dedicated PL pins. Access to the ARM DAP controller (for another tool) is done using the EMIO JTAG interface. This requires downloading a bitstream into the PL. You can download the bitstream using the Xilinx PL TAP controller.

eMMC

You can use the eMMC secondary boot option when QSPI is the primary boot option and a small QSPI memory is used. Typically, the FSBL will be loaded in to the QSPI and other partitions will be loaded into eMMC. Xilinx recommends using eMMC with the SDIO controller only in standard speed mode (max frequency of 25 MHz).

PCIe, Ethernet, USB, UART, and Custom FPGA Interface

You can also implement secondary boot using PCIe, Ethernet, USB, UART, or a custom FPGA interface. You can select an option appropriate to your application. The PCIe-based secondary boot option is discussed below.

PCIe secondary boot uses the PCIe protocol to fetch the second-stage boot loader from a host system that implements a PCIe root complex. The PCIe block in the Zynq-7000 AP SoC PL implements the end point function and forms the communication link with the PCIe root complex. You can fetch the FSBL image from external memory (QSPI, NAND, or NOR flash) and after FSBL execution is done, the FSBL can fetch the secondary U-Boot image from the host-system memory over PCIe.

To implement this option, you need to generate a bitstream instantiating the PCIe block in the design. For example, you can implement the programmed IO (PIO) design using a Xilinx 7 Series PCIe IP block. You also need to implement a set of PL registers for handshake status between the APU and the host CPU, and connect the register interface to the PS master GP port. The boot flow is:

- The FSBL will be loaded from the primary boot device.
- The APU waits until the FSBL performs the peripheral initialization process.
- The FSBL programs the bit file with the PIO design and handshake status register set.
- The APU programs the bitDone register after the bit file programming is done.
- The host PIO driver reads the bitDone register and writes the `U-Boot.elf` file to PL block RAM.
- After `U-Boot.elf` is written, the status is written to the U-BootDone register.
- The APU polls the U-BootDone bit, and after it is done copies the file to PS DDR memory.

File System

The flash choice influences the file system that can be implemented. The file system choices described below are specific to Linux, but provide a good outline of the issues to be considered.

eMMC and SD cards have a built-in controller that runs the Flash Translation Layer (FTL) firmware, allowing the device to appear to the OS as a block device. Conventional file systems, such FAT or ext3, work with block devices and can be implemented using these devices.

A raw flash device (such as NAND, QSPI, or parallel NOR) requires software management so that reads-from and writes-to memory cells function properly. Linux uses the memory technology device (MTD) subsystem to provide an abstraction layer between the hardware-specific device drivers and higher-level applications. Linux supports file systems that are layered on top of MTD devices, such as JFFS2 and UBIFS. These file systems are designed to include software management algorithms to handle issues like wear leveling and bad block management, and must be used for the device to function properly.

Optimizing Boot Time

The BootROM settings used to read from each flash device type are selected for maximum compatibility, often at the expense of performance. Boot interface performance can be improved by setting the corresponding controller registers in the register initialization portion of the BootROM header. The settings depend on the devices being used and the board layout parameters. Optimized register values should be obtained from vendor data sheets for the devices used. Examples of optimized values for each boot device can be found in the Register Initialization to Optimize Boot Times section of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] and the *Xilinx Answer Record 55572* [Ref 69]. The optimized values can be added to the boot image header using Bootgen. Refer to the "Using Bootgen" appendix in the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7] for more information. The BootROM Performance section of the Boot and Configuration chapter in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] provides methods to improve timing and bandwidth results during different boot stages.

Case Study: Variations in QSPI Boot Time

This case study describes boot-time measurements done on a Xilinx ZC702 board using different QSPI devices running at 100MHz, and varying the PS frequency.

The boot time (measured from U-Boot start to the Linux prompt) for a 32 MB Spansion single device with the PS running at 667 MHz is approximately 3036 milliseconds. The boot time a for a 64 MB Spansion dual-parallel device with the PS running at the same frequency is approximately 2994 milliseconds. Therefore, using a dual-parallel QSPI device that is twice the size of a single QSPI device improves boot time by approximately 41 milliseconds, without changing the PS frequency.

Changing the PS frequency also affects boot time. As mentioned above, the boot time for a 32 MB Spansion single QSPI device with the PS running at 667 MHz is approximately 3036 milliseconds. Using the same QSPI device but increasing the PS speed to 867 MHz results in a boot time of approximately 2523 milliseconds. Therefore, without changing the QSPI device, increasing the PS operating frequency from 667 MHz to 867 MHz improves boot time by approximately 513 milliseconds.

Memory Interfaces

DDR

The DDR multi-protocol memory controller in the Zynq-7000 AP SoC supports 1.8V DDR2, 1.2V LPDDR2, 1.5V DDR3, and 1.35V DDR3L. It can be configured to provide a 16-bit or 32-bit wide data bus. All devices support the 16-bit and 32-bit data bus width options, except the 7z010 CLG225 device that supports only the 16-bit data bus width. The controller optionally supports ECC in 32-bit configurations, with 16 data bits and 10 check bits. When ECC is enabled the data width is limited to 16 bits. A 1 GB address map is allocated to the DDR. However, if ECC is used, only 512 MB of address space is available.

The DDR memory controller contains three major blocks: an AXI memory port interface (DDRI), a core controller with a transaction scheduler (DDRC), and a controller with digital PHY (DDRP). The details of each block and other controller aspects are described at this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

The controller includes a digital PHY with a dedicated set of I/Os. On the fastest speed grade for some Zynq-7000 AP SoC devices, the maximum supported bus clock is 666 2/3 MHz in DDR3 mode. The maximum supported bus clock is 533 MHz in DDR3 mode for all other speed grades. The theoretical maximum bus bit-rate is:

```
Data transfer rate = 666 2/3 MHz * 2 bits (for double data rate) = 1333 Mb/s per data IO
```

Using the maximum bus width of 32 bits, the maximum bus bandwidth is 42.6 Gb/s, or 5.3 GB/s. This bandwidth is shared by multiple masters connected using the four DDRI slave ports as shown in Figure 3-2. The DDRI block connects to four 64-bit synchronous AXI interfaces to serve multiple AXI masters simultaneously. Each AXI interface has its own dedicated transaction FIFO. Port S0 is connected to the L2-cache and services only the PL CPU and ACP interfaces to ensure low latency and fast access. Port P1 is shared by all central-interconnect masters, such as PS peripherals and AXI GP ports. The four PL AXI_HP interfaces are multiplexed down in pairs and are connected to ports 2 and 3, as shown in Figure 3-2.

The maximum bus bandwidth is 5.3 GB/s, but it is not sustainable. The DDR efficiency must consider the overhead associated with DDR and is influence by the data access pattern. Address patterns that minimize page and row changes reduce page and row change overhead, resulting in higher utilization and higher system throughput. Refer to the "Row/Bank/Column Address Mapping" subsection of the "DDR Memory Controller" chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

How PL AXI masters are connected to the HP ports also determines system throughput. Multiple AXI masters are more likely to produce random address accesses that cause more page misses, resulting in lower DDR efficiency. In Figure 3-2, the AXI_HP to DDR interconnect multiplexes four PL AXI_HP down to two pairs connected to ports S2 and S3, both arbitrated by the interconnect. When there are only two PL AXI masters, higher performance can be achieved by connecting the masters to port AXI_HP0 and AXI_HP2, or AXI_HP1 and AXI_HP3, thus using both port 2 and 3 of the DDR controller. In a typical Linux-based video design, achievable system bandwidth from the AXI_HP ports is about 50%. This is described in *Designing High-Performance Video Systems with the Zynq-7000 All Programmable SoC* (XAPP792) [Ref 34].

The controller DDRC block includes a three-stage arbiter for improved DDR control latency. Latency can be controlled using register settings. For details on managing DDR latency, refer to the "DDRC Arbitration" subsection of the "DDR Memory Controller" chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

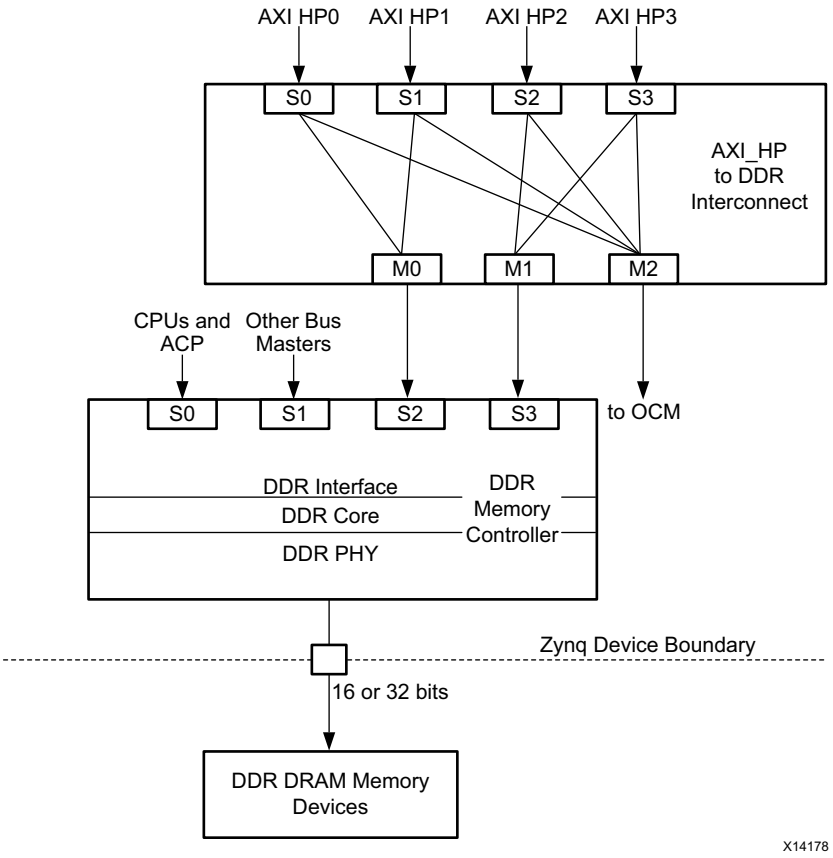

Figure 3-2: **DDR Memory Controller Block Diagram and Interconnect to AXI_HP Ports**

To facilitate high-bandwidth operation, the DDR interface timing must be properly initialized and calibrated. DDRP includes a DRAM training feature to help automatically determine the timing delays required to align data to the optimal window for reliable data capture. This feature is described in the "Initialization and Calibration" subsection of the "DDR Memory Controller" chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

The Zynq-7000 AP SoC tool flow helps automate the DDR bring-up. To do this, the PS DDRC board parameters need to be configured in accordance with *Xilinx Answer Record 46778* [Ref 65]. This will import the delay characteristics of DDR signals on the board during the hardware design process. Those characteristics are used to determine the initial values used by the automatic training algorithm, or for calculating static interface timing when the automatic algorithm is not supported by the particular DDR standard. The timing values are part of the design's hardware platform specification and are exported to the SDK and used in PS initialization code that is called by the FSBL. DDR is not used by the BootROM.

QSPI

The PS QSPI flash controller communicates to an external serial flash memory using the QSPI interface. The memory flash cells are arranged in parallel and are sometimes referred to as NOR flash. While this configuration is less dense and has smaller capacity than NAND flash memory, it allows single-byte reads anywhere in the array. For reads, it can behave like a standard address-mapped memory and is well-suited for code storage. Also, it supports the execute-in-place (XIP) feature where a CPU can execute code directly out of QSPI without reading the code into DDR or OCM first. The *Zynq-7000 AP SoC Boot - Booting and Running Without External Memory Tech Tip* wiki page [Ref 59] provides a "hello world" example design on a ZC702 board using XIP.

Of all the flash memories supported by a Zynq-7000 AP SoC, QSPI has the highest read performance and provides the fastest boot solution with a very low pin count requirement. It supports page-based write and sector-based erase, but at a substantially slower rate compared with other types of devices. For a cost-effective solution with fast boot time and large memory capacity, you can use a small QSPI as the primary boot device for storing the FSBL. All other partitions can be placed in a larger flash, such as eMMC or SD. For information on how to implement such a system, refer to the "eMMC Flash Devices" section (available at this link) in the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

The QSPI flash controller supports three different modes of operation: I/O mode, linear-addressing mode, and legacy SPI mode. In linear-addressing mode, the AXI transactions received by the QSPI controller are automatically translated into the corresponding command and data transactions on the QSPI bus connected to the flash device. The controller supports only 24-bits of flash address, therefore the maximum size of QSPI is limited to 16MB in linear mode. BootROM accesses the QSPI in linear mode. When QSPI is used in XIP mode, the length of the FSBL is limited to the capacity of the QSPI in linear mode, minus the length of the boot header. When XIP mode is enabled (through the boot header), instead of copying the FSBL to OCM, BootROM hands control to the FSBL executing directly from QSPI flash. When QSPI is not used in XIP mode, BootROM copies the FSBL into OCM for execution. In this case, the size of FSBL is limited by the 192 KB OCM capacity. The first three quarters of OCM is located at low memory, and the top one quarter is mapped to high memory. The FSBL could use most of the top one quarter for items such as stack and heap.

In I/O mode, software composes the command and data in the TXD registers, and the controller drives the content of the register to the flash memory in the proper format. The data driven by the flash memory is shifted into the corresponding RXD registers and the data is extracted by software. Using I/O mode, the software can issue flash commands to specifically modify specific register bits inside the flash memory, thus expanding the QSPI flash address space by effectively switching between pages. Using this method, the controller can support up to 128MB per QSPI. For more information on QSPI I/O mode, refer to the I/O mode section in the Quad-SPI Flash Controller chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

When a QSPI larger than 16MB is used on the board, any Zynq-7000 AP SoC platform reset must also trigger a reset to the QSPI so that the page register is reset. This ensures that when BootROM reads from address 0x0 in linear mode it can access the boot image. Refer to *Xilinx Answer Record 57744* [Ref 70] for more information on the reset requirement when using flash devices larger than 16 MB.

If either RSA encryption or XIP mode is used to store boot images in flash devices larger than 16 MB, the boot image cannot be placed at offset 0x0. Instead, the boot image can be located at offset 0x0+32K. If the image is located at 0x0, a duplicate header image can be located at offset 0x0+16MB, or a single x1 QSPI can be used.

Different QSPI flash devices require different dummy clock cycles depending on the QSPI clock frequency and the type of command used to perform the read. Depending on the way the QSPI is configured on the board (x1, x2, x4, single, stacked, or parallel) and the particular flash device used, the controller's LQSPI_CFG register must be set appropriately for proper communication with the flash device. When using Xilinx-supported QSPI flash devices, the Xilinx PS QSPI device driver automatically writes the appropriate values into the LQSPI_CFG register. Xilinx supports QSPI devices manufactured by Micron and Spansion. For information on which vendor flash devices are currently supported by the Zynq-7000 AP SoC tools, refer to *Xilinx Answer Record 50991* [Ref 67].

BootROM automatically issues read commands using either fast read (x1), quad output read (x4), dual-output fast read (x2), single mode, or parallel mode by examining the width-detection value and the image-identification value in the BootROM header. Based on the values, BootROM uses the widest supported I/O bus width to read data from the Quad-SPI device, but sends commands in x1. BootROM writes a set of initial values into the LQSPI_CFG register. Further details on those values can be found in the Quad-SPI Boot subsection of the Boot and Configuration chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

In a high-speed QSPI application where the memory-interface clock is greater than 40 MHz, QSPI feedback mode must be used. For more information on the QSPI Feedback clock, see the Quad-SPI Feedback Clock subsection in the Quad-SPI Flash Controller chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

The QSPI can be programmed by U-Boot, Linux, iMPACT, and the SDK.

The QSPI flash controller details can be found in the Quad-SPI Flash Controller chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

Static Memory Controller

The static memory controller has two interface modes: a NAND flash interface mode and a parallel port memory interface mode. The NAND flash interface mode supports NAND flash, while the parallel-port interface mode supports NOR flash and asynchronous SRAM. Because both QSPI and NOR flash use NOR-based memory cells, and because QSPI is limited to 16MB in linear mode, capacity is the differentiating factor in selecting NOR flash over QSPI. However, because the static memory controller limits the number of address lines to 26 bits, it supports NOR flash sizes only up to 64MB. The 16MB QSPI limit applies only during boot (the amount of data read by BootROM must not exceed 16MB). After BootROM loads FSBL into either OCM or DDR memory, the QSPI controller can switch into I/O mode and access up to 128MB of memory with the support of the flash-device's page register. Consequently, the NOR-flash controller's ability to support 64MB devices (at the expense of 40 pins) as compared to 128MB devices supported by QSPI (using just 8 pins) makes QSPI a preferred solution over NOR flash.

The following section describes the NAND-flash interface mode. For more information about using the parallel-port memory interface, refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

NAND Memory Controller

In NAND flash the memory cells are laid out in series, resulting in more densely-packed memory cells, higher capacity per silicon area, and cheaper per-bit cost than NOR flash. The NAND flash controller can support up to 1GB of external NAND flash with either an 8-bit or 16-bit I/O bus for address/data/command. It supports the Open NAND flash Interface 1.0 specification.

The dense memory-cell packing causes the cells to be stressed during programming and erase cycles, making them more prone to bit errors. ECC is used to mitigate those errors. The NAND flash controller includes hardware support for 1-bit ECC correction. Software is used to run flash management algorithms that make use of the ECC data to manage the various error modes, deal with bad blocks, homogenize wear across the memory cells, and improve cell endurance and data retention.

Because the controller only supports 1-bit ECC, only NAND devices with on-chip ECC or one-bit ECC can be used with Zynq-7000 AP SoCs. Currently, only single-level-cell (SLC) devices meet the ECC criteria and multi-level-cell (MLC) devices are not supported. Also, the controller only supports a single chip select. Xilinx supported NAND devices range in size from 128MB to 1GB, in both x8 and x16 configurations. Xilinx officially supports NAND devices from Micron and Spansion. For information on which vendor flash devices are supported by the Zynq-7000 AP SoC tools, refer to *Xilinx Answer Record 50991* [Ref 67].

Even in a high-capacity NAND device, the Zynq-7000 AP SoC BootROM will look for a boot image to start in the first 128MB. Therefore, the start of both standard and fallback boot images (if used) must begin in the first 128MB.

Communication with the NAND flash is based on a set of AC timing parameters that vary from device to device. For a Zynq-7000 AP SoC to communicate with a NAND device using the correct timing, a designer should input the relevant timing parameters into the CS0 column on the SMC Timing Calculation page based on the device's AC timing values. This is shown in Figure 3-3.

The CS0 cycle is automatically calculated based on the NAND clock frequency. The values are exported to the SDK's PS initialization code as part of the design's Hardware Platform Specification. The initialization code writes those values to the smc.SET_CYCLE register.

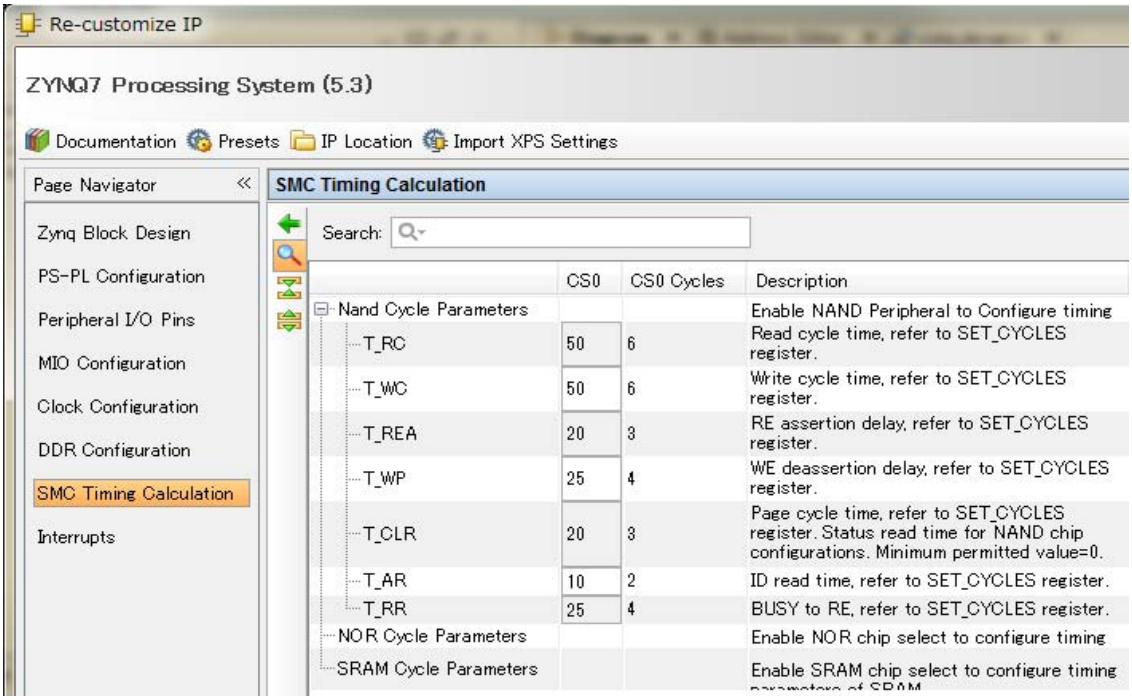

Figure 3-3: **SMC Timing Page Example**

BootROM reads from NAND flash prior to executing PS initialization code, using a set of initial values stored in the smc.SET_CYCLE register. Further details on those values can be found in the NAND Boot subsection of the "Boot and Configuration" chapter in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

In NAND flash, reads and writes occur in pages, while erasures occur in blocks. Because NAND flash does not behave like random access memory, Linux systems use it as a memory technology device (MTD) that provides an abstraction layer, allowing software to access the device using the MTD subsystem API. That API is common among different flash types and technologies. MTD is not a block device and it lacks the software management algorithms to handle issues like wear leveling and bad block management. Instead of traditional file systems like ext2, ext3, and FAT (which work on top of block devices), the file system must be designed to work on top of raw flash, such as JFFS2 or UBIFS. JFFS2 works on top of MTD subsystems. UBIFS works on top of UBI subsystems, and those work on top of MTD subsystems to provide software management algorithms required for NAND devices.

Peripherals

The memory-mapped peripherals present in the Zynq-7000 AP SoC PS, and the interaction of those peripherals with the APU and the memory controller, are described in this section. It is assumed that the reader is familiar with the AMBA bus topology (AHB and APB) and has an understanding of the USB, CAN, UART, Ethernet, and SPI protocols.

The Zynq-7000 AP SoC peripherals can be broadly categorized as:

- PS peripherals
- PL peripherals

The PS peripherals are hard-wired peripherals and are implemented as part of the standard ASSP implementation. The PL peripherals are programmable and reconfigurable on the fly. This section covers the PS peripherals and the PL peripherals are covered in the following section.

The PS peripherals are memory mapped and the peripheral memory map is predefined. Peripherals implement the Advanced Microcontroller Bus Architecture (AMBA) protocol-compliant bus architecture and are connected to either the central interconnect or part of the application processor unit. They communicate with the ARM Cortex-A9 processor using AMBA-specified transactions. The peripherals are ARM TrustZone security aware and can optionally deny non-secure accesses coming from a non-secure master. Peripherals connected to the central interconnect can optionally be routed to the PL that enables PL access.

Figure 3-4 shows the top-level block diagram of the peripherals present in PS.

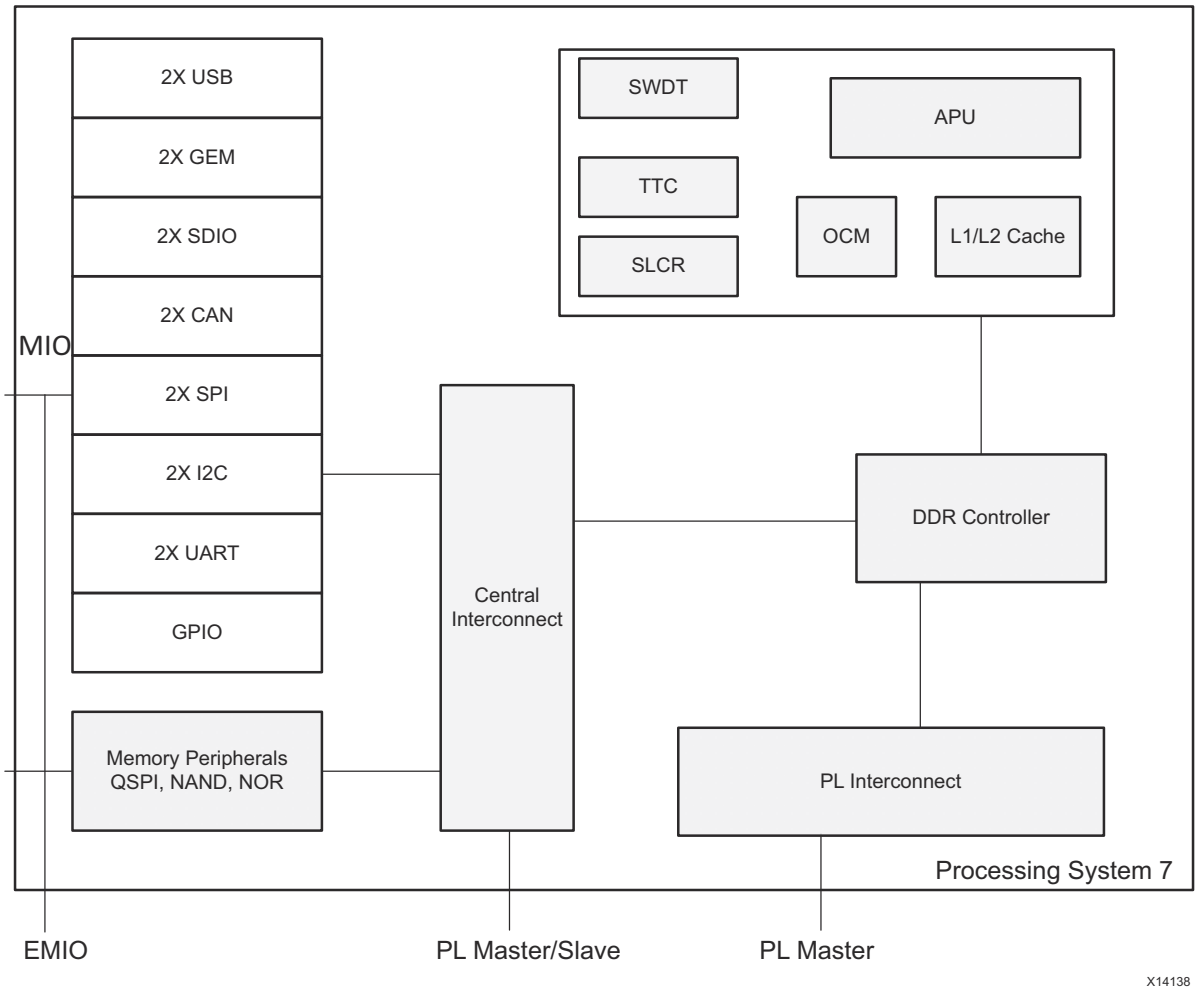

Figure 3-4: **PS Peripheral Block Diagram**

Peripheral Descriptions

The following are key peripherals present in the PS block.

USB Peripheral

The PS USB controller is USB 2.0 compliant and can operate in any of the following modes:

- Device mode
- Host mode
- On-The-Go (OTG) mode

The USB controller attaches to an external PHY using the ULPI interface, which is implemented as an 8-line SDR data bus using the MIO. The ULPI PHY is not part of the USB controller, and you can implement any ULPI-compliant PHY to connect to the USB controller in the Zynq-7000 AP SoC. The port-controller indication, power select, and power failure signals can be routed to the PL via EMIO. The sideband signals can also be routed to the SelectIO™ pins. The USB controller has a built-in DMA engine with a transmit and receive FIFO that transfers the data to and from memory using the AHB bus.

Refer to the USB Host, Device, and OTG Controllers chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

Data Flow

The data transferred by the DMA from memory is encapsulated using the USB 2.0 protocol format and is transmitted over the ULPI interface to the PHY.

System Level Considerations

The USB controller is clocked from two clock domains:

- The AHB interface is clocked by the CPU_1x clock domain.
- The USB protocol engine and the port controller interface use the 60 MHz clock generated by the ULPI PHY.

You can use three different methods to reset the USB controller:

- You can trigger a controller reset with a PS reset by writing to the USB command register using the APB interface.
- You can trigger a ULPI PHY reset under the control of a GPIO signal.
- You can trigger a USB bus reset with the auto reset feature in OTG mode.

You can configure the controller as a USB host, USB device, or as an OTG. The configuration is determined by your application. The controller host mode requires the USB host controller interface be built in the Linux image. You can set different device mode options, either USB mass storage, isochronous, or interrupt-based. For a high-speed endpoint, the maximum packet size is 512 bytes for bulk transfers and 1024 bytes for isochronous devices. You can use isochronous mode for streaming applications, such as video.

Ethernet Peripheral

The gigabit Ethernet MAC (GEM) implements a 10/100/1000 Mb/s Ethernet MAC compatible with the IEEE 802.3-2008 standard. It is capable of operating in either half mode or full-duplex mode at all three speeds. You can configure each of the two GEM controllers independently. To save pins, each controller uses an RGMII interface through the MIO. Access to the PL is through the EMIO which provides the GMII interface.

You can create other Ethernet communication interfaces in the PL using the GMII available on the EMIO interface.

Because the GEM controller in the Zynq-7000 AP SoC provides only the MAC function, you must implement a separate PHY.

Data Flow

The DMA block present in the controller fetches data from system memory using the AHB bus and stores data in the transmit FIFO. The FIFO data is converted to Ethernet protocol format by the controller and is sent to the Ethernet PHY using an RGMII or GMII interface, depending on whether the user selects the MIO or EMIO interface.

Registers are used to configure the features of the MAC, select different modes of operation, and enable and monitor network management statistics

The controllers provide MDIO interfaces for PHY management. You can control the PHYs from either of the MDIO interfaces.

System Level Considerations

The GEM controller offers a number of system-level features that can be implemented based on system requirements. For example, you can enable checksum off-loading in a system that requires better network efficiency and less CPU utilization. Another system-level consideration is implementation of precision time protocol (PTP) as defined in IEEE 1588. The controller detects and responds to PTP events and requires a PTP compliant network peer.

You can optionally route Ethernet packets from the controller to the PL using the DMA interface. This option may require implementation of packet acceleration or packet inspection IP in the PL.

The GEM controller does not support jumbo frames. If you require jumbo frame support, you can implement the Xilinx AXI Ethernet MAC IP in the PL. You can use the AXI DMA IP to read and write Ethernet frames to and from PS DDR memory. The GEM controller does not support 3.3V IO voltage. Refer to the Gigabit Ethernet Controller chapter (available at this link) in *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

SDIO Peripheral

The SD/SDIO controller communicates with the SDIO devices and SD memory cards. You can route the SDIO interface through the MIO multiplexer to the MIO pins, or through the EMIO to SelectIO pins in the PL. The controller can support SD and SDIO applications in a wide range of portable low-power applications such as 802.11 devices, GPS, WiMAX, and others.

The SD/SDIO controller is compatible with the standard SD Host Controller Specification Version 2.0 Part A2 with SDMA (single operation DMA), ADMA1 (4 KB boundary limited DMA), and ADMA2 support. ADMA2 allows data at any location and any size to be transferred in a 32-bit system memory using scatter-gather DMA. The core also supports up to seven functions in SD1 and SD4, but does not support SPI mode. The core does support the SD high-speed (SDHS) and SD High Capacity (SDHC) card standards. You should be familiar with the SD2.0/SDIO 2.0 specifications [Ref 89].

The SD/SDIO controller is accessed by the ARM processor via the AHB bus. The controller also includes a DMA unit with an internal FIFO to meet throughput requirements. The SD/SDIO controller complies with the MMC 3.31 specification.

Data Flow

The SD/SDIO controller has a DMA built into the controller that can fetch data from system memory using the AHB bus. The controller implements a pair of FIFOs to read and write data to the attached card to maximize throughput. The data is written to the attached card or read back based on an SD-specific command initiated by the APU.

System Level Considerations

SD cards are essentially NAND flash devices with built-in controllers that implement the flash translation layer (FTL). The FTL handles ECC, block management, and wear leveling so that the memory behaves like a block device. Because of this, conventional file systems (such as FAT, ext2, and ext3) can be implemented. Because of multi-chip packaging technologies and modern advances in multi-level cell technology, SD memory devices provide the highest density and capacity of all flash memory choices available in a Zynq-7000 AP SoC. The drawback of SD memory is the requirement for a mechanical SD card connector. When a physical connector is undesirable, eMMC is an alternative. eMMC consists of flash memory and a controller packaged in a small ball grid array (BGA) that can be directly mounted onto a circuit board without a mechanical connector. The SD and eMMC solutions differ in their ability to function as boot devices. A Zynq-7000 AP SoC can boot directly from an SD card, but it cannot do so from an eMMC. eMMC solutions require an additional boot device, such as QSPI.

In addition to the CLK, CMD and Data signals, card detect (CDn) signals indicate the insertion or presence of the SD card, and write protect (WPn) signals indicate the position of the write protect switch on the memory card. Some software drivers rely on the use of these signals. Thus, if the signals are not available, a simple solution is to tie CDn active and WPn inactive.

You can also use the SD/SDIO peripheral as a boot mode device. Refer to Configuration and Boot Devices, page 63 and to the SD/SDIO Controller chapter (available at this link) in *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

UART Peripheral

The UART controller is a full-duplex asynchronous receiver and transmitter that supports a wide range of programmable baud rates and I/O signal formats. The controller can accommodate automatic parity generation and multi-master detection mode.

The UART operations are controlled by the configuration and mode registers. The state of the FIFOs, modem signals, and other controller functions are read using the status, interrupt status, and modem status registers.

Data Flow

The communication between the controller and the APU occurs over the APB bus. The controller is structured with separate Rx and Tx data paths. Each path includes a 64-byte FIFO. The data that arrives from system memory over the APB bus are stored into the transmit FIFO. The received data from the MIO/EMIO interface is stored in the receive FIFO and transmitted to system memory using the APB bus.

The controller serializes and de-serializes data in the Rx and Tx FIFOs and includes a mode switch to support loopback configurations for the RxD and TxD signals. The FIFO interrupt status bits support polling or an interrupt-driven handler. Software reads and writes data bytes using the Rx and Tx data port registers.

System Level Considerations

Zynq-7000 AP SoCs include two UART controllers that are commonly used as debug ports for embedded systems. They are also often used in providing a terminal connection with a host PC. Because of their utility, they are supported by all layers of software: FSBL, U-Boot, stand-alone, and various operating systems.

The UART is used by the FSBL early in the boot process to send information to the terminal. The simplest connection to the UART controller requires two signals, TX and RX. It is recommended that the user have access to UART1 for debug, even when using a PMOD with two test points, TX and RX.

Because of its low bandwidth, high voltage swings, and large connector, most modern computers no longer have a UART port and instead use USB as the standard peripheral bus. Therefore, it may be necessary to add a USB-to-UART bridge controller on the board to enable connection between the host USB port and the Zynq-7000 AP SoC UART port. This is shown in Figure 3-5. The host PC terminal software recognizes the bridge as a virtual COM port and any of the widely-available terminal programs can be used to communicate with a Zynq-7000 AP SoC.

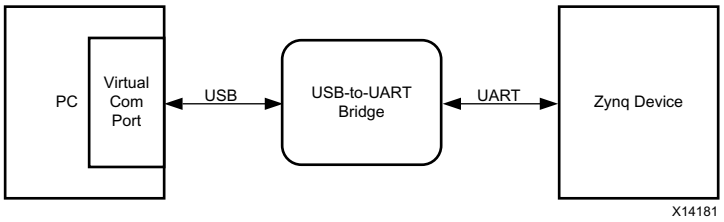

Figure 3-5: **Connecting a Zynq-7000 AP SoC UART to PC via USB-to-UART Bridge**

Figure 3-6 is an example of the Cypress USB-to-UART bridge device being used to enable a connection between a host computer and a Zynq-7000 AP SoC. In the simplest implementation, only the UART controller's TXD and RXD signals are required to support a terminal. If flow control is required, this can be added using the extended MIO. Cypress provides a royalty-free Virtual COM Port (VCP) driver that allows the CY7C64225 USB-to-UART bridge to appear as a COM port to the communication software on the host computer, such as TeraTerm or HyperTerm.

Table 3-2: **CY7C6 Connections**

EPP Pin	UART Function in EPP	Schematic Net Name	CY7C6 Pin	UART Function in CY7C64225
D11 (MIO Bank 1/501)	TX, data out	USB_1_RXD	23	RXD, data in
C15 (MIO Bank 1/501)	RX, data in	USB_1_TXD	4	TXD, data out

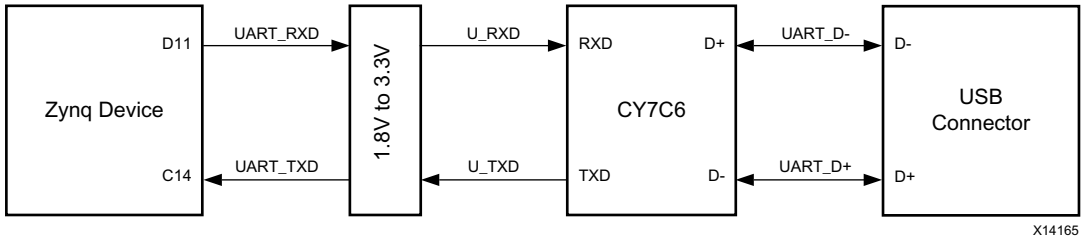

Figure 3-6: **Connection using a Cypress USB-to-UART Bridge**

An alternative implementation uses the Silicon Labs CP2103GM USB-to-UART bridge device. See the *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850) [Ref 8] for details.

Only a two-pin UART can be routed through the MIO. A full 8-pin UART requires routing through EMIO. It supports a programmable baud rate plus protocol and parity. Details of the UART controller can be found in the UART Controller chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

CAN Peripheral

The CAN controller in PS is a memory-mapped peripheral that supports bit rates up to 1 Mb/s. The controller implements a transmit and receive FIFO with a capability of storing 64 messages. The controller implements 16-bit time stamping for receive messages and provides receive and transmit error counters.

Data Flow

The CAN controller has an APB interface connecting the peripheral to the central interconnect and you can use it to configure the control registers. You can route the CAN transmit and receive signals to either MIO or the PL through EMIO. The CAN controller supports five modes of operation:

- Configuration mode
- Normal mode
- Sleep mode
- Loopback mode
- Snoop mode

You can initiate each of these modes with a register write operation using the APB bus attached to the controller. Refer to the CAN Controller chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

I2C Peripheral

I2C (Inter-Integrated Circuit) is a multi-master serial single-ended bus that uses two bidirectional open-drain lines, Serial Data (SDA) and Serial Clock (SCL), both pulled up with resistors. The protocol includes a slave address, an optional register address within the slave device, and per-byte ACK/NACK bits. I2C is used for attaching low-speed peripherals such as sensors, EEPROMs, I/O expanders, programmable clock devices, or A/D and D/A converters to an embedded system. Several bus implementations are derived from the I2C bus, including the System Management Bus (SMBus), Power Management Bus (PMBus), and Intelligent Platform Management Interface (IPMI).

Zynq-7000 AP SoCs include two I2C controllers that can operate at the common I2C bus speeds of 100 Kb/s (standard mode) and 400 Kb/s (fast mode). Each controller can function as a master or a slave in a multi-master design. The master can be programmed to use both normal (7-bit) addressing and extended (10-bit) addressing modes. The master is

responsible for generating the clock and controlling the data transfer. Data can be transmitted or received in both master and slave mode configurations.

Data Flow

In slave monitor mode the I2C interface can be set to monitor slave-busy status. In this mode the I2C continuously attempts a transfer to a particular slave device until that slave device responds with an ACK.

In slave mode, extended address support is determined automatically by detecting a specific code in bits [7:3] of the first address byte. You can set the HOLD bit to prevent the master from continuing with the transfer, preventing an overflow condition in the slave. However, you must clear the HOLD bit before a timeout occurs in the I2C controller. You can program different timeout values.

You can program the I2C controller registers using the APB slave interface. The I2C interface specific SCL and SDA signals can be routed either to the PL via EMIO, or they can be routed to MIO. If routed through EMIO, the SCL and SDA signals are often implemented using tri-stated I/O buffers to communicate with I2C devices connected to one of the PL I/O banks. The controller raises the completion interrupt to indicate to the host that a transfer is complete. In master mode, any NACK received by the controller is communicated to the host using NACK interrupt. Refer to the I2C Controller chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

System Level Considerations

A pull-up resistor shall be placed at the far end of the SCL and SDA lines, furthest from the Zynq-7000 AP SoC. See *Design Calculations for Robust I2C Communications* [Ref 76] for pull-up resistor example calculation. Typical voltages are +5 V or +3.3 V. A level-shifter/repeater might be required depending on the voltages of I2C devices and the Zynq-7000 AP SoC I/O bank used. PCB and package delay skew for SDA to SCL should be less than ±500 ps.

Not all I2C slave devices have programmable addresses, thus address conflicts can occur when connecting multiple devices with identical addresses to a single bus. For this reason, an I2C adapter port is typically routed through an I2C bus switch to connect I2C slaves with identical addresses to different bus segments. Applications must first address and then configure the bus switch to select the desired I2C channel before communicating with slave devices on one of the I2C bus segments. There can be more than one device per I2C bus segment if the I2C addresses are different and the capacitance of a bus segment is below the allowable value. An example I2C bus topology including a 1:8 bus switch is shown in Figure 3-7, page 85. For more information, refer to *PCA9548A Low Voltage 8-Channel I2C Switch With Reset* [Ref 88].

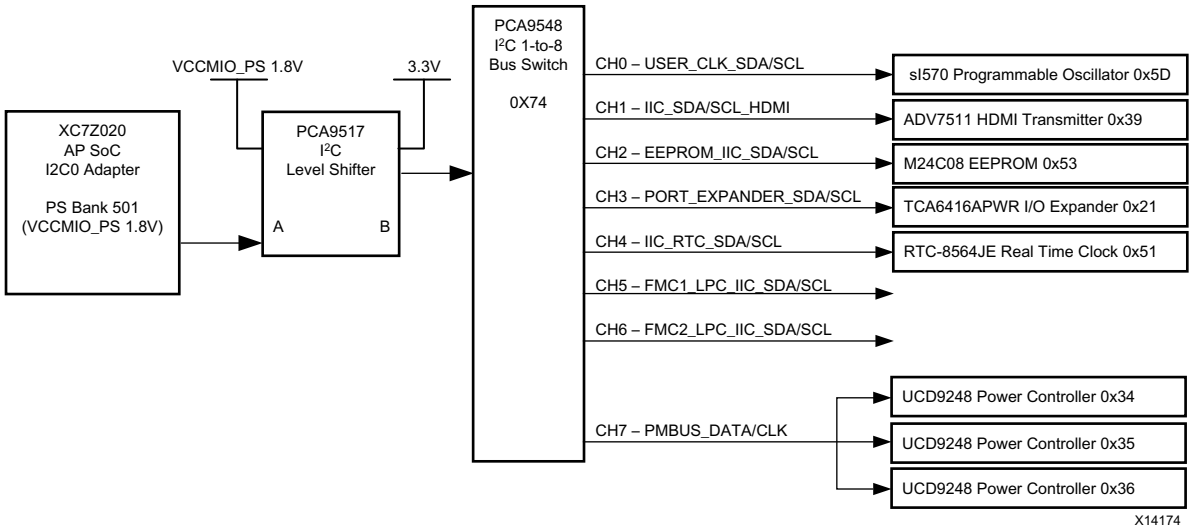

Figure 3-7: **Example I2C Bus Topology**

Stand-alone, U-Boot, and Linux drivers are available to operate the I2C controller. The stand-alone driver can be used inside the FSBL for early initialization of peripherals. All common transfer modes are supported for both master and slave mode. Refer to the Xilinx Linux I2C driver wiki page [Ref 52] for more information on the Linux driver as well as a simple application example.

SPI Peripheral

SPI (Serial Peripheral Interface) is a synchronous serial bus that operates in full-duplex mode. It uses four wires: SCLK (Serial Clock), MISO (Master Input, Slave Output), MOSI (Master Output, Slave Input), and SS (Slave Select). Devices communicate in master/slave mode where the master device initiates the data transfer. The master does not use an addressing mechanism when communicating with a slave. Multiple slave devices are allowed, using individual slave-select lines that are typically active-low. SPI is commonly used to talk to A/D and D/A converters, flash and EEPROM memories, LCDs, and many other peripherals. The JTAG standard is essentially an application stack for a three-wire SPI protocol.

Zynq-7000 AP SoCs include 2 SPI controllers that can operate in master, slave, or multi-master mode. The controller always operates in full-duplex mode, receiving and transmitting data simultaneously.

Data Flow

In master mode, up to three slave devices can be targeted using individual slave-select signals. It is possible to add an external peripheral-select 3-to-8 decoder on the board to target up to eight slave devices. When using the 3-to-8 decoder option you can use software to control the three output pins to the decoder. Software can use this feature to control an external tri-state when a device has a single bidirectional SPI data pin.

In slave mode, the controller receives messages from an external master and transmits a simultaneous reply. The controller reads and writes to slave devices via the 128-byte TX/RX FIFOs using the 32-bit register-mapped data-port registers. The read and write FIFOs provide buffering between the SPI I/O interface and software servicing the controller via APB slave interface. The FIFO is used for both slave and master I/O modes. Refer to the SPI Controller chapter (available at this link) in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

CAUTION! *All SPI transactions must be byte-aligned (multiple of 8 bits). Slave devices that implement non-byte aligned SPI transactions, such as 10-bit address and 16-bit data, are not supported.*

System Level Considerations

Both SPI interfaces, SPI[0,1] can be routed to MIO pins or to the PL via EMIO. If routed through EMIO, the most common method is to implement the interface signals using tri-stated I/O buffers to communicate with SPI devices connected to one of the PL I/O banks (if this is done, the SPI interface cannot be used until the PL has been programmed). The slave select signals SS[1,2] can be optionally enabled when using the core in master mode. These signals are true outputs because they are only available when the core is configured as a master. The SS0 signal is tri-stated because it is also used in slave mode. The SCLK clock frequency can operate up to 50 MHz when the I/O signals are routed to MIO pins. When the I/O signals are routed through EMIO, the SCLK frequency can operate up to 25 MHz.

Figure 3-8 shows the SPI controller configured for master mode and routed through MIO. SS0 must be routed through an EMIO pin. In this configuration, when SS0 is used it needs to be pulled high using a pull-up resistor and must not be driven low by an external device. Up to three slave devices can be connected simultaneously and selected using the corresponding slave-select line (SSn). If less than three slaves are connected, any of the SS signals can be used. If SS0 is not used, it must be pulled to Vcc and not used for any other purposes.

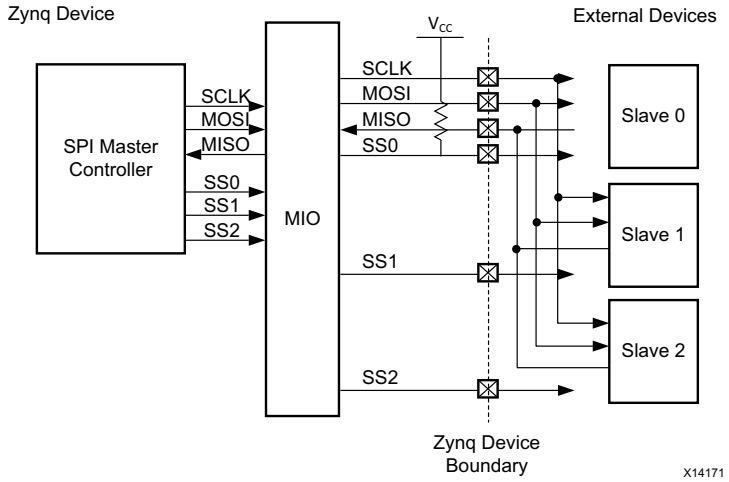

Figure 3-8: **SPI Slave Connection to SPI Controller in Master Mode**

When routed via EMIO, connect the EMIOSPIxSSON0 output signal to the PL slave and tie the EMIOSPIxSSIN input signal to Vcc. It is important that SS0 is high because the controller snoops this signal in master mode to detect a multi-master mode situation. If SS0 is low, the controller will assume multi-master mode and issue a Mode_Fail interrupt. Multi-master mode is not recommended. For details on driving SPI control signals, refer to *Xilinx Answer Record 47511* [Ref 66].

Figure 3-9 shows the SPI controller configured for slave mode and routed through MIO. The slave-select line SS0 is used to connect the external master device to the slave controller. Other external slave devices can be connected off-chip depending on the capabilities of the master device.

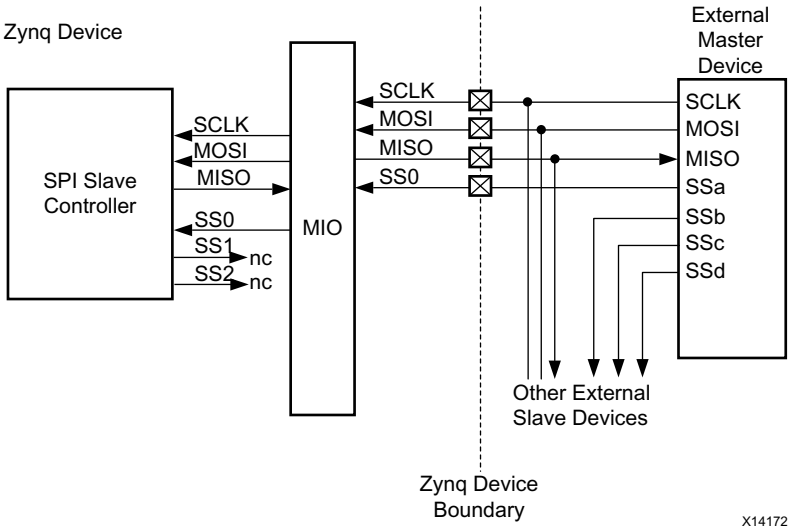

Figure 3-9: **SPI Master Connection to SPI Controller in Slave Mode**

It is recommended that the SCLK, MISO, MOSI, and SS lines have matched lengths to help meet setup and hold times. PCB and package delay skew for the MISO, MOSI, and SS lines relative to SCLK should be less than ±50 ps.

Stand-alone, U-Boot, and Linux drivers are available to operate the SPI controller. The stand-alone driver can be used inside the FSBL for early peripheral initialization if the controller is routed via MIO. If the controller is routed via EMIO, the initialization can take place only after the bitfile is downloaded. All common transfer modes are supported for both master and slave mode. Refer to the Xilinx Linux SPI driver wiki page [Ref 53] for more information on the Linux driver as well as a simple application example.

GPIO Peripheral

GPIO (General Purpose Input/Output) are generic pins that can be programmed by the user at run-time, including whether it is an input or output. GPIO pins are typically used for connecting LEDs, DIP switches and push buttons, or to connect the interrupt and reset signals of board peripherals. Using GPIO via EMIO provides a solution for controlling PL

reset without worrying about the side affects of controlling FCLK_RST. It also provides access to 64 inputs from the PL, 64 true outputs, and 64 output enables to the PL through the EMIO interface. You can use output enables as an additional 64 outputs by controlling the direction registers, if the output enables are not required. Each GPIO is independently and dynamically programmed as input, output, or interrupt-sensing. When configured to sense interrupts, it can be set to level-sensitive (high or low) or edge-sensitive (positive, negative, or both).

System Level Considerations

The GPIO controller has four banks. Bank 0 has 32 pins and bank 1 has 24 pins. The total of 54 GPIO pins in the two banks are dedicated to MIO. These pins are tri-stated and can be configured as inputs or outputs. Banks 2 and 3 have 32 pins each, and the total of 64 GPIO pins are connected to the PL using EMIO. These GPIO contain three signals: input, output, and output enable. Tri-state buffers can be instantiated at the device boundary, however, the PS to PL connections are simple wires.

Each GPIO is independently and dynamically programmed as input, output, or interrupt-sensing. When configured to sense interrupts, it can be set to level-sensitive (high or low) or edge-sensitive (positive, negative, or both). Software can read all GPIO values within a bank using a single load instruction, or write data to one or more GPIOs (within a range of GPIOs) using a single store instruction.

CAUTION! *MIO pins [8:7] are available as output only. GPIO channels 7 and 8 can only be configured as outputs.*

I/O standards and voltages for on-board peripherals need to match the MIO or PL I/O pin configuration.

Both stand-alone and Linux drivers are available to operate the GPIO controller. The stand-alone driver can be used inside the FSBL for early initialization of peripherals. Refer to the Xilinx Linux GPIO driver wiki page [Ref 51] for more information on the Linux driver as well as a simple application example.

Cortex-A9 Multiprocessing Peripherals

The Cortex-A9 multiprocessing peripherals include the SWDT and TTC with auto decrement feature and they can be used as general purpose timers. These timers serve as a mechanism to startup the processor from standby mode.

The System Level Control Register (SLCR) acts as a peripheral to the APU and external PL master. You can access the SLCR only in the secure mode of the master requesting a register access. The SLCR consists of registers that configure various clocking, reset, and security settings of the peripherals.

PS DMA Controller

You can use the PS DMA controller to transfer data from any PS or PL peripheral to DDR, OCM, linear QSPI, SMC, a PL peripheral, or linear addressable memory connected to the M_AXI_GP port in the PS-PL boundary.

You can use the DMA controller in an application requiring hardware co-processor acceleration, where a PL algorithm is used to process data written by the Cortex-A9 into DDR memory. In such an application where data is processed by both a PS and PL peripheral, the DMA controller plays an important role.

You can use the DMA controller to save system power. The CPU can go into low-power mode and bulk data transfers can occur over the PS DMA interface. Because the CPU clock is much faster than the DMA clock, this approach saves considerable dynamic power. Refer to the DMA Controller chapter in *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

XADC

The XADC contains two 12-bit 1-MSPS ADCs with separate track and hold amplifiers, an on-chip analog multiplexer, and on-chip thermal and supply sensors. The two ADCs can be configured to simultaneously sample two external analog-input channels. The track and hold amplifiers support a range of analog input signal types including, unipolar, bipolar, and differential. The external inputs include one pair of dedicated differential analog-input, (Vp/Vn) and sixteen multi-function pins that can support analog input or can be used as digital I/O (Vauxn/p). The dedicated analog inputs can support signal bandwidths of at least 500 KHz at sample rates of 1MSPS, or 250KHz on the auxiliary channels.

The XADC also includes a number of on-chip sensors that support measurement of the on-chip power supply voltages and die temperature. The results of the ADC conversion from any source (temperature, supply voltage, or analog input channels) are stored in a set of status registers inside the XADC block. Because the temperature is available and the clocks are programmable, you could adjust the Zynq-7000 AP SoC clocks as temperature changes.

The XADC block also includes a set of control registers used to configure and control XADC operations. This includes alarm registers that are used to specify automatic alarm thresholds for the internally-measured sensors (temperature and voltage). The alarm automatically triggers a PS interrupt when temperature or voltages falls outside the acceptable ranges defined by the registers.

The XADC analog inputs can be used to support motor control, a touch sensor, or many other applications that require an analog front-end. The voltage and temperature sensing capability can be used to monitor system health and to react appropriately when the system experiences abnormal temperature or voltage conditions.

Two application notes describing the XADC used as a system monitor and to perform ADC operations on external analog signals are:

- *Using the Zynq-7000 Processing System (PS) to Xilinx Analog to Digital Converter (XADC) Dedicated Interface to Implement System Monitoring and External Channel Measurements* (XAPP1172) [Ref 39]
- *System Monitoring Using the Zynq-7000 AP SoC Processing System with the XADC AXI Interface* (XAPP1182) [Ref 41]

The XADC status and control registers are accessible using either the parallel DRP interface or the serial JTAG-DRP interface, as shown in Figure 3-10, page 91. The LogiCORE™ IP can be used to encapsulate the entire XADC hard block with the necessary logic to turn the XADC into a 32-bit AXI slave peripheral that can be connected to the master AXI_GP port of the PS. This solution is required for the PS to access the data at a 1Mbps rate. The AXI XADC LogiCORE IP has the disadvantage that it consumes PL resources to implement, however, it provides a fast and clean interface to the XADC. It takes full advantage of the parallel data path and is the only method that can realize the full 1MSPS, as demonstrated in *System Monitoring Using the Zynq-7000 AP SoC Processing System with the XADC AXI Interface* (XAPP1182) [Ref 41].

An alternative solution is to use the built-in PS_XADC interface block. The advantage is that it requires no additional PL logic (the FPGA does not have to be configured when using built-in features). Reading and writing the XADC DRP registers requires a write of the proper command to XADCIF_CMDFIFO and a read from XADCIF_RDFIFO. However, because the block serializes the FIFO contents and shifts data in and out of the XADC hard block one bit a time via the DRP JTAG interface, this solution is slower and cannot keep up with the higher data rate. As demonstrated in *Using the Zynq-7000 Processing System (PS) to Xilinx Analog to Digital Converter (XADC) Dedicated Interface to Implement System Monitoring and External Channel Measurements* (XAPP1172) [Ref 39], this interface was able to achieve only 100KHz. Also, the PL-JTAG interface and the internal PS-XADC interface cannot be used at the same time.

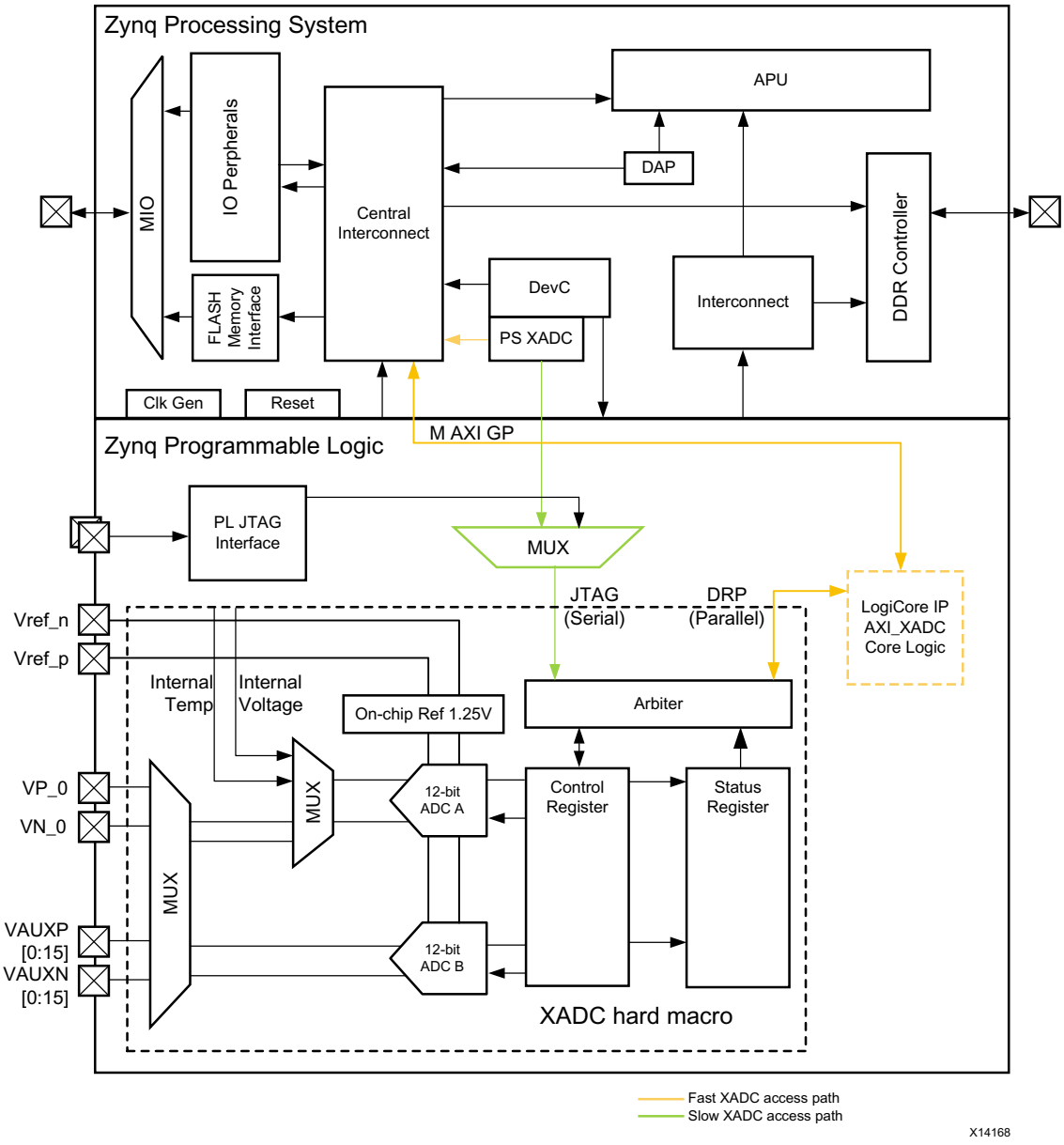

Figure 3-10: **XADC Block Diagram**

Communication with Peripherals

The peripherals discussed above implement an AMBA-specific bus structure to communicate with the APU and the memory system. USB, GEM, SDIO, and SPI peripherals have two types of communication interface:

- The APB interface connects the peripheral slave to the APU master. The interface communicates using the central interconnect.
- The AHB interface enables high speed bus transactions between the DMA controller embedded in the peripheral and the system memory. The interface connects to the system memory using the central interconnect.

Figure 3-11 shows the peripheral bus connection to main memory and the APU. Figure 3-11 is a generic block diagram of a peripheral with DMA and control register sets implemented.

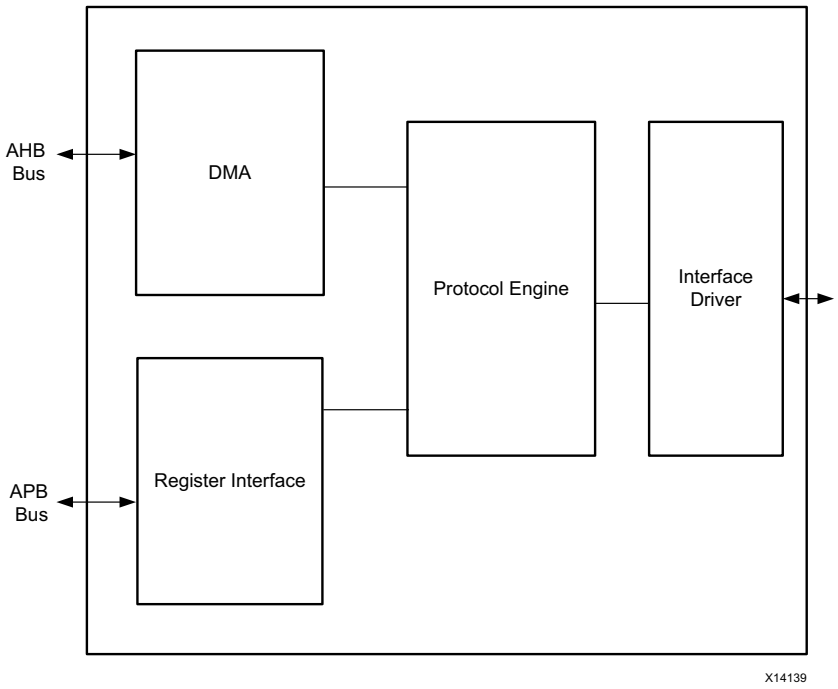

Figure 3-11: **Peripheral Communication Bus**

Peripheral Design Example

This example shows things to consider when designing with peripherals. The example uses the Zynq-7000 AP SoC GEM controllers and two different physical interfaces. One controller has an RGMII PHY using the MIO interface and the other controller has a 1000BASE-X PHY using the EMIO interface. You can select any supported interface to meet your requirements.

1. The PS has two instances of a GEM controller. One uses the RGMII interface and the other uses the 1000BASE-X interface.
2. TrustZone security aware access of the peripheral from the Cortex-A9 multiprocessing core.
3. You can enable and disable the TCP checksum off-load to GEM hardware, noting the CPU utilization and performance differences between the two options.

Figure 3-12 shows the hardware block diagram.

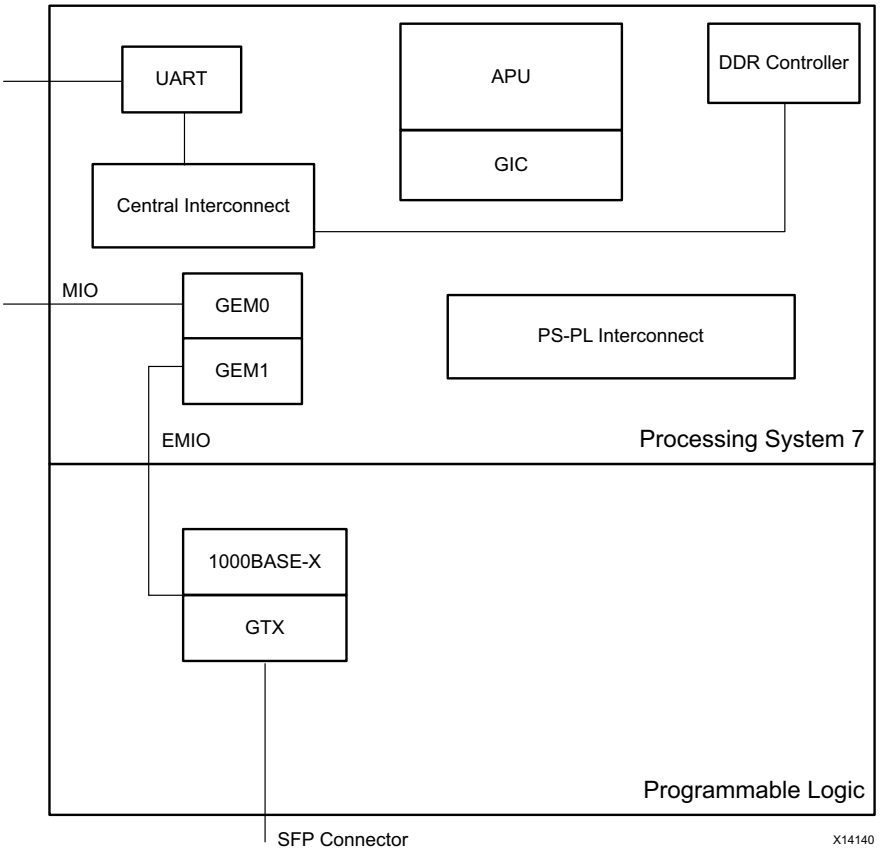

Figure 3-12: Example GEM Instantiation

Although this example is built around a gigabit Ethernet peripheral, you can extend the concepts to other PS peripherals.

Hardware Design Considerations

You can create the hardware design using the Vivado® IP integrator flow. You must enable MIO connectivity for GEM0 and EMIO connectivity for GEM1. Enabling EMIO for GEM1 brings the transmit and receive GMII signals to the PS IP top-level instance. The user must instantiate the 1000BASE-X IP core from Xilinx in the IP integrator design and connect it to the GEM1 EMIO ports. You can implement the design using the Vivado Design Suite implementation flow to generate the bitstream. See *PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 AP SoC* (XAPP1082) [Ref 38] for more information.

Software Design Considerations

The GEM peripheral supports off-loading the TCP checksum to hardware to boost Ethernet performance and improve CPU utilization by performing compute-intensive checksum calculations in hardware. The user can enable checksum off-load by writing into the GEM configuration register using the APB interface.

The GEM driver configures the GEM DMA for a specific transfer size. The driver also sets up the descriptor ring, DMA allocation, and recycling, and programs the source and destination MAC addresses.

The DMA transfer begins after enabling the transmitter and receiver by writing to the network control register.

Designing IP Blocks

Pre-verified intellectual property (IP) blocks are used to reduce time-to-market, and to enable feature-rich digital and analog circuitry on the SoC. These IP blocks include embedded processors, memory blocks, interface blocks, analog blocks, and components that handle application-specific processing functions. Use of standard interface specifications, such as the AMBA high-speed specification and AXI specification, improve IP block reusability. You can refer to the Vivado IP catalog for more information on the IP blocks supported by the selected Zynq-7000 AP SoC device.

There are two main categories of IP blocks:

- **Soft IP Blocks**: You can implement these blocks in an FPGA fabric and are specified using RTL or higher-level descriptions. They are more suitable for digital cores, because a hardware description language (HDL) is process-independent and can be synthesized to the gate level. The HDL form has the advantage of flexibility, portability, and reusability, with the drawback of not having guaranteed timing or power characteristics.

- **PS IP Blocks**: These blocks have fixed layouts and are optimized for a specific application and process. Their primary advantage is predictable performance, which comes with additional effort and cost, plus lack of portability, that may greatly limit the areas of application. PS IP blocks are usually prequalified, meaning the provider has tested it in silicon. This adds greater assurance to its correctness.

The inherent complexity of SoC designs leads to the implementation of more complex interfaces around a processing block. The high-speed serial transceiver present on a Zynq-7000 AP SoC is a typical example. The transceiver's high-speed serial IOs are capable of transferring data in multiples of gigahertz. The transceiver block is capable of performing byte-boundary alignment, clock data recovery, and skew elimination across multiple transceiver lanes. It is also capable of clock compensation in systems that use independent transmit and receive clocks with ppm variation. User logic connects to the transceiver through a complex interface, and the functionality of that logic is complex because it must control the various functional blocks within the transceiver.

An IP block plays a significant role in handling the transceiver configuration by masking the underlying complexity from the user interface. An Aurora IP block from Xilinx with an AXI4 Stream-compliant user interface performs the transceiver initialization, clock compensation, channel bonding, and byte-boundary alignment. A user can build a transceiver application based on the AXI4 Stream interface without knowledge of the underlying transceiver complexity.

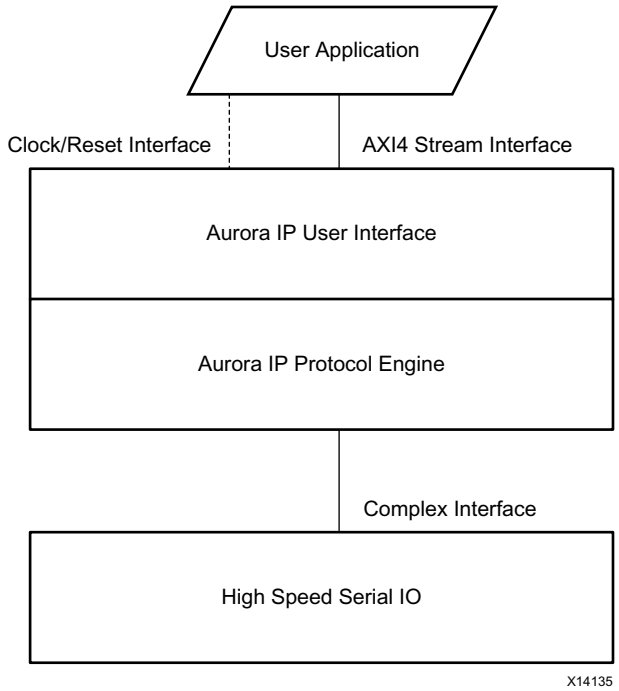

Figure 3-13: Aurora IP Example

Figure 3-13 shows how the Aurora IP plugs into the transceiver interface in a Zynq-7000 AP SoC.

IP Core Design Methodology

Xilinx enables different methodologies for creating IP blocks and offers an easy maintenance approach. You can reuse the IP blocks in the embedded system designs created for Zynq-7000 AP SoCs and other processing systems, such as MicroBlaze™ processors. The following sections describe the IP creation methodologies using the Xilinx tool chain.

System Generator

System Generator is a tool that can create DSP IP blocks based on a user configuration, using MathWorks Simulink for FPGA system design. It offers a high-level model-based environment for system designs. A large number of building blocks are available in System Generator, from simple mathematical operators to complex DSP operations.

In the Vivado Design Suite, the IP Packager compilation target allows a user to package the IP generated from the System Generator and include it in the Vivado IP Catalog. You can use the System Generator design like other IP blocks from the IP Catalog and instantiate it in your design.

HDL Coder

HDL Coder is a tool from MathWorks that generates synthesizable HDL code from MATLAB functions and Simulink models. It provides a workflow that analyzes the MATLAB/Simulink model and converts the model from floating-point to fixed-point, providing a high-level abstraction. The workflow also provides verification code, allowing the HDL code to be tested with the original MATLAB/Simulink model.

Not all the functions present in MATLAB and Simulink support HDL generation. Because of this, some of a model's functionality may have to be modified to use functions or blocks that support HDL generation.

Vivado High Level Synthesis

Vivado High Level Synthesis (HLS) is a tool from Xilinx that is capable of converting C-based designs to RTL design files for implementation on Xilinx FPGA devices. The Vivado HLS tool flow is shown in Figure 3-14.

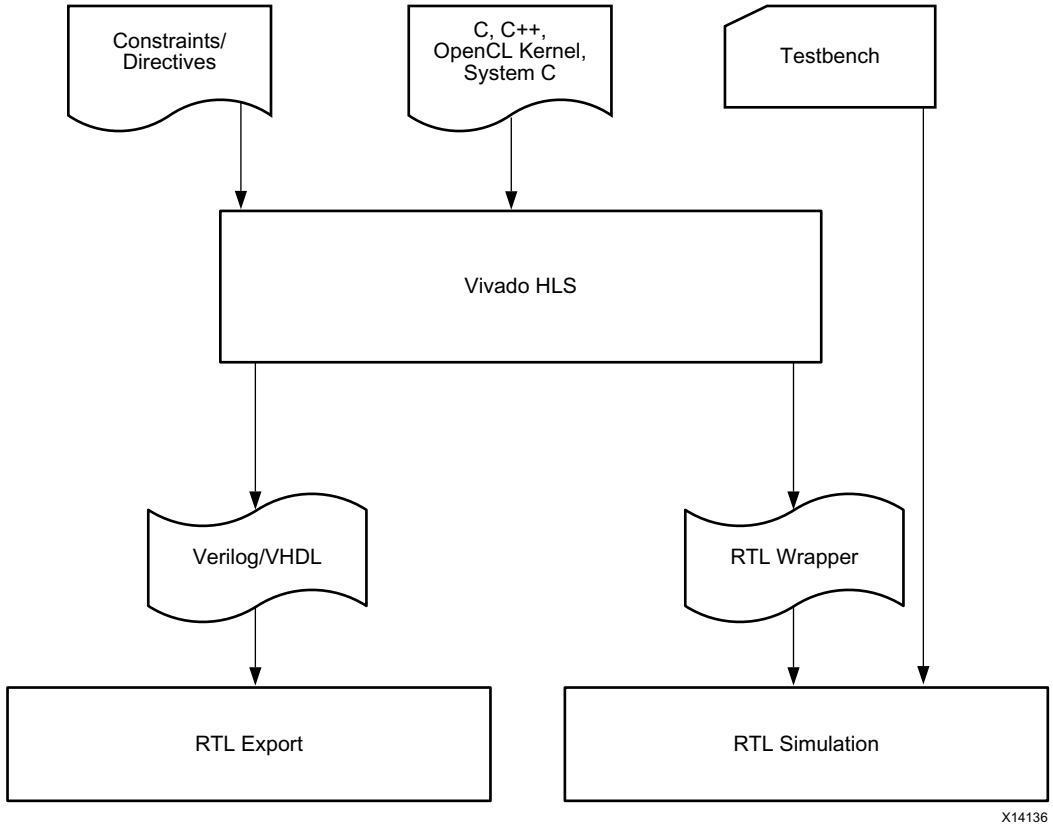

Figure 3-14: Vivado HLS Flow

The Vivado HLS flow packages the RTL output into the following formats, allowing you to easily integrate the output into other Xilinx design tools:

- IP Catalog: Contains a ZIP file that you can add to the Vivado IP Catalog.
- Synthesized Checkpoint (.dcp): A Vivado checkpoint file that you can add directly into a design in the Vivado Design Suite.
- System Generator for DSP: You can add this output to the Vivado edition of System Generator for DSP.
- System Generator for DSP (ISE®): You can add this output to the ISE edition of System Generator for DSP.
- Pcore for EDK: You can add this output to the Xilinx Platform Studio.

In addition to the packaged output formats, the RTL files are available as stand-alone files (not part of the recommended packaged formats).

IP Core Design Considerations

When designing an IP block, reusability of that IP block across multiple embedded system designs and across different user configuration options should be considered. Parameterization and implementation of standard bus topologies allows an IP block to be reusable in most system-level scenarios.

You should consider the following when designing an IP block:

- **Parameterization**: Parameterization includes interface-level configuration and IP feature-specific configuration. Interface-level configuration is important when the IP block interfaces with other IP blocks and the user must control different bus parameters, including the data and address bus width, clocking, reset interface configuration, etc. IP feature-specific configuration enables or disables a feature in the IP block depending on the user selection.

  You can handle parameterization in IP blocks in multiple ways. The basic level of parameterization can be implemented in the HDL code. The HDL parameters can be configured from the IP configuration wizard available after packaging the IP block using Vivado IP integrator tool. You can implement a customized environment to parameterize the IP block.

- **Bus Topology**: The reusability of IP blocks is an important consideration when designing an IP block. You can enhance reusability by implementing a bus topology that complies with industry-standard bus interfaces. One of the most widely used standards is AMBA, and a broad class of IP blocks exist that comply with that bus specification. A bus topology implemented with industry-standard interfaces enables easy plug and play of master and slave IP blocks at the system level. Such IP blocks can also offer flexibility in selecting a particular version of the standard protocol and interface-level configurations.

  The choice of bus interface depends on system-level requirements. The AXI4-Lite bus is primarily used as a control bus from the PS, and the AXI4 Memory-Mapped bus is for memory transfers between the IP core. The AXI4-Stream bus is used to connect to streaming blocks in the design that do not have addressing context. The following figure shows the AXI4-Lite, AXI4 Memory-Mapped, and AXI4-Stream buses in an AXI DMA-based design. In this figure, the AXI DMA is responsible for moving data between the PS system memory and the AXI4-Stream compliant Sobel IP using the high performance port. The PS sets up the AXI DMA's buffer descriptor using the AXI4-Lite interface.

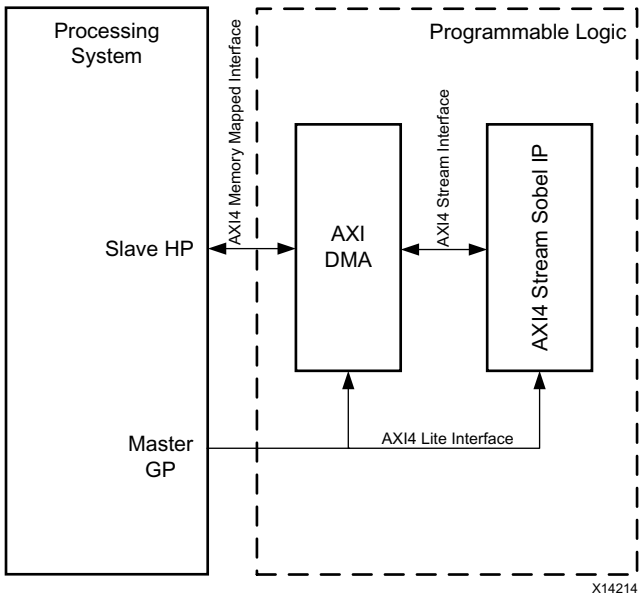

Figure 3-15: **AXI4-Lite, AXI4 Memory-Mapped, and AXI4-Stream Buses**

- **IP Security and Documentation**: Protecting a design against copyright infringement is an important consideration in designing IP blocks. You can employ various security algorithms to encrypt the HDL file. The Xilinx tools allow you to encrypt and decrypt the HDL using the AES algorithm.

Connecting IP Blocks to the PS

When the interface between the PS and PL uses AMBA-compliant interfaces, you can implement a design using soft IP blocks. The PS-PL interface has a GP port (master and slave) that can implement an AXI3 or AXI4 interface, and can connect the PS to a soft IP core register interface. You can use the high-performance (HP) ports as AXI3 slave ports that are driven by an AXI3 or AXI4 master IP block in the PL. You can use the ACP interface in a similar way. The ACP interface implements the AWCACHE and ARCACHE signals, and the user IP block implementing the ACP master controller needs to implement these signals. For more information, refer to ACP and Cache Coherency, page 116.

The ACP interface enables connectivity to the L2 cache, and the interface can be used where the IP block in the PL is co-processing a compute-intensive operation requiring considerable CPU utilization.

You can implement the AXI3/4 memory mapped or AXI4 stream protocol. The AXI3/4 memory mapped interface has read/write addressing signals and data interface signals. The AXI4 stream interface does not have address signals.

When designing a Zynq AP SoC with an IP block, the PL must account for the following:

1. The IP block complies with the AXI3/4 protocol specification. The AXI interfaces in the PS-PL boundary are AXI3-protocol compliant. You can use a Xilinx-provided AXI interconnect IP block to connect the AXI4-compliant IP to the PS-PL interface ports.
2. The IP block is part of an IP integrator repository, simplifying integration of the IP block into a system level design.
3. The PS-PL interface supports memory-mapped access of implemented peripherals.
4. You can select the AXI4 stream interface to connect IP blocks that do not require addressing context. The AXI4 stream interface is widely used for video-based IP blocks. You can store the stream video input from the video IP block in the PS DDR memory using the video DMA IP block connected to the HP port interface.

Case Study: Designing with Xilinx IP Blocks

This section describes how to design a system using IP blocks from the Xilinx IP integrator catalog. You can apply the methods to any IP block.

The case study is of a notch filter, which is a DSP IP that can process analog samples to remove non-linearity introduced by an external sensor. When an analog signal passes through a non-linear sensor, for example an RTD sensor, the sensor can introduce nonlinear behavior at the analog signal output. This is due to variations in the sensor transfer function based on temperature. The non-linear output requires compensation, and you can implement the compensation algorithm in either the analog or digital domains. Digital algorithms offer more cost effective and accurate reconstruction of the non-linear signal.

A Zynq-7000 AP SoC has a Xilinx ADC in the PL, and the IP block that instantiates the XADC provides a user wrapper with an option to convert the XADC samples to an AXI4-Stream compliant interface. The user can design an IP block that compensates for the non-linear sensor by preemptively correcting the XADC's samples with lookup polynomial coefficients.

You can design the sensor-compensating IP block using DSP48 blocks available in the Zynq-7000 AP SoC. The IP can implement two user interfaces: an AXI4-Stream interface for data and an AXI4-Lite interface for control. The AXI4-Stream interface can connect to the XADC Wizard IP, and the AXI4-Lite interface can connect to the PS's GP0 master-interface for controlling the IP-specific parameters.

Figure 3-16, page 101 shows a block diagram of the IP block.

The process of creating the IP block is:

1. Write HDL code that implements an AXI4-Stream interface for data and an AXI4-Lite interface for control. Implement the function that multiplies the AXI4-Stream data with the interpolated coefficients.
2. Package the HDL code using Vivado IP Packager. Auto-infer interfaces and associate clocks with each of the interfaces.
3. Set up a local repository and copy the packed IP XACT files to the local repository.
4. Create a Vivado project and add the local repository.
5. Create the block design by instantiating the local IP block with other IP blocks and connect them together.
6. Implement the design using Vivado Design Suite.

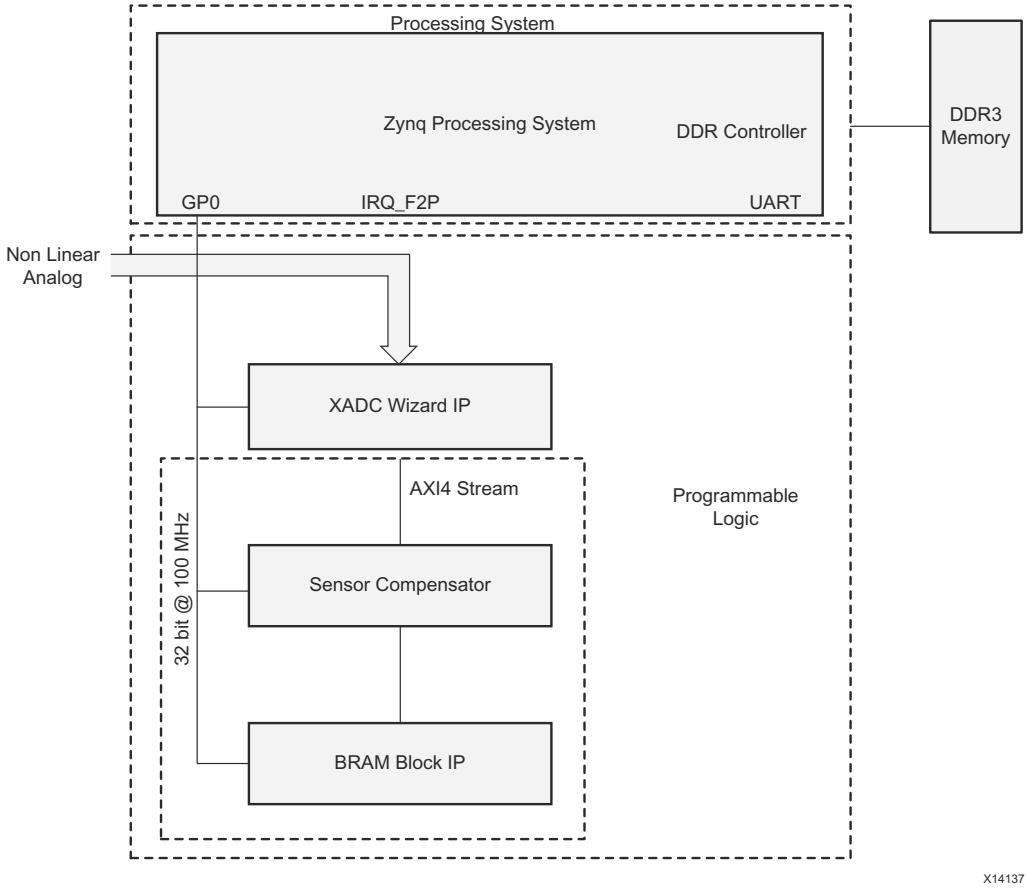

Figure 3-16: IP Use-Case Illustration

Hardware Performance Considerations

The PL on a Zynq-7000 AP SoC has multiple communication paths for software and hardware to share data and resources. Each of these paths has configuration settings for altering hardware-specific performance. In this section, hardware performance metrics will be defined, and methods for tuning performance of AXI masters, AXI slaves, and AXI datapaths will be described.

Hardware Performance Metrics

An important metric of PL performance is the AXI throughput and latency when accessing performance-critical IP in a Zynq-7000 AP SoC. Xilinx uses ARM AMBA AXI interconnects in PL to connect and map user IP into a global address space. Some application datapaths will be throughput constrained (such as video and networking) or latency constrained (such as real-time applications or protocol-specific constraints). Often when a Zynq-7000 AP SoC design is started, the system architect can only specify performance using the two simple metrics of throughput and latency, something observed by many Zynq-7000 AP SoC customers.

To highlight the trade-off between latency and throughput, an ACP master that issues eight outstanding 32-byte transactions will have an average read latency 130% higher than a single outstanding transaction master. However, the master with eight outstanding transactions also has a 262% higher average throughput. This is because multiple outstanding transactions are pipelined, leading to a higher throughput and latency performance profile. The designer can decide if latency or throughput is more important to system performance.

These two metrics form the foundation of visualizing and tuning application performance. The AXI interfaces that IP might access within a Zynq-7000 AP SoC are described below. AXI IP design best-practices are not considered.

High-Performance AXI Masters

An AXI master's design and configuration affects its ability to drive high-performance traffic. Following are two examples of AXI masters that have different performance characteristics as seen from an AXI interface.

AXI Master #1. Supports a 32-bit AXI-Lite interface and issues one outstanding AXI transaction at a time. This master generates single-beat transactions and runs at 100 MHz on a ZC702 board.

AXI Master #2. This master uses a 64-bit AXI4 interface and can issue 16 outstanding transactions at a time, with a maximum burst length of 16 beats per transaction. This master can run at 200 MHz on the same Zynq-7000 AP SoC board.

Independent of the datapath and the AXI destination slave, the performance of AXI master #1 will not match AXI Master #2. To minimize throughput constraints, the AXI master should fully use the AXI address and data channels. To minimize latency constraints, limit the number of outstanding transactions because transactions will pipeline sequentially in the AXI data channel.

This simple example might be intuitive, but building high-performance AXI masters adds design complexity and may consume more resources. Often, performance is only considered after performance goals are not met. The following section outlines performance considerations when designing an AXI master.

Building High-Performance AXI Masters

Many designs are generated using prebuilt IP, and performance choices are limited. However, the following general guidelines can help build high-performance AXI masters, or tune existing IP when parameterization is possible.

- **For throughput-constrained masters, support multiple outstanding transactions**. This requirement adds complexity to AXI masters, but Zynq-7000 AP SoC interfaces can internally support varying numbers of multiple transactions. This data pipelining enables higher utilization of AXI data channels, often the only channel that matters in throughput-based designs.
- **For low latency masters, throttle the number of outstanding transactions to maximize latency-throughput behavior**. This item can be difficult to quantify, but through experimentation, the latency curve can be shaped to minimize latency the master sees while allowing multiple outstanding transactions. Figure 3-17 is an example latency-throughput curve showing how an increase in the number of requested transactions affects latency.

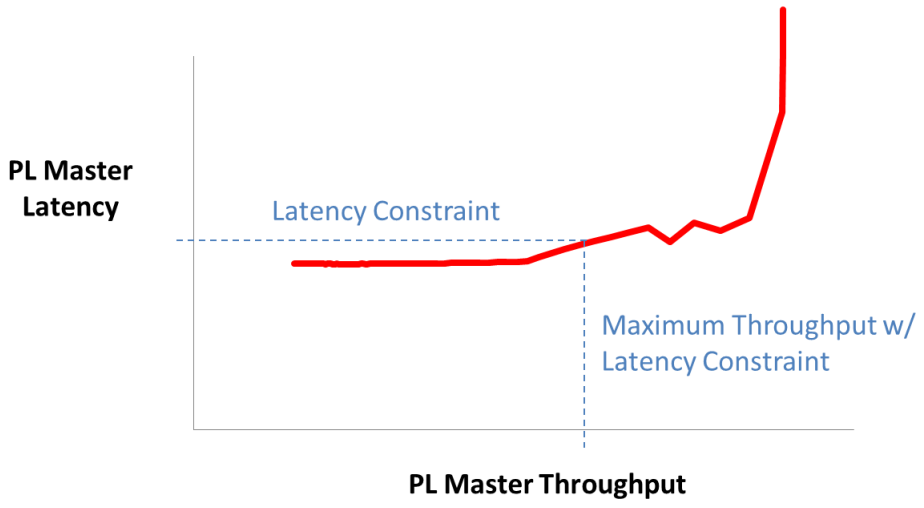

Figure 3-17: Example Latency Throughput Curve for HP Port Reads

- **Ignore the AXI response channel**. If possible, use posted writes to reduce the transaction latency. For example, consider a transaction complete on an AXI rdlast or wrlast signal, ignoring the write response channel response. In a well-formed design, there will be no channel or slave errors that an AXI master must manage. If writes cannot be posted, then this channel cannot be ignored. However, the AXI write response channel typically adds tens of cycles to a transaction in many Zynq-7000 AP SoC datapaths.
- **Consider AXI master addressing behavior and the destination AXI slave response**. The address sequence that an AXI master generates affects the destination slave's ability to service that traffic. For example, if an AXI master generates random addresses to a DDR-based memory, it can cause lower memory performance. Also, multiple masters accessing DDR address ranges can cause suboptimal DDR scheduling. This is not always an issue because some memory slaves, such as the OCM, are better at handling random accesses. Also, if random access is kept within a small address range, the L2 cache can service this traffic. These AXI slaves will be discussed later.
- **Maximize clock rate, data width, and transaction burst size**. These parameters should be matched to what is supported natively by a Zynq-7000 AP SoC (such as internal 64-bit datapaths and a maximum AXI3 burst length of 16 beats). Typically, increasing the AXI master clock rate affects just a portion of the total datapath, because a transaction likely also travels through the PS and its clock domains. PL data widths will be internally translated to the PS data widths, when needed. Using burst size as an example, if a PL AXI master drives AXI4 traffic into the PS, the transactions might be translated into AXI3 transactions.

Using Performance-Critical AXI Slaves

Performance-critical slaves respond to data-movement requests from PL. Zynq-7000 AP SoCs have three AXI slaves that are especially performance critical: the OCM, L2 cache, and DDRC. The addition of a configurable memory-interface generator (MIG) to the PL is also possible. The benefits of the different slaves are:

- **On-Chip Memory.** The OCM is a 256KB memory block accessible to both ARM cores and the PL from the GP, HP, and ACP ports. The OCM is an ideal component for use in synchronization or scratchpad applications.
- **L2 Cache.** The 512KB L2 cache is only accessible by the ACP from the PL. It is a shared resource with the two ARM Cortex-A9 cores. This slave has excellent performance characteristics when data traffic fits within 512KB. However, performance degrades if there is heavy processor contention. The ACP and coherent memory access is described in ACP and Cache Coherency, page 116.

- **DDR Controller.** You can connect the DDRC to a variety of off-chip memory devices, such as DDR3 and LPDDR2. The DDRC is accessible from the PL using the AXI master interfaces (GP, HP, and ACP). Each PL interface has a unique datapath to the DDRC. The DDRC has higher memory capacity and throughput performance than the OCM and L2 cache.
- **PL Attached MIG (Optional).** Zynq-7000 AP SoCs can support an MIG-generated memory controller in the PL. This soft IP option provides another high-performance memory interface that can isolate PL memory from the processor system's software access to the DDRC. The MIG exports an AXI interface that allows designs to attach standard AXI-master IP. The greatest benefit of this core is the ability to customize the PHY into banking and addressing schemes for a specific application.

Selecting Performance-Critical AXI Slaves

The AXI slave selected for a datapath is often done so to meet functional requirements, such as CPU coherency or memory size. The selection of an AXI slave can also be driven by performance requirements. The following selection considerations have been used by several Zynq-7000 AP SoC customers:

- **Avoid or isolate performance-critical datapaths.** Often during the design process, certain datapaths are marked as critical to system performance and other paths are marked as secondary. For example, a network-processing application may need dedicated access to off-chip memory, whereas packet inspection can occur at a slower rate on less critical datapaths. You can isolate ACP access to the L2 cache from HP to DDRC traffic by limiting the memory footprint of the ACP access. Similarly, OCM accesses from GP can be isolated from ACP and HP traffic.
- **Low latency.** The amount of latency that tolerated depends on the system-level application and behavior of data traffic. The OCM has excellent low-latency characteristics. Accessing small memory regions through the ACP enables an application to take advantage of low-latency L2 cache access.
- **High throughput.** The high-performance ports provide high throughput to the PS DDRC from the PL. A common problem with the HP performance is L2 cache contention at the DDRC. A memory-intensive software application can cause a reduction in HP throughput. However, you can reduce contention by using the DDRC priority settings and the QoS 301 settings on the L2 cache to DDRC datapath.
- **Configurability and confidence.** Some designs might use a PL MIG to avoid memory access contention with the processor subsystem. Other designs might use a custom memory interface based on previous designs or developer experience. Prototyping boards for Zynq-7000 AP SoC support MIG (such as the ZC706 board) and enable designers to evaluate a custom MIG core and compare the results with standard Zynq-7000 AP SoC datapaths to the DDRC.

High-Performance Datapaths

There are two symmetric datapaths for the PS to communicate with the PL (master GP0, master GP1), and there are seven datapaths for PL AXI Masters to drive traffic into the PS.

The AXI interfaces are described in the following sections:

- ACP and Cache Coherency, page 116
- PL High-Performance Port Access, page 120
- GPs and Direct PL Access from APU, page 133

The process of using the system-performance goals to select between those interfaces is described below.

Selecting a High-Performance Datapath

The selection of a datapath is often based on design function. However, performance considerations can also drive datapath selection.

- **Choosing between the ACP and HP ports.** When the cache and user AXI4 signals are set for a cache access, the ACP port provides access to the L2 cache and maintains cache coherency with the ARM Cortex-A9 cores. Using the ACP without the cache enabled can be useful to a PL master because the memory map matches that seen by the Cortex-A9 cores.The L2 cache is a shared resource, and using the ACP port can impact processor memory bandwidth. Also, if the traffic generated by AXI masters spans a large address range (such as video frames) or is random in nature (such as scatter-gather DMA), L2 cache misses will increase, causing a corresponding increase in latency and potentially lower throughput.

  The HP ports support 32-bit and 64-bit data paths, providing high-throughput access to the DDRC. The HP ports contain FIFOs for increased burst throughput and you can control their prioritization. The HP ports also provide low latency and high bandwidth to the OCM. You might find this useful if the accelerators are in the PL and the OCM is large enough for the buffer.

- **Selecting the GP ports.** The slave GP ports provide PL masters direct access to many resources isolated from performance-critical datapaths on the ACP and HP ports.
- **Memory-access footprint.** If a design uses the ACP for non-coherent access with poor cache locality, then the DDRC serves the majority of ACP traffic. In an example Zynq-7000 AP SoC deployment, ACP latency is reduced up to 53% by increasing cache locality of transactions. HP and GP transactions do not access the L2 cache and are not directly affected by cache behavior. For more information on using the ACP for coherent access, see ACP and Cache Coherency, page 116.

- **PS datapath performance controls.** System clock rates, QoS-301 signals, and DDRC priority settings can all impact datapath performance. You can set most of the memory-mapped registers and functions that use IP integrator when building the design. You can set QoS-301 bits at runtime. Although higher system clock rates help move data faster, the QoS-301 signals and DDRC priority settings can shift performance between contending AXI masters. For example, QoS-301 settings on the L2 to DDRC datapath can limit L2 cache transactions to the DDRC, allowing HP traffic to be serviced by the DDRC more frequently. DDRC priority settings can also be used to prioritize read and write transactions within the DDRC.

Monitoring Hardware Performance

Xilinx provides an AXI performance monitor (APM) core shown in Figure 3-18 that you can use to observe and measure latency and throughput on PL AXI interfaces. This core is available in IP integrator and is the basis for all AXI performance measurements referenced in this document.

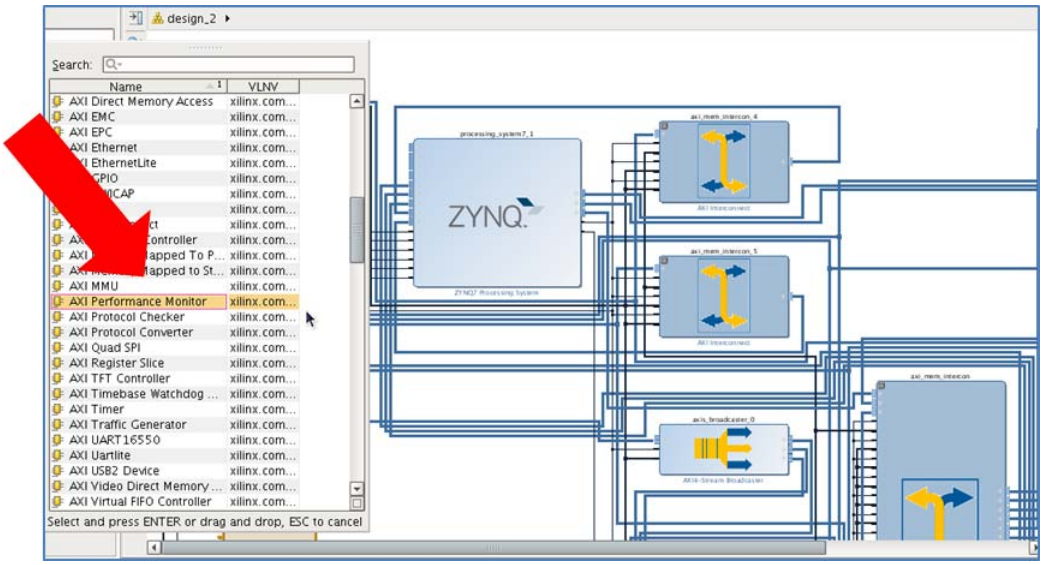

Figure 3-18: Adding an AXI Performance Monitor into a Vivado 2013.4 IP integrator Design

Zynq-7000 AP SoC developers can insert this APM into their PL design like other IP, and verify performance metrics throughout the design cycle.

Dataflow

This section provides an overview of the dataflow in a Zynq-7000 AP SoC. The first part describes the dataflow within the PS, focusing on APU access of PS DDR, and of PS peripherals to PS DDR. The second part describes dataflow from the PL to the PS, focusing on the following PS-PL AXI interfaces: general purpose interfaces, ACP interface, and high performance interfaces. Throughout, major system design characteristics and considerations are described. For more information, refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

The various interconnect blocks used within the Zynq-7000 AP SoC devices are designed to meet the needs of various functional blocks. The PS interconnect is based on high-performance data path switches that enable data transfers between various peripherals and DDR memory, or between the APU and DDR or OCM memories. The PS-PL interface provides an AXI interface in the PL, enabling PL peripherals to connect to the PS and subsequent dataflow.

The block diagram in Figure 3-19 shows the major data paths that are covered in this section.

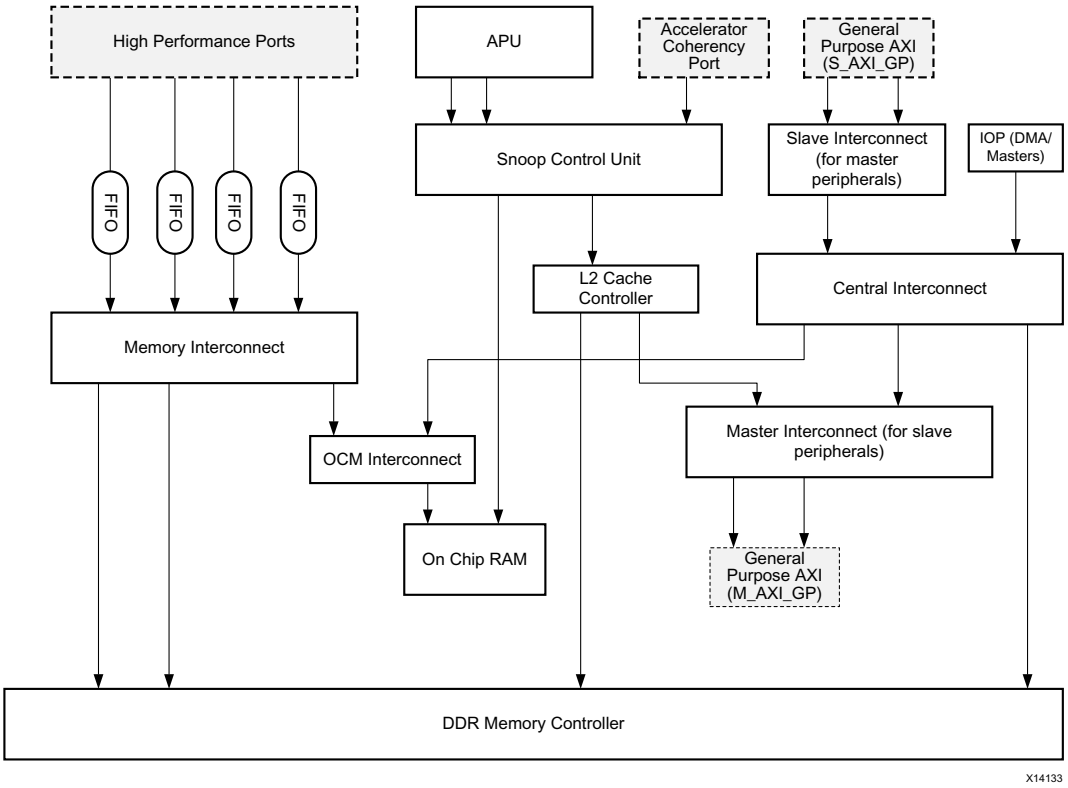

Figure 3-19: Block Level Data Path Overview

Interconnect within PS

There are two primary data paths within the PS: the APU interconnect to PS DDR, and the PS peripheral interconnect to PS DDR.

The central interconnect is 64 bits and is the core of the ARM NIC-301 switches. The master and slave interconnect route low-to-medium level traffic between the central interconnect and the general purpose interconnects and IO peripherals in the PS. Memory interconnect provides a direct, high-speed data path from the PL HP interface to the DDR memory controller. It also provides access to on-chip RAM through the OCM interconnect.

The interconnect does not offer a full cross-bar structure; not all masters can access all slaves. The details on which masters can access which slaves are available in the Interconnect Datapaths table at this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

APU Access to PS-DDR

Each processor provides two 64-bit pseudo-AXI interfaces. All AXI transactions are routed through the SCU to the OCM or L2-cache controller based on their addresses. The APU accesses DDR memory through the SCU and L2-cache controller. Both the SCU and L2-cache controller behave like switches because of their address-filtering feature. The SCU also provides the lowest latency path from the APU or ACP peripherals in the PL to the OCM, and also contains the intelligence to manage data coherency between the two processors and L1cache.

When the L2-cache controller is disabled, the transactions are directed either to the DDR controller or master interconnect based on their addresses.

PS Peripheral Access to PS-DDR

The IO peripherals (USB, GEM, and SDIO, for example) implement two interfaces for communication:

- An APB interface for connecting peripheral slaves to the APU master
- An AHB interface that facilitates high-speed bus transactions for connecting an embedded DMA controller in the high speed peripherals to system memory

The APB interface is used primarily to control peripherals. The AHB interface (from a DMA master in a peripheral) is connected to central interconnect, providing access to the DDR controller.

PS-PL AXI Interfaces

This section describes the PS-PL AXI interfaces that are used to connect PL-based peripherals and functions to the PS. Usage models for each interface are described in their respective sections.

General Purpose AXI Interface (AXI_GP)

The AXI_GP interface has four 32-bit general-purpose ports with two master and two slave ports; the PL is the master for two ports and the PS is the master for the other two ports. The ports are connected to the master and slave interconnect ports in the PS.

The AXI_GP ports are provided for general purpose use and are not designed for high performance.

A typical use model for the M_AXI_GP port is to connect it to the control-plane interface of a peripheral implemented in the PL. You can use the S_AXI_GP port to communicate with addressable PS peripherals from the PL. When PL peripheral performance requirements are not high, you can also use S_AXI_GP to access PS-DDR.

Accelerator Coherency Port (ACP)

The ACP provides an interface to masters in the PL for direct connection to the SCU. The PL masters maintain memory coherency with L1 caches and have direct access to the L2 cache and OCM. The ACP has the same level of connectivity to peripherals and memory as the CPU. For more information, refer to the address map in the System Addresses chapter (available at this link) of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

Addresses on the ACP port are snooped by the SCU to provide full I/O coherency. Reads on the ACP will hit in any CPU's L1 data cache, and writes will invalidate any stale data in L1 and write through to L2. This results in significant performance benefits and simplifies driver software.

The ACP allows a device, such as an external PL-DMA, direct access to CPU-coherent data regardless of where the data is in the CPU cache or memory hierarchy. The resulting system coherence relieves the software driver of performing costly operations such as cache flushing and helps improve system performance.

The ACP is optimized for cache-line length transfers and also has certain limitations that should be taken into account when designing a system with an ACP. Further details are available in ACP and Cache Coherency, page 116.

Use of ACP for high performance applications should be considered with care because the peripherals connected to the ACP will compete with the CPUs for access to caches and the memory subsystem in the PS, impacting overall performance.

High Performance Ports (AXI_HP)

Four high-performance AXI interfaces are provided to enable a high-throughput data path between the PL masters, and DDR and OCM memory in the PS. Each datapath is selectable between 32-bit and 64-bit. A dedicated memory interconnect provides direct connectivity of these AXI-HP ports to the DDR memory controller in the PS. Also, FIFO buffers for read and write traffic are present, providing high-performance capabilities. These ports are also referred to as the AXI FIFO Interface (AFI).

The four AXI-HP ports are multiplexed down in pairs and are connected to two ports on the memory controller. The memory interconnect switch arbitrates between the two ports. When using two HP ports, use of port-0 and port-2 is recommended because these are on separate DDR interfaces.

The HP AXI interface provides additional capabilities, such as QoS, FIFO occupancy, and issuance throttling on interconnect. This enables bandwidth management when there are multiple masters with different types of traffic flow. You can control the QoS signals using PL signals or by statically configuring the APB registers.

In addition to the PS-PL AXI Interfaces discussed above, there are additional PS-PL interfaces:

a. Extended MIO allows routing of IO peripheral signals to the PL. This provides an interface between the PS IOP controller and user logic in PL, and it also enables use of PL device pins. For more information, refer to this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

b. PS-PL clock, reset, and interrupt interfaces that provide clock, reset, and interrupt signals in the PL. For more information, refer to this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

c. Device configuration interface, used to configure the PL under PS software control. For more information, refer to this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

d. The DMA interface between PS and PL can be used for high-speed data transfer between the PL IP block and other peripherals or memory blocks in the PS. The interface between the PS and the fabric can be used to propagate CPU interrupt events.

System Level Design Considerations

When managing the bandwidth of PS-PL AXI interfaces, the following should be considered:

- The general-purpose ports should be used for low-to-medium types of traffic, such as controlling PL peripherals from the PS, or as a data path for low-speed PL peripherals. For example, you can move Ethernet packets into the PL through the S_AXI_GP port because the bandwidth and latency requirements are satisfied by the port. However, in applications requiring guaranteed latency and high performance, such as video implemented in PL, use of HP ports is preferable.
- The accelerator coherency port is useful when applications in the PL need to maintain coherency with the processor L1 caches, and need reduced software overhead.
- The high-performance port is recommended when applications in the PL need high-bandwidth access to the DDR controller or the OCM. Multiple ports are provided so that you can distribute the PL masters across four HP ports for improved load balancing, rather than restricting PL masters to a single HP port. You can obtain even higher bandwidth for one PL master by using multiple HP port sets (HP0 and HP2, or HP1 and HP3).

  For more information about using multiple port sets, refer to this link in the *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 4].

PL Clocking Methodology

This section covers the clock methodology in Zynq-7000 AP SoC PL, describing the different clock sources available in the PL and their recommended usage. Those clock sources are:

- Clock from PS (FCLK)
- Clock Recovered from a GT
- Clock from External Source
- Clock Generated by MMCM

At the end of this section a video system design is discussed as a typical model demonstrating use of the various clock sources.

Clock from PS (FCLK)

Refer to the Clocking and Reset in Chapter 2 for information on using FCLK.

Clock Recovered from a GT

The receiver clock data recovery (CDR) circuit in the GT transceiver extracts clock and data from an incoming data stream. The recovered clock has a direct relationship to the transceiver line rate and the data-path width. You can use the recovered clock in a PL design requiring a high-speed serial transceiver for communication with the link partner. The recovered clock typically has inherent jitter, and that jitter must be corrected before using the clock in a design that is sensitive to input jitter. In general, the recovered clock is used to sample data received by the PL logic. The architecture of the CDR is shown in Figure 3-20.

For more information, refer to the Receiver chapter of *7 Series FPGAs GTX/GTH Transceivers* (UG476) [Ref 2] and *7 Series FPGAs GTP Transceivers* (UG482) [Ref 3].

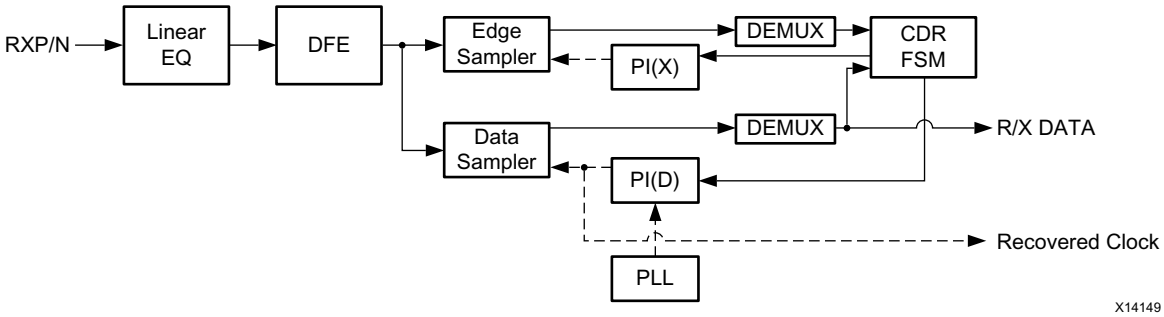

Figure 3-20: CDR Architecture

Clock from External Source

An FPGA clock-capable IO pin that supports differential clocks can supply a PL clock. If a single-ended clock is used, it should be connected to the P (master) side of the clock-capable input pin pair.

The choice of single-ended versus differential-input clocking depends on the design's clocking requirements. Differential clocking is recommended because it eliminates power-supply noise and line-coupling noise. If the design is pin limited, you can use single-ended clocking.

When a single-ended clock is connected to the P side of an input pin pair, the N side cannot be used for another single-ended clock. However, you can use it as a user IO.

The clock-capable pins provide dedicated, high-speed access to the internal global and regional clock sources, as shown in Figure 3-21, page 114.

For more information, refer to *7 Series FPGAs Clocking Resources* (UG472) [Ref 1].

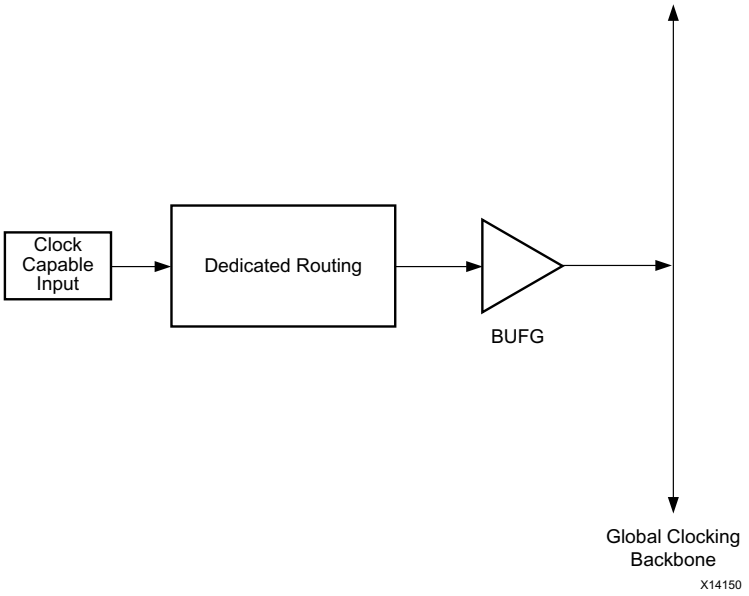

Figure 3-21: Clock Capable Inputs

Clock Generated by MMCM

The MMCM is used to generate multiple clocks with different frequency and phase relationships. The MMCM also de-skews the clock output. A use model is shown in Figure 3-22.

The MMCM primitive, MMCME2_ADV, provides the functions listed above and the input clock selection using the Dynamic Reconfiguration Port (DRP).

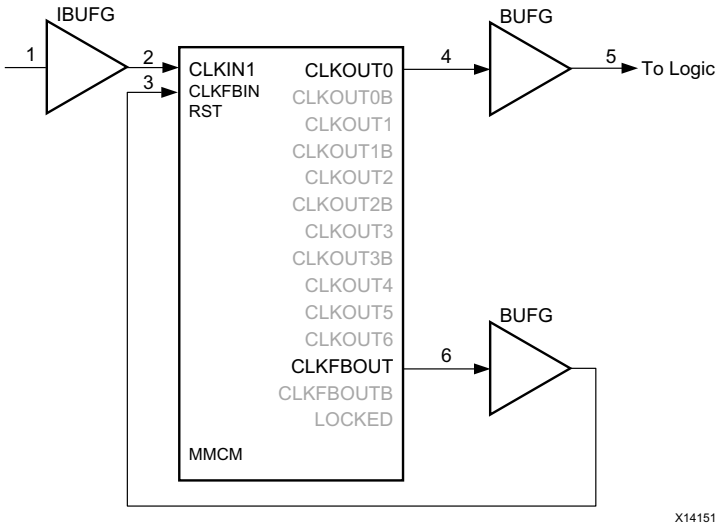

Figure 3-22: MMCM Use Model

FCLK Use Model

An example FCLK use model is a video application that displays a test pattern on a display monitor. Figure 3-23 illustrates this model.

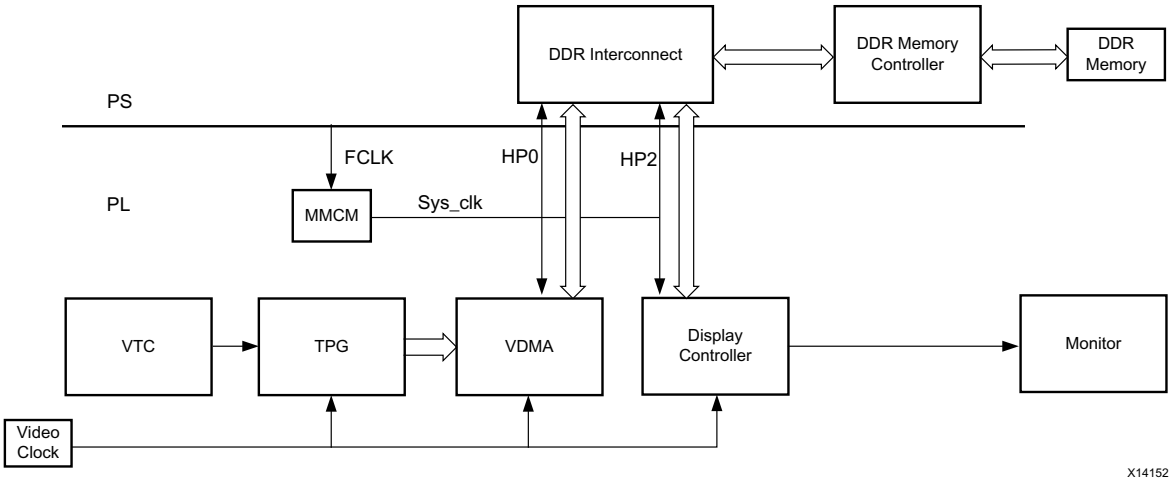

Figure 3-23: FCLK Use Model

The use model has the following IP blocks:

- **Test pattern generator (TPG)**: Generates video frames of different test patterns.
- **Video timing controller (VTC)**: Generates the timing for TPG.
- **Video direct memory access module (VDMA)**: Writes the video frames generated by TPG in to the DDR memory.
- **Display controller**: Fetches the frames from the DDR through HP2 port and displays on monitor.
- **On-board clock synthesizer**: Used as a video clock driving all input video modules and the display controller. It is programmable by the PS.
- **Clock derived from FCLK using the MMCM**: This clock has a higher frequency than Sys_clk. It is used by the VDMA and display controller AXI master interfaces because of display controller latency requirements.

In this example, the design could use an external clock routed to PL and to an MMCM that generates the desired Sys_clk output. The PS AXI interconnect provides the clock domain between the HP/GP port and the PL.

ACP and Cache Coherency

The accelerator coherency port provides low-latency access for PL masters, including optional cache coherency with the dual core ARM Cortex-A9 CPUs. From a system perspective, the ACP interface has similar connectivity as the APU CPUs. Because of this, the ACP competes with the APU CPUs for resource access outside the APU block. Figure 3-24 shows an overview of the ACP connectivity.

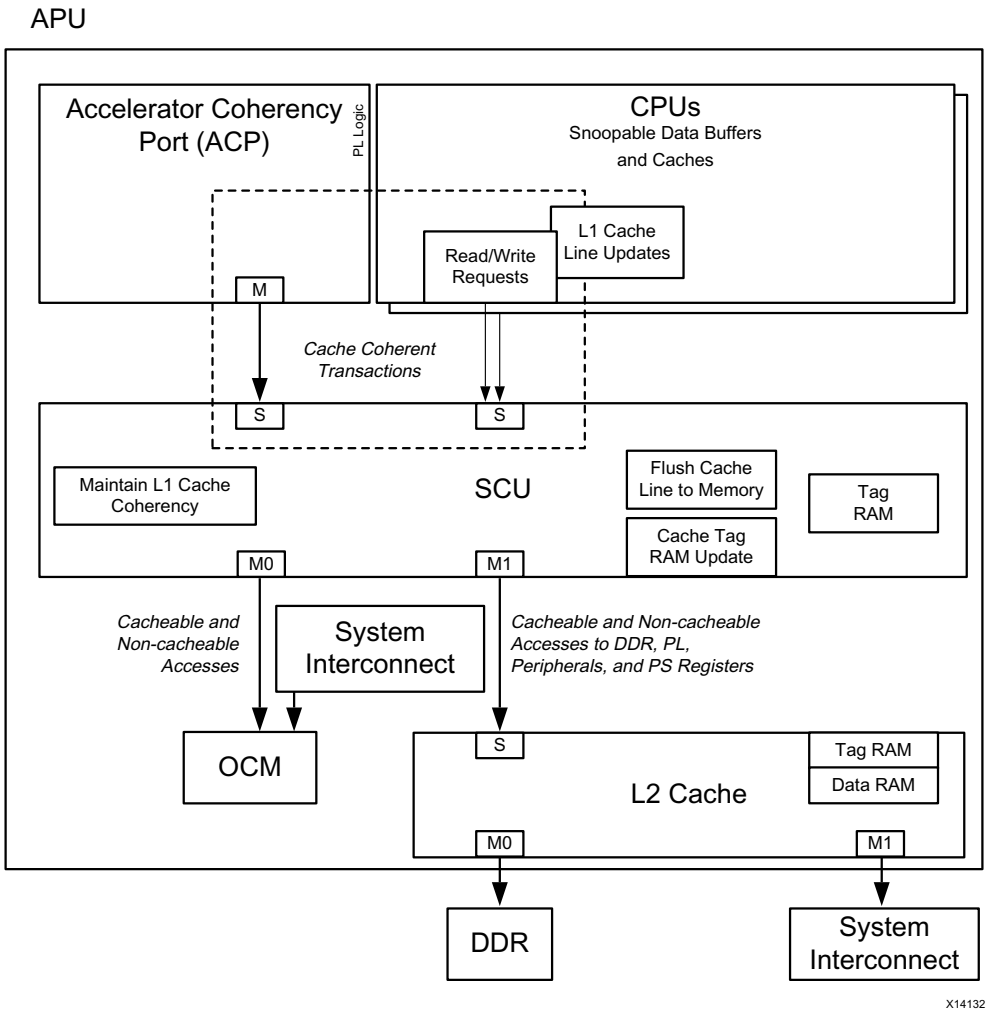

Figure 3-24: ACP Connectivity Diagram

The ACP's main benefit, compared to other PL masters, is access to CPU cache-coherent memory and use of the low-latency 512KB L2 cache. In a common Zynq-7000 AP SoC implementation, a PL accelerator is connected to the ACP port. It is configured by the CPU over one of the GP ports, and then the accelerator does its data movement over the ACP. Coherent data movement is managed in hardware, allowing the PL ACP-master to issue standard AXI reads and writes. The only requirement on ACP cache-coherent accesses is that those transactions have the A*CACHE and A*USER bits set to all 1s.

This hardware-managed coherency model is an alternative to having PL accelerators use the HP or GP ports to access non-coherent memory directly from the DDR Controller. Here, software-managed coherency would be necessary, whereby the processor would have to flush shared memory back to the DDR to guarantee coherency. While there are many ways to implement software-managed coherency, the ACP does so in hardware, easing software design.

Because the ACP supports cache coherency, ACP-attached AXI masters compete with the CPUs for L2-cache resources (the L1 cache is only available to the CPU). Any AXI read or write transaction coming from the ACP will use L2-cache resources to service the data movement (arbitration, tag matching, and cache-line fetches to DDR Controller). If this data movement is coordinated with CPU activity (such as the accelerator example above), then sharing the memory resources benefits the larger system application. If this data movement operates in parallel with unrelated CPU tasks, there are additional opportunities to mitigate contention on the CPUs and ACP masters, as described in the next section.

ACP Design Methodology

The following design methodology best-practices for effectively using the ACP are based on several customer use-cases plus performance analysis of the Zynq-7000 AP SoC ACP.

ACP-Master Design Methodology

- **Avoid Thrashing the Cache**. The 512KB L2 cache is a shared resource between the two ARM processors and the ACP. The ability of the ACP to access large quantities of data has the potential to lower software performance. Locking and partitioning the cache can help ensure the shared data has a high cache hit rate. For example, video-style accesses to the ACP port would have little cache locality and, without cache management, hurt memory system performance.

  In one example performance test case, ACP accesses that have good cache locality will show 54% lower latency when compared to an ACP access pattern with no cache locality. Results will vary by application.

- **Match AXI Transaction Parameters to ACP**. The ACP is an AXI3 compliant 64-bit interface. Transactions for AXI3 or AXI4 compliant masters attached to the ACP will be converted to 64-bit-wide transactions with a maximum burst length of 16 beats. All conversion is done using soft IP such as the protocol converter or the AXI interconnect. Converted non-64-bit transactions and high beat-count transactions have the potential to cause extra data-transfer cycles. Additional ACP access optimizations and constraints are described in the ARM Infocenter Accelerator Coherency Port web page [Ref 73].

- **Choose Blocking versus Non-blocking Transactions**. Care must be taken with transactions completing out of order on the ACP port. Under certain conditions, writes and/or reads can complete out of order unless the axresp channel is used as a barrier between subsequent transactions. Ideally, the response channel should be ignored, allowing transactions to complete as soon as data is transferred. However, care is required when issuing multiple reads and writes to the same address range. For DMA bulk transfers, this is typically not a problem. However, you can see the problem when multiple outstanding read and write transactions are issued to a small address range.

  In an example use-case, if the ACP's response channel is ignored during multiple read-modify-write operations to the same address, out-of-order read-after-write collisions can occur.

- **ACP use for non-coherent access**. It is recommended that non-cache-coherent traffic from PL be routed through the HP or GP ports. If required, however, you can set the AXI 4-bit cache fields to modify the behavior of ACP accesses. For more information, refer to Section 2.4 Accelerator Coherency Port in the *Cortex-A9 Technical Reference Manual* [Ref 75].

PS Configuration for the ACP Design Methodology

- **L2-Cache Clocking and Prefetching**. The L2 cache is clocked at the CPU clock rate and, with good cache locality, will service the majority of ACP accesses. While you can raise the PL clocks to decrease the access latency, you should also clock the L2 cache as fast as possible. You can use L2 prefetch settings to turn prefetch on and off, and use other settings to control the number of cache lines fetched per miss. For example, ACP accesses with little cache locality may not benefit from prefetching because they can cause mis-speculated fetches from the L2 to the DDR controller.

  Refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information on optimizing the PS settings for ACP accesses and modifying the L2 cache settings.

Contention with Software on CPU Methodologies

- **Consider CPU Memory Access Patterns**. The effect of cache contention with the processor is also dictated by the processor's memory-access patterns. Software that runs entirely out of a processor's L1 cache will not be affected by ACP accesses. However, ACP accesses can affect software with low cache locality, such as a memory-striding application.

  For example, a memory-intensive software application running on the ARM Cortex-A9 is unaffected by heavy ACP traffic accessing a single 4 KB memory page from the L2 cache.

  As a further example, running L1-cache-bound benchmarks, such as Dhrystone or Coremark, on the CPU is unaffected by heavy ACP traffic accessing a 512 KB buffer held within the L2 cache.

- **Advanced Cache Management.** You can use techniques such as cache locking and cache partitioning to assist in cache sharing and cache isolation between multiple CPU and ACP masters. These techniques may require you modify linker scripts and tailor ARM memory descriptors to the applications memory layout. For a tutorial on how to lock caches and the possible performance effects, refer to Zynq-7000 AP SoC Boot - Locking and Executing out of L2 Cache Tech Tip wiki page [Ref 60].
- **Monitoring Contention**. It is important to monitor and measure contention when design goals are not being met. You can monitor the L2 cache, the CPUs, and the ACP itself for ACP performance. You can monitor the L2 cache using event counters, processor cache accesses using the ARM Performance Monitoring Units (PMU), and the ACP using Xilinx's AXI performance monitor implemented in the PL. These monitoring methodologies are further described in Performance, page 13.

PL High-Performance Port Access

The high performance (HP) ports give the PL direct access to the DDR controller and to the On-Chip Memory (OCM). The Zynq-7000 AP SoC Technical Reference Manual provides an excellent functional description of the HP ports and possible performance optimizations. This section describes common design-driven optimizations implemented in customer designs that use the HP ports.

The HP ports are typically used in a design needing PL high-throughput access to either the OCM or, more commonly, DRAM attached to the DDR controller. Figure 3-25 shows that the HP ports contend with each other and other PS masters at different points in the PS architecture. Methods for achieving a high-throughput design on multiple HP interfaces with system contention are described below.

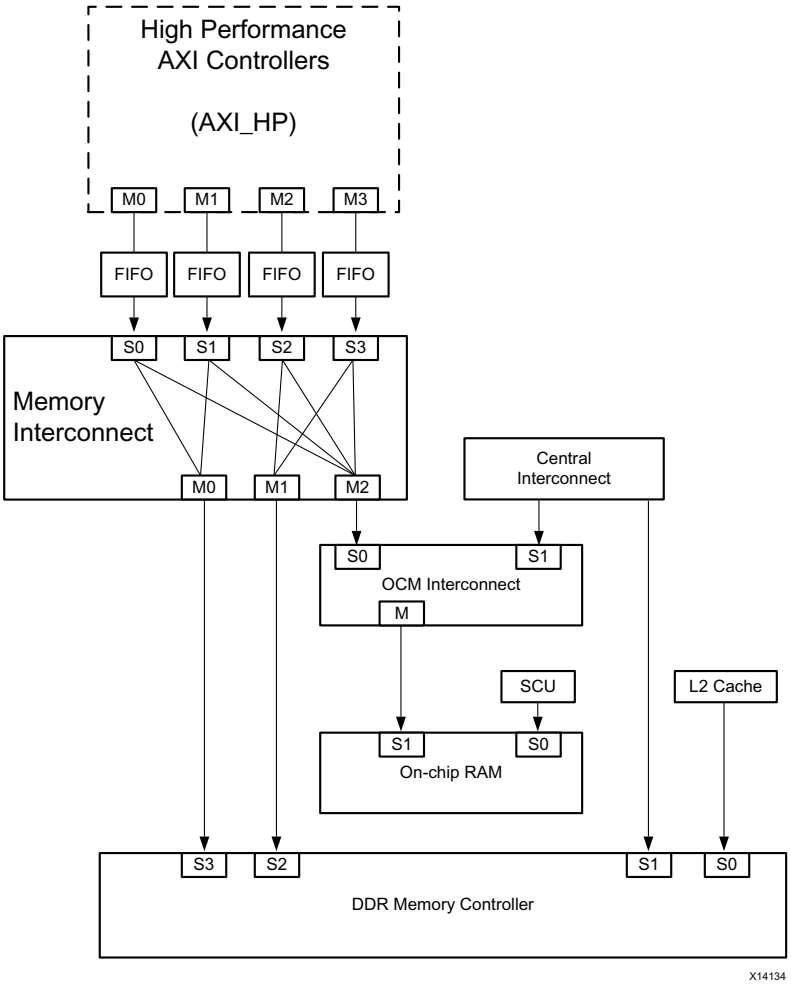

Figure 3-25: High Performance (HP) Port Connectivity

HP Master Design Methodology

- **Order of adding HP ports**. Because of the HP data paths, HP ports should be added to minimize data path contention. In general, HP ports should be added in the following order: HP0, HP2, HP1, and HP3. Using this order, HP masters can take full advantage of the two dedicated DDR controller ports.

  For example, in a design requiring two HP ports, a 15% improvement in combined throughput can be realized if heavy read traffic is routed through HP0 and HP2 instead of HP0 and HP1. As higher throughput traffic is driven, the performance difference increases.

  If more than four HP masters are required, you can use an AXI interconnect in the PL to multiplex the masters onto the four HP ports. When assigning the masters to HP ports, the goal is to maximize HP port throughput. For example, you can assign high-traffic masters to separate HP ports and pair them with low-traffic masters. An important consideration is that a single HP port could saturate at a lower throughput than the DDRC's saturation point.

- **AXI Addressing Optimizations**. Standard DRAM access optimizations should be considered, including address optimizations and minimizing read/write mix, if possible. *Designing High-Performance Video Systems with the Zynq-7000 All Programmable SoC* (XAPP792) [Ref 34] provides guidance on using nonlinear addressing to match the row-bank-column layout of DRAM for a video application. Also, whenever possible, the read/write mix should be minimized, although this is not always possible due to design behavior.

  For example, in a system with read-only HP0 and HP2 ports, the read throughput on HP0 drops 0 12% when HP2 is changed to write-only. The performance degradation range depends on traffic addressing and generated transaction rate.

- **Increasing the PL clock rate**. Increasing the programmable clock speeds is a common method used to maximize application performance. However, there are diminishing returns on increasing the PL clock rate when performance bottlenecks exist in moving data between the PL and the DDR controller. Referring to the data path in Figure 3-25, although the PL master initiates traffic in the PL clock domain, the majority of the data path is in PS clock domains. Also, the maximum DDRC bandwidth should be considered because the DDRC will eventually saturate due to increased PL master traffic.

  For example, in a design where two HP ports are driving heavy read and write traffic to the 533MHz DDR controller, increasing the PL clock speed from 50 MHz to 100 MHz increases total HP throughput 92%, whereas an increase from 100MHz to 200 MHz increases throughput 13%. In the latter case, traffic is driven more often from the PL, resulting in a saturated DDR controller.

PS Configuration for the HP Design Methodology

When designing with HP ports, the contention of high-throughput traffic at the DDR controller and attached DRAM destination cannot be ignored. The HP ports have two dedicated ports to the DDR controller, but those are arbitrated with the L2 Cache and the central interconnect in the DDR controller. You can minimize contention from the 512KB L2 cache port because the L2 cache services CPU and ACP requests if the L2 cache holds the requested data. However, if the CPUs or ACP access memory with little cache locality, those cache accesses are serviced by the DDRC, potentially hurting HP performance. Also, the central interconnect contention at the DDRC is application dependent, with the potential for heavy traffic coming from USB or gigabit Ethernet masters.

You can use two advanced techniques to limit L2 and central interconnect access to the DDRC, with a goal of improving HP throughput in high-contention implementations. You should use these techniques after default settings are tried and an HP performance goal is not met. You can modify either of the following:

- The DDRC priority settings
- The QoS-301 settings within the PS

The techniques can prioritize HP traffic to the DDR, but do not necessarily increase overall performance. Instead, they slow down other PS masters to the benefit of the HP ports.

- **DDR Controller Priority Settings**. These settings control the order that transactions are serviced at the DDR controller ports and at the DDR PHY. The high priority read feature and go2critical counter values are the most important features controlled by the DDR controller priority settings. You can use the high priority read options on each DDR controller port to increase a port's read priority over that of other DDR controller ports. This increase in priority can have an effect on the HP throughput of highly loaded systems (such as where all HP ports are driving high throughput traffic). These advanced features are useful when a Zynq-7000 AP SoC is pushed to its performance limits, such as when the DDR controller is saturated at theoretical maximum throughput. Refer to this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.
- **QoS-301 Settings**. The AXI data path from the L2 cache to the DDR controller has QoS-301 settings that are currently accessible only through direct register writes from your application. You cannot modify them using the Vivado GUI. This feature allows transaction throttling on that path, benefitting HP port performance under heavy DDR traffic. This feature is best used when memory-intensive software with poor L2-cache locality is running. Refer to this link in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4] for more information.

Contention Mitigation for DDR Controller Methodologies

Similarly to the ACP contention sources, HP performance is affected by system loading, specifically at the DDR controller. The biggest potential source of contention will be the L2 cache making cache line requests to service CPU or ACP accesses. CPU and ACP contention will depend on memory access patterns exceeding the capacity of the 512KB L2 cache. Ethernet and USB traffic from the central switch can also drive large amounts of traffic to the DDR controller.

- **Monitoring Contention**. It is important to monitor and measure contention when design goals are not being met. For HP performance, you can monitor the L2 cache, Ethernet controller, and the HP ports. You can monitor the L2 cache and Ethernet controller using event counters within the core, and monitor the HP ports using Xilinx's AXI performance monitor implemented in PL.
- **How much contention is an issue?** Most contention-mitigation techniques noticeably improve performance when the total system traffic to the DDR approaches the DDR controller's theoretical maximum. For example, a 32-bit DDR controller running on the ZC702 board at 533MHz can theoretically sustain 4.3GB/s of memory throughput. The DDR controller is designed to arbitrate fairly and sustain several high-throughput sources, and contention effects will not be visible at low DDR controller utilization.
- **Using an attached PL Memory Interface Generator (MIG)**. Xilinx also provides PL memory interfaces to off-chip DDR that can supplement or replace the HP to DDR controller data path. Functionally, you can isolate this added memory from the PS memory map, or you can access it using the GP ports. For more information, refer to Xilinx Answer Record 58387 [Ref 71].

  The MIG is typically a better-performing memory option for large bulk data transfers to contiguous addresses from a single master, such as Ethernet and video frames. Also, this memory option is best for buffering data isolated from PS accesses, because direct CPU accesses to this PL DDR would go through the 32-bit GP ports, limiting MIG performance. Otherwise, the PS DDR controller provides excellent memory performance for HP workloads.

System Management Hardware Assistance

This section describes the hardware components that enable system management in a Zynq-7000 AP SoC PS. System management includes control of system-level parameters based on user-specific inputs. Chip-level management includes control of die temperature, management of errors, implementation of low-power mode during system inactivity, management of secure and non-secure access to peripherals.

Software components play an important role while performing system management. The focus of this section is on hardware assistance for system management on Zynq-7000 AP SoCs.

Xilinx Analog-to-Digital Converter

The Xilinx analog-to-digital converter (XADC) is a flexible analog interface that resides in PL and assists in monitoring temperature and power-supply sensors. The XADC has three interfaces, JTAG, DRP and PS-XADC, with which system applications can monitor the temperature and voltage status and take appropriate action. The JTAG and DAP interfaces are present in all 7-series FPGAs. However, the PS-XADC interface is only present in Zynq-7000 AP SoC devices.

Figure 3-26 shows how the XADC arbitrates between the AXI XADC DRP interface and the PL-JTAG or PS-XADC interface.

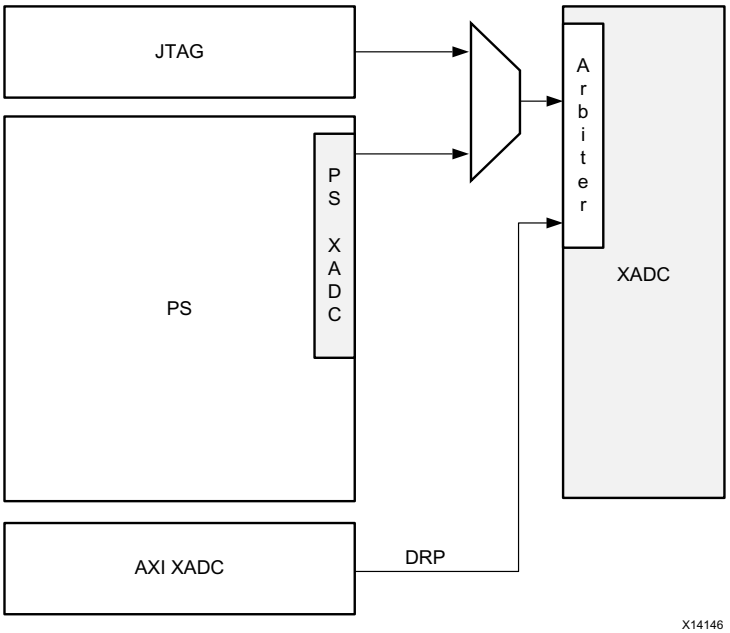

Figure 3-26: XADC Arbitration

TrustZone Security

The ARM TrustZone technology supports system-wide security at multiple levels, including hardware, software, and memory. The Zynq-7000 AP SoC PS architecture supports multiple operating modes, including supervisor, system, and user modes, to provide different levels of protection at the application level.

A processor switches between two separate worlds, secure world and normal world, through a processor mode called monitor mode, as shown in Figure 3-27. The TrustZone technology is implemented in many PS components, including: APU, L1-cache controller, memory management unit, SCU, SLCR, triple timer counter, watch-dog timer, I2C, GPIO, SPI, CAN, UART, QSPI, NOR, DDR memory controller, L2-cache controller, AXI interconnects, on-chip memory, DMAC, Gigabit Ethernet controller, SDIO controller, and USB controller.

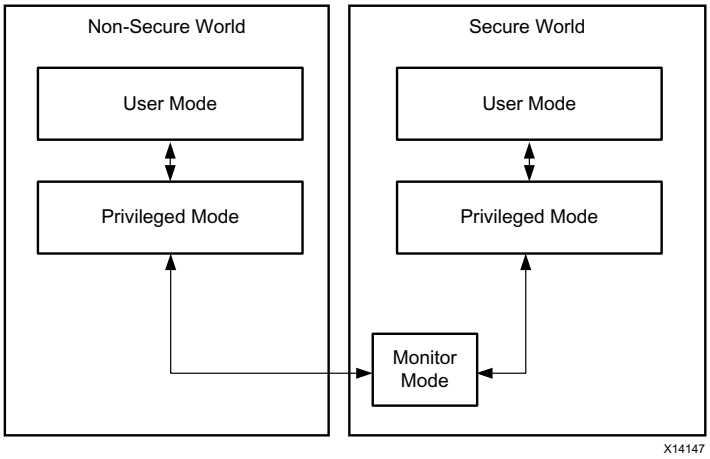

Figure 3-27: Switching Between Secure and Non-Secure World

DDR Memory Controller

You can configure DDR memory in 64MB segments, and configure each segment to be secure or non-secure using a control register, TZ_DDR_RAM (0xF8000430).

- A bit value of 0 indicates a secure memory region for the associated memory segment.
- A bit value of 1 indicates a non-secure memory region for the associated memory segment.

When there is non-secure access to a secure segment of DDR memory, a DECERR response is returned to the initiator of the transaction. Write transactions are masked and result in no write to the DDR memory, whereas read transactions return all zeroes.

L2 Cache Controller

The L2 cache controller supports TrustZone technology by adding an NS bit to cached data. In this way, the cache controller treats secure and non-secure as being in two different memory spaces. For a read transfer, the cache controller sends a line-fill command to external memory, propagates any security errors from external memory to the processor, and does not allocate the line in L2 cache if there is an error.

DDR Error Recovery

An error correction code (ECC) mechanism is supported for 16-bit DDR content. This allows recovery of corrupted DDR data. Ten ECC bits for each 16 data bits support correction of single-bit errors and detection of two-bit errors. When ECC is enabled for writes, ECC code is computed and written to DDR along with the data. When ECC is read, the DDR controller checks the data against the stored ECC code. Writing DDR memory without enabling ECC, and then reading with ECC enabled, may return ECC errors.

When an error is correctable, the controller corrects the data and sends it without generating an interrupt or AXI response. When an error is uncorrectable, the controller sends an AXI SLVERR response along with the corrupted data to the AXI master. AXI masters in the PS might also generate L2 or DMA interrupts, or CPU prefetch or data exceptions. When they do, the same AXI SLVERR response is sent to the PS AXI master.

Low Power

You can power-optimize Zynq-7000 AP SoCs in various ways: PL power off, processor standby mode, clock gating of PS subsystems, PLL configuration, or I/O voltage control.

When the PL is not in use, you can power-off the entire PL. However, the PL loses state when it is powered off, and it must be reconfigured when it is powered on again. The system software should determine when to power-off the PL and it should reconfigure the PL when it is needed again.

Clock Gating

The PS has multiple clock domains, and each clock has gating control to stop the clock. System software can enable gating of a particular clock when that clock domain is not in use. This reduces dynamic power dissipation.

You can disable clocks using system-level control registers (SLCR), which starts at address 0xF8000000.

For example, when the SPI controller is not in use, you can disable the controller reference clock by writing an appropriate value to register SPI_CLK_CTRL (0xF8000158).

Figure 3-28 shows how clock gating of the SPI reference clock works.

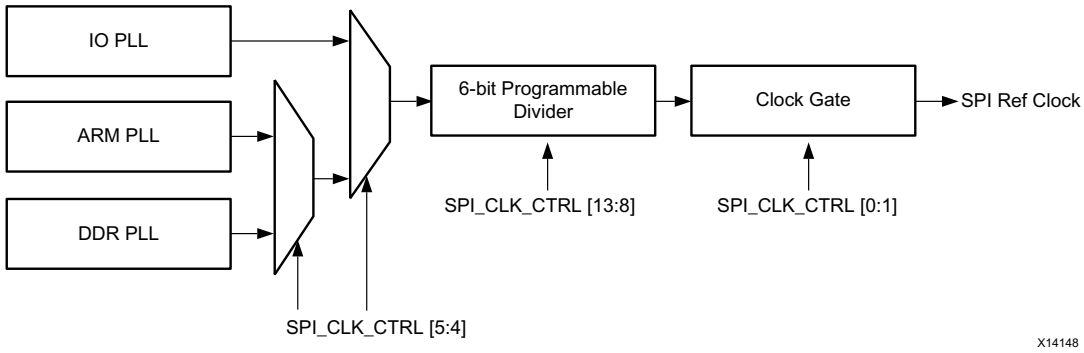

Figure 3-28: Clock Gating

Managing Hardware Reconfiguration

Partial reconfiguration is the ability to reconfigure a portion of an FPGA chip to implement different logic functions without disturbing the remaining logic on the chip. The technology enables an FPGA to implement different functional blocks in a time-sliced manner.

TIP: *The Partial Reconfiguration Design Hub in the Documentation Navigator provides links to additional information about Partial Reconfiguration. For more information, see* *Related Design Hubs, page 225*.

FPGA partial reconfiguration is supported using the following constructs:

- A configuration-select line that enables partial reconfiguration of an SRAM FPGA without powering down the supply voltage.
- A configuration latch that can store the configuration data used for partial reconfiguration.

In the Zynq-7000 AP SoC PL architecture, the basic programmable unit is a configuration frame composed of a LUT, DSP48, and BRAM. In partial reconfiguration, the configuration frames are reprogrammed using a Processor Configuration Access Port (PCAP) that acts as a reconfiguration agent. Each configuration frame has a unique address identified with a top/bottom address bit, a major address, and a minor address. The partial bitstreams are generated using a Xilinx Vivado Tool flow, and the partial bitstreams are parsed by the PCAP to reconfigure the frames.

Partial reconfiguration offers the following advantages over a traditional full configuration:

- **Reduced hardware resource utilization**: A designer can fit more logic into an existing device by dynamically time-slicing design functions.
- **Increased productivity and scalability**: Only the modified function needs to be implemented in context with the already-verified design
- **Enhanced maintainability and reduced system down-time**: You can deploy new functions and insert them dynamically while the system is up and running

Figure 3-29 shows a conceptual view of the partial reconfiguration methodology. The static design does not change over time and establishes communication between the partial reconfiguration modules and the static design. The communication between the static logic and the partial reconfiguration modules occurs over a set of communication macros consisting of a bidirectional communication entity and tri-state gates.

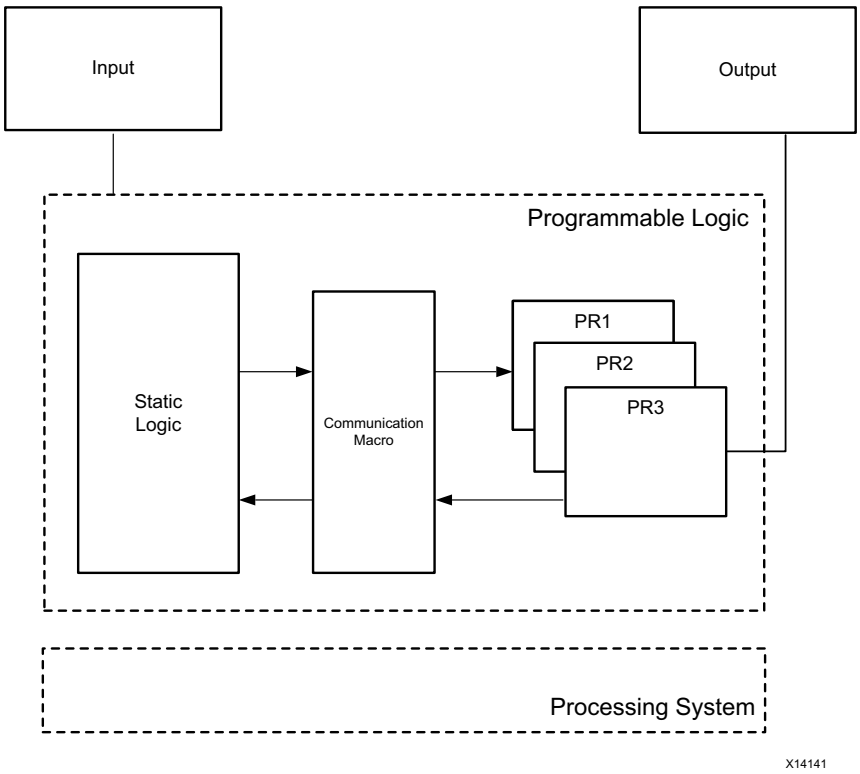

Figure 3-29: Conceptual View of Partial Reconfiguration

The Zynq-7000 AP SoC has a PL portion that you can reconfigure using partial reconfiguration. The PL is programmed using the Processor Configuration Access Port (PCAP) that is part of the PS Device Configuration Interface. The PCAP block communicates with the CPU and system memory using a PCAP to APB bridge that converts APB transactions to PCAP read/write transactions. You can store the partial bitstreams generated using the Xilinx Vivado design tool into the PS DDR memory, and retrieve them to configure the PL over the PCAP interface using the partial reconfiguration methodology.

Partial Reconfiguration Terminology

The reconfigurable partition refers to the physical location on the FPGA selected for partial reconfiguration. The remainder of the design is referred to as static logic. You can refer to a specific design implementation as a reconfigurable module. A configuration defines a complete FPGA design and produces a full bitstream for a reconfigurable module and static logic, plus a partial bitstream for the reconfigurable module.

System Level Considerations

When using partial reconfiguration in your design flow, you should take the following system-level considerations into account.

RECOMMENDED: *To learn how to design with Partial Reconfiguration in Vivado, refer to the Partial Reconfiguration Design Hub in the Documentation Navigator. For more information, see* *Related Design Hubs, page 225*.

Hardware Design Flow

The first step in creating a partially reconfigurable design is to create a design using a reconfigurable module and static design flow. Implementation of a partial reconfiguration design flow requires the Xilinx Vivado design tool, as described in the following steps:

- Create a partial reconfiguration project using the Xilinx Vivado design tool targeting the evaluation platform, and import the netlists and constraint files.
- Define the reconfigurable partition. This partition ensures that the logic and routing common to each of the multiple designs is identical.
- Create reconfigurable modules for the reconfigurable partition by adding the corresponding netlist and constraint files. Constraints that apply only to specific reconfigurable modules must be scoped to the module level and should be provided with the corresponding netlist. Constraints applied to the static logic and any constraints that are shared across all reconfigurable modules should be included in the top-level constraint file.
- Floorplan the reconfigurable partition by setting the physical size of the partition and the types of resources desired. Xilinx FPGAs support reconfiguration of CLBs (Flip-Flops, LUTs, distributed RAM, Multiplexers, etc.), BRAM, and DSP blocks, plus all associated routing resources. The designer must floorplan the reconfigurable partition such that it can accommodate the resources required by the reconfigurable modules. As a guide, 20% overhead should be assumed for routing resources. The location of a reconfigurable partition inside the PL with respect to the static logic depends on the data flow and how the reconfigurable modules communicate with the rest of the design. A simple strategy is to implement the configuration with the highest resource utilization without floorplanning, identify the region where most of the resources are

placed, and create a partition around this region that is big enough to hold all the resources.

- When building reconfigurable design configurations, the first configuration implemented should be the most challenging one.
- Run the partial reconfiguration verify-configuration utility to validate consistency between the reconfigurable-module implementations.
- Generate full and partial bitstreams for the reconfigurable modules.

Figure 3-30 shows the hardware design flow implementing partial reconfiguration.

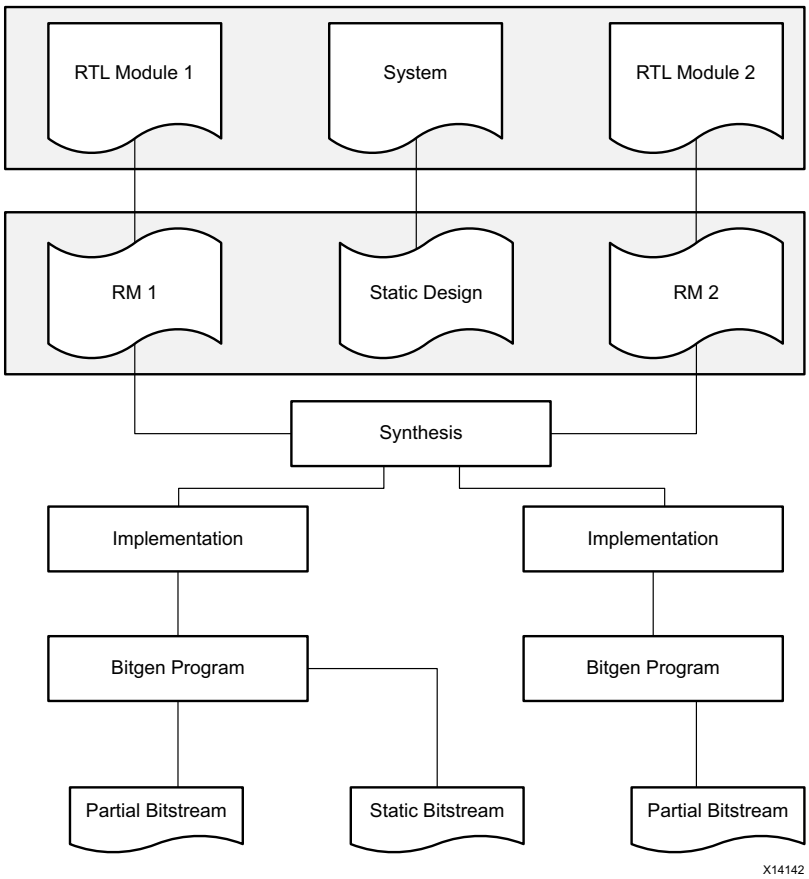

Figure 3-30: HW Design Flow for Partial Reconfiguration

System Design Flow

The Device Configuration (DevC) interface containing the AXI-PCAP bridge is used to implement the partial reconfiguration flow on a Zynq-7000 AP SoC.

The following example summarizes the boot sequence for partial reconfiguration:

1. After power-on reset, the BootROM determines the external memory interface or boot mode (SD flash memory) and the encryption status (non-secure). The BootROM uses the DevC's DMA to load the First Stage Boot Loader (FSBL) into on-chip RAM (OCM).
2. The BootROM shuts down and releases CPU control to the FSBL which in turn configures the PL with the full bitstream via the PCAP.
3. The FSBL loads and releases control to the user application in bare-metal OS.
4. The user application loads the partial bitstreams into DDR memory during start-up. This is to maximize the configuration throughput over the PCAP interface, speed up the configuration time, and take advantage of cache.
5. The application can use the partial bitstreams at any time to modify the pre-defined PL regions while the rest of the FPGA remains fully active and uninterrupted. This is done by transferring the reconfigurable module bitstream from DDR to the PL via PCAP.
6. A single configuration engine handles both full configuration and partial reconfiguration. The task of loading a partial bitstream into the PL does not require knowledge of the reconfigurable module's physical location because configuration-frame addressing information is included in the partial bitstream

Figure 3-31, page 132 shows the system level flow for partial reconfiguration. The Xilinx device configuration driver implements low-level APIs to interact with the PCAP block, and the user application can call the APIs to initiate the partial reconfiguration flow. The user application must be aware of the source addresses in the DDR where the partial bitstreams are stored.

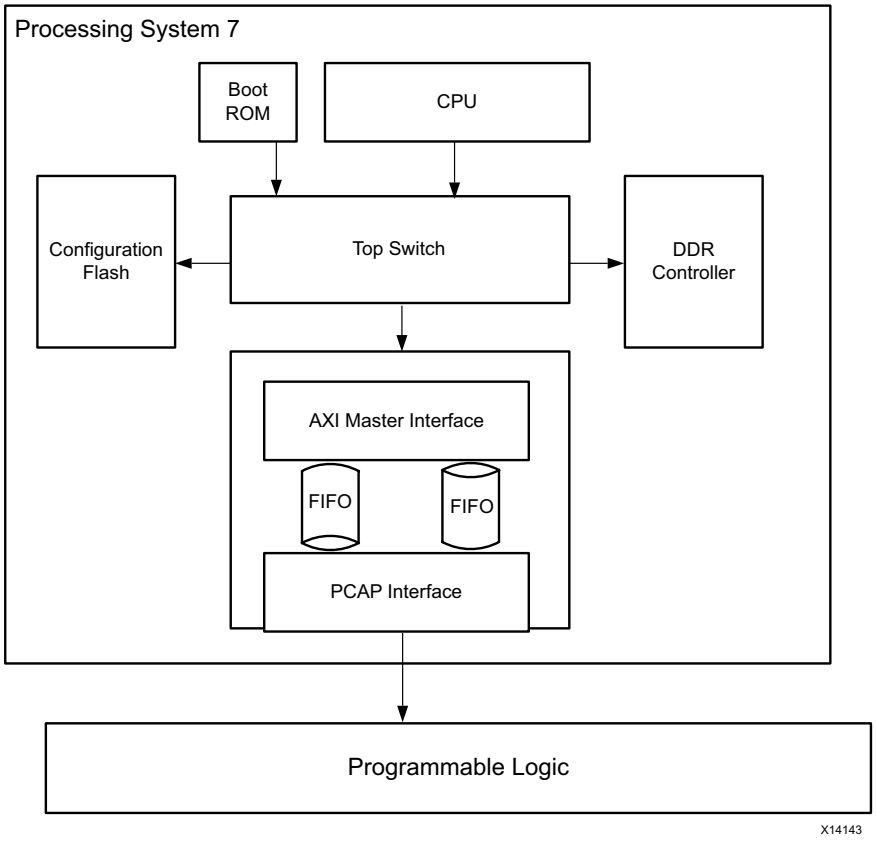

Figure 3-31: System Level Flow for PR

The example above shows how to load partial bitstreams from the PS-DDR memory. In general, the bitstreams can exist in any external memory that the device configuration interface can access.

Managing Reconfiguration using U-Boot and Linux

You can use Linux and U-Boot to dynamically load the reconfigurable bitstream to the PL. If an application is running on Linux and a portion of the hardware must be reconfigured, the application can send a request to the Linux kernel to load the appropriate PL bitstream and the corresponding kernel module supporting the hardware device.

If the FSBL execution time is critical, you can use partial reconfiguration to partition the design so that the smaller bitstreams are loaded faster during FSBL execution. This reduces overall boot time. U-Boot can configure and initialize the remaining design. Partitioning is useful when a specific PL functions must be started within a specific time limit.

You can encrypt the partially reconfigured bitstream using an encryption key. You can decrypt using either U-Boot or Linux, depending on the application requirements. Once decrypted, you can use the bitstream to configure the PL using the PCAP interface.

GPs and Direct PL Access from APU

This section describes APU access to registers and memory in the PL through its AXI general-purpose interfaces, the address requirements for AXI slaves in PL, and issues relating to security and performance. A system design is described that deals with direct PL access by ARM Cores and represents a typical use case.

The APU in PS can access registers and memory in PL using GP master AXI interfaces, which are part of the PS-PL interface. There are two GP master AXI interfaces that the APU can use to initiate an AXI transaction to a slave implemented in PL.

M_AXI_GP ports have the following features:

- AXI-3 protocol.
- Data bus width of 32-bits.
- Master port ID width of 12-bits.
- Master-port issuing capability of eight reads and eight writes.

The AXI interfaces, M_AXI_GP0 and M_AXI_GP1, each occupy 1GB of system address space, as shown in Table 3-3.

Table 3-3: **Address Range of GP Ports**

Interface	Low Address	High Address
M_AXI_GP0	0x40000000	0x7FFFFFFF
M_AXI_GP1	0x80000000	0xBFFFFFFF

All APU accesses within the address ranges shown in Table 3-3 pass through the Interconnect in the PS to either M_AXI_GP0 or M_AXI_GP1, depending on the address. If the transactions initiated by the APU are outside these address ranges, the transactions will not be routed to master GP ports.

Figure 3-32 shows the path used by an APU accessing an AXI slave in PL.

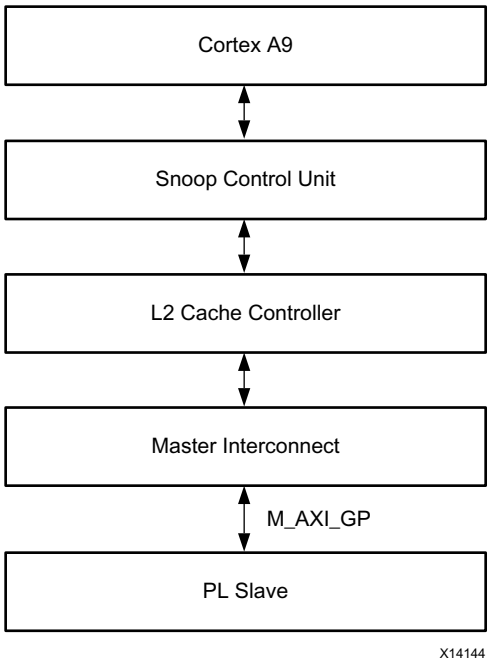

Figure 3-32: APU Accessing PL Slave through GP Port

Performance

The GP ports are directly connected to the PS master interconnect ports without FIFO buffering, unlike HP ports that have FIFO buffering to increase performance and throughput. Therefore, GP ports are for general purpose and not for high performance. The GP port has a latency of 38 M_AXI_GP interface clock cycles.

Design Considerations for AXI Master GP Port

- When AXI interconnect in the PL is connected to the PS GP master AXI interface and the HP AXI slave interface, you need to disable HP slave port access from the master port GP. In the IP integrator address editor under processing_system7 addressing, un-assign the HP port address to avoid the address conflict.
- Accesses to the GP interface are not cached by default. You can cache accesses by changing the Translation Lookaside Buffers (TLBs).
- When using the GP interface with an AXI interconnect in PL, system software should never access an address that does not exist in the AXI interconnect. This will avoid permanent AXI interconnect lock-up (there is no timeout set by the AXI interconnect).
- The voltage translators at the PS-PL boundary must be configured before the APU uses the master GP interface to access the AXI slave in PL. This is normally done by the FSBL.

Use Case

Figure 3-33 shows how the GP master AXI interface on the PS access AXI slave registers in the PL to configure peripherals, read status, and access DDR memory.

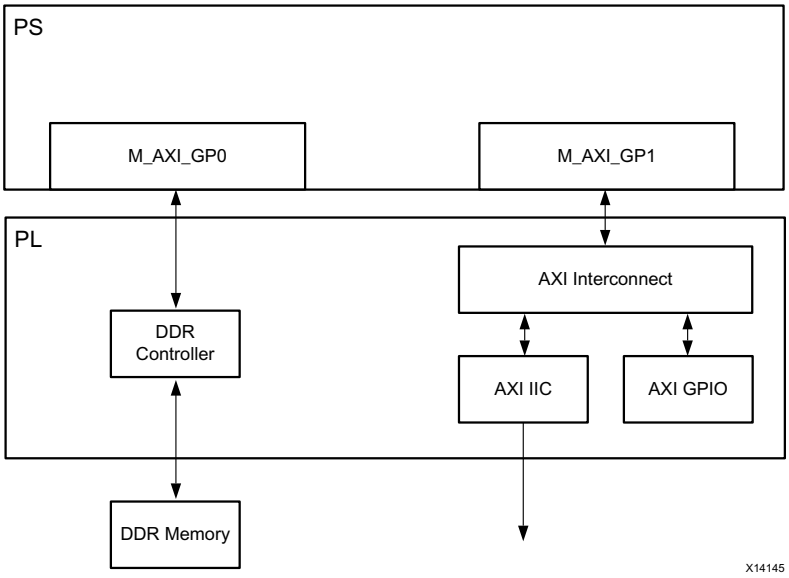

Figure 3-33: APU Accessing AXI Slaves in PL

The GP0 port is connected to a DDR controller to increase the DDR memory available to the PS through the DDR controller in the PL. The GP1 port is used to configure the AXI IIC and AXI GPIO IP blocks. The TLB has to be changed to allow caching of PL DDR memory.

Software Design Considerations

This chapter covers the following software design issues to be considered when designing with a Zynq®-7000 AP SoC:

- Processor Configuration, page 137: Configuring the processor to meet system requirements is an important consideration in implementing an embedded system. This section will help you optimize the ARM cores to best fit your application needs.
- OS and RTOS Choices, page 142: This section will help you understand the pros and cons when selecting the appropriate software platform for your embedded system — OS, RTOS, or bare metal.
- Libraries and Middleware, page 152: Libraries help you combine common and useful functions in a single place so that they can be re-used by your applications. Middleware sits on top of the operating system and reduces complexity by centralizing software stacks that otherwise would be redundant in an application layer. This section describes how you use libraries and middleware to improve application-programming productivity and reliability.
- Boot Loaders, page 156: There are a variety of ways you can use to boot and bring up applications in your embedded system. This section describes the Zynq-7000 AP SoC boot flow options from power-up to application execution.
- Software Development Tools, page 162: The Xilinx software development tools you can use to build software components for your Zynq-7000 AP SoC are described in this section.

Processor Configuration

To fully use an SoC in an embedded system, it is important to configure the processor to meet system requirements. This section describes the aspects to consider. The content here is specific to Zynq-7000 AP SoC systems, but it can be applied to any system.

Clock Speed and Multiple Cores

A good balance between power and performance can make an embedded system highly effective and efficient. Processors can be configured to run at lower clock frequencies than the maximum limit, potentially saving power. If an operating system supports multiple processor cores, power could be saved by running the cores at a lower frequency instead of a single core at a high frequency. Power savings can be determined by analyzing the power requirements and consumption of the multiple processor cores running at the lower clock frequency and comparing the result with a single core running at a high clock frequency.

The processor clock frequency can be dynamically changed by system software. When running both processor cores, they always run at the same frequency. Multi-threaded software that scales across cores helps take full advantage of a multicore system. Such software can increase performance by executing threads in parallel, without pushing the system to the maximum clock frequency.

SMP and AMP Configuration

Deciding to use a multicore processor affects the software system design, including operating system selection. Most multicore SoCs have an option to run the system as an asymmetric multiprocessor (AMP) or symmetric multiprocessor (SMP).

An AMP configuration uses each processor core for different (usually unrelated) tasks. Each core works independently, and at various times signals or synchronizes with other cores to exchange messages. Typically, each core has its own memory address space, except for the memory used to pass messages. In practice, the AMP configuration is typically used to run different operating systems or different, independent instances of the same operating system.

An SMP configuration uses multiple processor cores to perform tasks within the same operating environment, such as when multi-core Linux runs multiple tasks simultaneously. A common scheduler is responsible for dispatching different tasks on different processor cores. The operation is symmetrical because the system view from each core is identical.

Different AMP and SMP configurations use different cache coherency and memory-sharing implementations. The system's AMP or SMP behavior can be controlled using the processor core's system-control private registers (for example, CP15 in the ARM v7 core).

Figure 4-1 and Figure 4-2 show typical examples of AMP and SMP systems, respectively (processor refers to processor core).

Figure 4-1 shows two AMP examples:

- In example 1, each processor is allocated private memory, and that memory is not shared between the processors. Both processors have access to common memory and a set of peripherals.
- In example 2, one processor does all memory operations but cannot access the peripherals. The second processor accesses the peripherals but does not perform memory operations.

In Figure 4-2, both processors have equal access to all peripherals and memory.

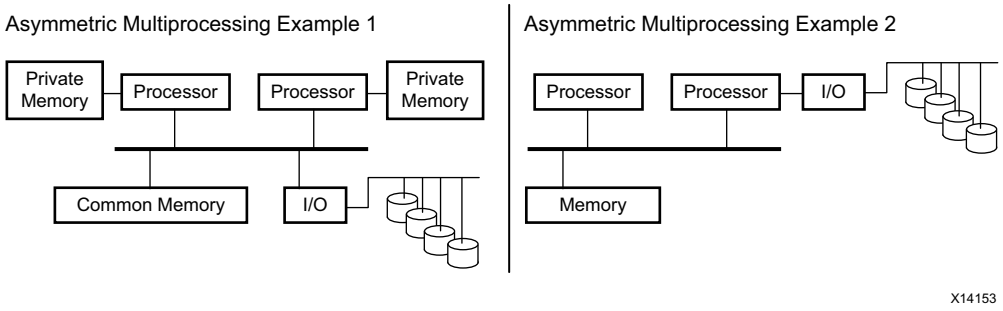

Figure 4-1: **Typical AMP Configurations**

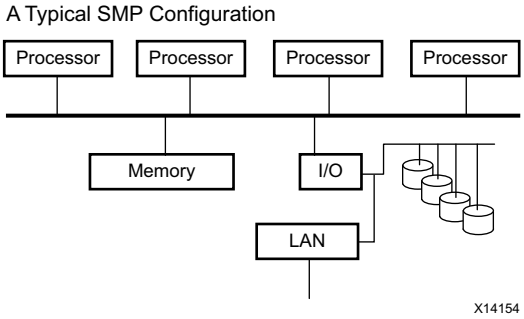

Figure 4-2: **Typical SMP Configuration**

Using Coprocessors

Hardware coprocessors can improve system performance. For example, using NEON-specific instructions can improve performance if the system will perform lots of floating-point or SIMD operations. Those instructions use the NEON media coprocessor capability in the ARM Cortex-A9 MPCore processor.

Although the NEON technology is power-optimized, most coprocessors are power-hungry. However, because coprocessors are designed for specific tasks, they are faster and more power efficient than using the main processor for performing those tasks. Coprocessors reduce CPU bandwidth and allow CPU power reduction when workload is reduced.

Co-processors execute specific instructions designed for their use. For example, the *Zynq-7000 AP SoC Spectrum Analyzer part 2-Building ARM NEON Library Tech Tip* [Ref 62] shows how to use NEON libraries to target the NEON capabilities of a Zynq-7000 AP SoC.

Cache Considerations

Caches can greatly improve system performance and should be used when possible. Most SoCs have integrated L1 and L2 caches. Each CPU core typically has a dedicated L1 cache that can be individually configured for use by that core. The L2 cache is usually shared by all CPU cores. Coherency between the caches is maintained by a snoop control unit (SCU), and coherency decisions are determined by the AMP or SMP multi-processor configuration.

The caches should be invalidated before they are enabled. The L1 instruction cache must be enabled at the beginning of the boot process (typically, by the first-stage boot loader), and it should not be disabled thereafter. The L1 and L2 data caches can be configured and enabled at a later time.

All cache operations are done on cache lines. The cache controller can write back (flush) and invalidate cache lines by cache way or by physical address.

Usually, an SoC contains differently ordered memory (such as strongly ordered or device memory), and the system can use or not use cache operations on that memory. Also, there can be cache policies, such as write-through or write-back, that are program specific.

If any system peripheral uses memory that is shared with the CPU, coherency between that memory and the CPU cache should be maintained by software using cache operations (which are costly). Similarly, in some complex systems, a peripheral might have access to cached memory for improved performance. In that case, coherency between the peripheral cache, memory, and the CPU cache should be maintained by software. The software overhead of maintaining coherency introduces latency. The latency can be avoided by using the hardware coherency provided by cache-coherent ports, such as the accelerator coherence port (ACP).

Zynq-7000 AP SoC systems provide an ACP that gives external ports access to processor caches. An ACP allows other non-cached master peripherals and accelerators, such as a DMA engine or cryptographic accelerator core, to be cache-coherent with the processor core L1 caches.

Processor State after Power on Reset

In the Zynq-7000 AP SoC, both processor cores start after power-on reset (POR). CPU0 starts at address 0x0, where BootROM loads the first-stage boot loader. CPU1 is sent into a wait-for-event (WFE) loop. Any interrupt will cause CPU1 to wake up and exit the WFE state. The boot loader, kernel, or equivalent software is typically responsible for starting CPU1 at a later time, when it is going to be used.

In most cases, DDR memory is not initialized at POR, so the first-stage boot loader is loaded into simpler memory, such as on-chip memory, and is mapped to address 0x0 in the SoC.

ARM processors start in supervisor (SVC) mode after POR. Then, the kernel or equivalent software changes the ARM processor mode as required.

Interrupt Handling

ARM cores provide a generic interrupt controller (GIC) and private peripheral interrupt (PPI) controller for handling interrupts. The GIC has a flexible approach to inter-processor communication, and the routing and prioritization of system interrupts. Software can control up to 224 independent interrupts. Each interrupt can be distributed to one or both processor cores, hardware-prioritized, and routed between the operating system and the ARM TrustZone technology software management layer (see Secure Configurations, page 142). There are also priority mask registers that allow interrupts below a specified priority to be ignored.

Apart from the GIC, a processor core can mask all incoming interrupts by setting an interrupt mask in a processor system control register. In ARM processors, the IRQ and FIQ lines can be masked independently for each processor core.

Interrupt handlers should be configured at the beginning of the kernel boot-up. There are three possible interrupt sources:

- Private Peripheral Interrupts (PPIs):

  These per-processor-core interrupts have limited function. They include FIQs, CPU watchdogs, CPU timers, and some IRQs which might include interrupt signals from FPGA logic.

- Shared Peripheral Interrupts (SPIs):

  These interrupts are shared between the processor cores. The GIC decides how to distribute the SPIs to particular processor cores. These are generally hardware interrupts, and there can be large numbers of them.

- Software Generated Interrupts (SGIs):

  These are synchronous interrupts generated by writing an interrupt number on a particular SGI register on the system.

You can configure the GIC registers to handle the interrupts listed above. Figure 4-3 shows the interrupt routing mechanism of GIC v1.0 (PL310).

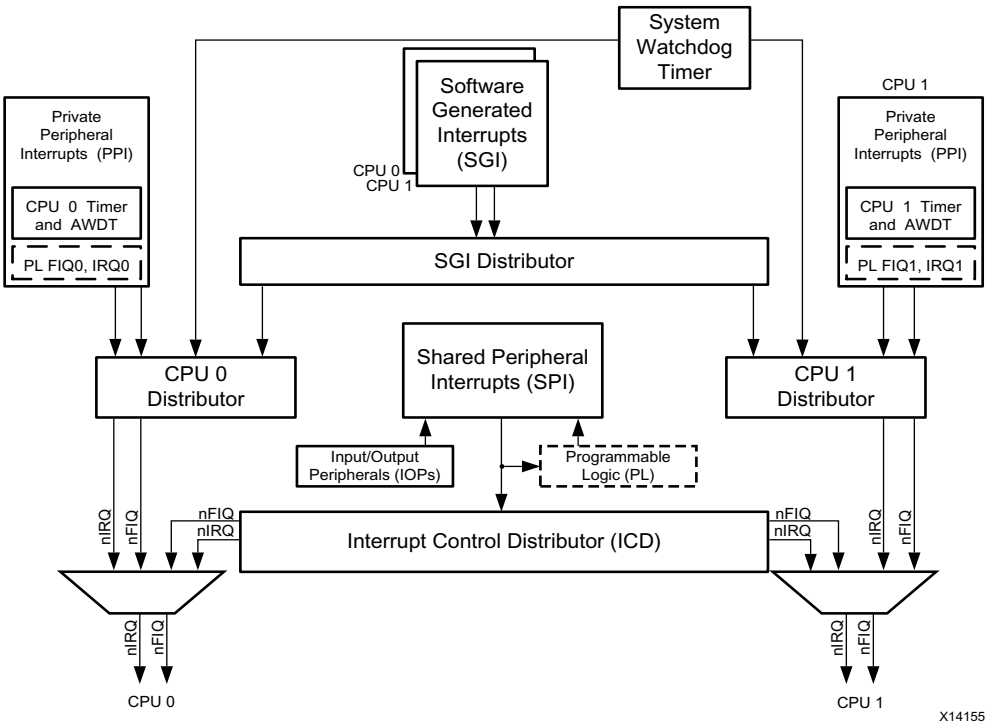

Figure 4-3: **Interrupt Handling**

Timers

Each processor core has private timers that can be used in the context of that processor core. The timers can be useful for tracking the core run time, instead of the entire system. The SoC also has global timers that give system-wide timing information. Interrupts generated by these timers are private peripheral interrupts (PPIs) and are assigned on a per-processor-core basis. Timer interrupts are often essential to system operation, because they are used for scheduling and time-slicing tasks.

MMU Configurations

Most operating systems, like Linux, require a virtual memory system. The memory management unit (MMU) is a hardware block that facilitates virtual-memory access. The MMU uses data structures, such as page tables, for address translation. The operating-system kernel must configure the MMU and set up page tables to implement the virtual-memory system. Different page tables are required for different operating-system modes, such as the secure mode of TrustZone technology.

Secure Configurations

The Zynq-7000 AP SoC provides an option of dividing a system into two worlds, secure and non-secure, by using the TrustZone technology mechanism. The processor mode must be switched to monitor mode (secure mode) to enter the secure world for performing secure operations. Also, there must be a separate MMU configuration for the different worlds.

For more information on TrustZone technology, see TrustZone Security, page 125.

OS and RTOS Choices

Operating system software manages the operations and protects the hardware resources of a computing system. An OS or RTOS can support multitasking and manage resource usage. Bare-metal software can serve as a minimal implementation of necessary system-management functions. This section covers comparisons of system-management types, hardware requirements, available OS and RTOS ports, running multiple OSs, and development and debug tools.

There are also security considerations between a monolithic OS and a microkernel. A full-featured monolithic OS like Linux uses the MMU for secure and non-secure memory regions, whereas a microkernel RTOS does not use the MMU and thus will not offer the same memory protection as a monolithic kernel. For details, refer to the Xilinx Security Solutions webpage [Ref 46].

Types of Embedded-System Software

Table 4-1 compares features of bare-metal, RTOS, and OS system-management software.

Table 4-1: Bare-Metal Software, RTOS, and OS Comparison

	Bare Metal Software	RTOS	General-Purpose OS
Capabilities	Simple	Simple to Advanced	Advanced
Hardware abstraction	None	Minimal	High-level
Multi-tasking	No	Yes	Yes
Real-time	Guaranteed response time	Guaranteed response time	As quickly as possible
Memory management	None	Basic memory protection	Advanced virtual-memory management
Code footprint	Small (KBs)	Small to Medium (KBs to MBs)	Medium to Large (MBs)
Example	Bare-metal software with SDK	VxWorks, FreeRTOS	Windows Embedded, Linux
Open-source considerations	Available with Vivado® and SDK tools. These tools require specific licenses.	Some are available with open-source licenses, some are paid versions	Some are available with open-source licenses, some are paid versions

The selection of system-management software can depend on the following factors:

- **Prior-Experience and/or Existing Solutions:** This approach to selecting system-management software saves time and validation effort. It also simplifies designing systems with variations of an existing OS or RTOS with added features. This is because of the learning curve involved in using a new OS or RTOS with a different set of supported features.
- **Capabilities:** OSs and RTOSs differ in their features and middleware support. It is important to select one based on the needs of the overall system design, such as middleware stacks for USB, TCP/IP, and file-system support.
- **Configurability:** Some RTOSs have GUI-based configuration options used to enable caches, prioritize interrupts, select the RTOS tick periods, etc. Full-featured OSs like Linux can be feature-configured using tools like `menuconfig`. These configuration interfaces and tools save time and highlight available feature options.
- **Performance:** Interrupts and task-switching latencies may vary between different OSs and RTOSs. Thoroughly investigate the hard real-time, soft real-time, and overall latencies when choosing an OS or RTOS.
- **Memory Size:** Memory-size requirements for an OS, RTOS, and application tasks need to be estimated when choosing an OS or RTOS.
- **Cost:** There are free GNU license-based OSs or RTOSs. Some vendors charge, based on their support and services model:
  - Bare-metal BSP and drivers are provided free with Xilinx development environments like the SDK.
  - RTOS selection can be based on free or paid commercial licenses, depending on the features and support model.
  - Linux selection can be based on free or paid commercial licenses, depending on the features and support model.
  - Commercial variants come with support and additional features, like USB middleware stacks or RTOS file-system stack.
- **Application Requirements:**
  - If an application needs advanced software with ready-made libraries, then consider a full-featured OS, such as Linux.
  - If an application needs hard real-time response for many tasks, then consider an RTOS like freeRTOS.
  - If an application needs both a full-featured OS and real-time response, then consider a multi-OS environment like Linux Symmetric Multi-Processing (SMP) with real-time patches, or asymmetric multiprocessing (AMP) with Linux and RTOS running on the dual ARM cores or hypervisor.

Hardware Requirements for Running an OS or RTOS

An OS or RTOS requires the following:

- **System Timer:** This is a hardware timer for the kernel heartbeat, like jiffies in the Linux OS. The timer is used by the OS or RTOS for all time-related kernel activities, such as scheduling and delays.
- **Memory Footprint:** As low as 5 to 30 KB of RAM and ROM for an RTOS. A few KBs to 100 MBs for Linux OS, based on the kernel and services configuration selections.
- **Memory Management Unit (MMU):** Required for a full-featured, general-purpose OS like Linux. May be used for memory protection by some RTOS implementations.

The Zynq-7000 AP SoC has all the hardware resources to run an RTOS and/or a full-featured OS like Linux. Achieving the time determinism of a real-time solution depends on many other factors, like resource-allocation and access times, overall solution architecture, etc., along with an RTOS.

OS and RTOS Ports

There following describes several example OS and RTOS ports to the Zynq-7000 AP SoC, but the full list is not limited to this information.

Linux OS and PetaLinux Tools

The Xilinx® Linux distribution includes the Linux OS and a complete configuration, build, and deploy environment for the Zynq-7000 AP SoC. Because PetaLinux is full-featured, integrated, tested, and documented, it is the recommended and supported Linux deployment mechanism for Xilinx customers. As a Xilinx product, PetaLinux has the same level of product management, support, bug-tracking, and emphasis that other Xilinx tools receive.

Although the Linux OS built using the PetaLinux Tools is based on a stable and thoroughly tested Linux kernel, also from the Xilinx Git server, PetaLinux offers much more than what can be downloaded from the Xilinx Git server. PetaLinux includes an installer, development tools, board support packages (BSPs), platform-management utilities, application and library frameworks, and documentation. PetaLinux is available under no-charge and commercial licenses, as described in the PetaLinux licensing page. All PetaLinux users are entitled to community (forum-based) support; commercial licensees may also access support directly from Xilinx. For details, see the Xilinx Linux wiki page [Ref 54] and the Xilinx PetaLinux wiki page [Ref 56].

Xilinx's PetaLinux Tools supports configuring, customizing, building, and packaging a Linux BSP for full-featured OS functionality on the Zynq-7000 AP SoC. For details on how to configure and use these BSPs, refer to the Xilinx PetaLinux Tools web site [Ref 48].

OS and RTOS Ecosystem

In addition to the Xilinx OS and RTOS ports, described above, the Zynq-7000 AP SoC has strong ecosystem support products from 3rd-party and alliance members. These products may have differences in supported features, context-switch performance, interrupt latencies, technical support, and in their middleware support for file systems, USB, PCIe, Ethernet, etc.

Table 4-2 and the following sections describe the 3rd-party and alliance member ecosystem solutions. For additional information, refer to the Xilinx Zynq-7000 AP SoC Ecosystem webpage [Ref 47].

Table 4-2: OS, RTOS, and Middleware Products

Alliance Member	OS, RTOS, and Middleware Products	More Information
Adeneo Embedded	Windows Embedded Compact 7, Linux, Android, and QNX	[Ref 84]
Discretix	Security-centric software and IP solutions	[Ref 77]
ENEA Software AB	OSE RTOS and ENEA Linux	[Ref 78]
eSOL	uITRON 4.0 RTOS, T-Kernel RTOS and IDE	[Ref 79]
Green Hills Software	INTEGRITY RTOS	[Ref 81]
Express Logic	ThreadX RTOS	[Ref 80]
iVeia	Android On Zynq-7000 AP SoC	[Ref 84]
Mentor Graphics	Nucleus RTOS	[Ref 85]
Micrium	µC/OS RTOS	[Ref 86]
MontaVista Software	MontaVista Carrier Grade Linux	[Ref 87]
Open Kernel Labs	OKL4 Microvisor	[Ref 90]
QNX	QNX Neutrino RTOS	[Ref 91]
Quadros	RTXC RTOS For information about features, see Quadros Systems Features, page 146	[Ref 92]
Real Time Engineers Ltd.	FreeRTOS	[Ref 93]
Sciopta	Sciopta RTOS	[Ref 94]
Sierraware	Open source SierraVisor Hypervisor and SierraTEE Trusted Execution Environment	[Ref 95]
SYSGO	PikeOS	[Ref 96] [Ref 97]
Timesys	LinuxLink	[Ref 99]
Wind River	VxWorks, Linux, and Workbench IDE	[Ref 100]

Quadros Systems Features

RTXC RTOS, from Quadros Systems, has these features:

- Small, scalable footprint to fit any ROM or RAM budget (typically 25 KB or less)
- Delivered in source code, with sample code for a working project using ARM DS-5 development tools
- Graphical configuration program, RTXCgen, for ease of building the application's RTOS
- Graphical trace tool, RTXCview, for visibility into system performance and behavior
- RTXC/mp available for asymmetric multiprocessing (AMP)
- License fees apply

The following middleware stacks are supported along with RTXC RTOS:

- Ethernet TCP/IP v4 and v6
- Ethernet packet prioritization
- Ethernet security suite
- Application server:
  - AJAX, JSON, and XML-RPC
  - Remote device management, with persistent sockets (WebSockets)
  - Remote GUI
- USB host and device
- High performance, fail-safe file systems:
  - NAND and NOR
  - SD, MMC, and flash drive

Support for Zynq-7000 AP SoC and ZC702 development boards:

- Timer initialization
- UART initialization
- MMU initialization
- Interrupt controller setup and initialization

- Instruction- and data-cache initialization.
- Advanced support for ARM Vector Floating Point (VFP) and NEON:
  - Explicit or automatic enabling of VFP-register context extensions for tasks that use VFP or NEON.
  - Optional support for a VFP and NEON register set that reduces context-switch overhead.
  - Uses "lazy swapping" to minimize interrupt latency and context-switch overhead for VFP and NEON register sets.
  - Control of VFP context, such as manual suspend and automatic or manual resume. Fine-level control allows application to minimize VFP-register context overhead in non-VFP and non-NEON execution paths. Useful in mostly integer environments in which tasks rarely need VFP or NEON, except in isolated functions. When used with lazy swapping, explicit suspend control can be used to improve overall system throughput.

Real Time Engineers Ltd.

FreeRTOS is free to use without any need to expose custom code. FreeRTOS supports real-time tasks, queues, binary semaphores, counting semaphores, mutexes, etc., for communication and synchronization between tasks and interrupt-service routines. The FreeRTOS memory footprint is about 4 to 9KB. The middleware stacks supported with freeRTOS include:

- FAT file system
- UDP/IP and TCP/IP stacks
- CLI
- Safety
- CyaSSL SSL/TSL

Multi-OS

There are multiple ways to enable multiple operating systems running on both CPU cores of the dual-core ARM Cortex-A9 MPCore processor in the Zynq-7000 AP SoC. See the following links for information about the symmetric and asymmetric multi-processing options:

- *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7]
- Xilinx Multi-OS Support wiki page [Ref 55]

Simple AMP

A simple multi-OS AMP implementation can consist of running, for example, Linux OS and a bare-metal application or of running two bare-metal applications.

The two CPU cores of the ARM processor share memory and peripherals. Asymmetric multi-processing (AMP) is a mechanism that allows both cores to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources. Figure 4-4 shows an example of AMP with Linux OS and a bare-metal application. Figure 4-5, page 149 shows an example of AMP with Linux OS and an RTOS.

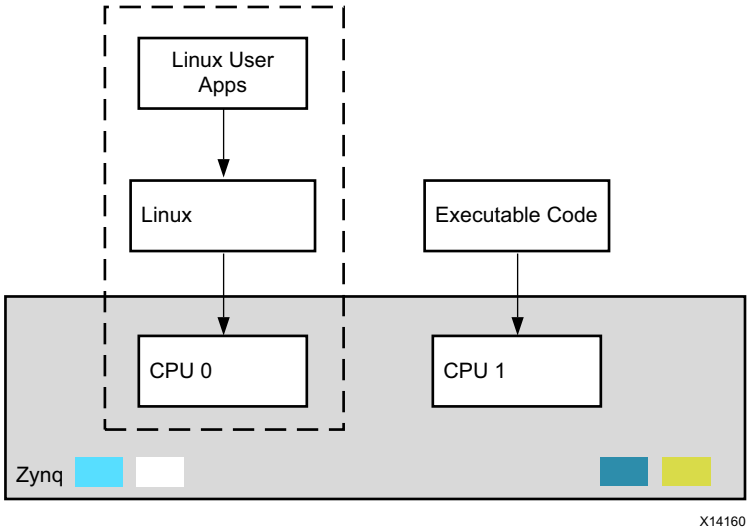

Figure 4-4: **AMP with Linux OS and Bare-Metal Application**

Referring to Figure 4-4, AMP with Linux OS and a bare-metal application operates as follows:

- Linux runs on CPU0.
  - Linux then starts the CPU1 executable loaded into memory by the FSBL.
- The bare-metal application runs in dedicated memory space on CPU1.
  - MMU0 is used by Linux as normal. MMU1 is used to define the bare-metal application's memory context, but no special coding is required.
  - The bare-metal application does not run within the Linux memory context.
  - Linux is not aware of the memory used by CPU1.

The MMU can be used to contain the application.

  - The MMU defines what addresses (memory and AXI devices) the application can access normally.
  - CPU1 is not restricted from accessing the Linux memory spaces or shared devices (ICD or SCU), but Linux can detect if and when that occurs and take the appropriate action.

- Implementation approach.
  - Communication between Linux and the bare-metal application use the OCM.
  - OCM caching is disabled for improved determinism.
- The use case is supported by Xilinx Worldwide Technical Support department.

For more information, refer to:

- *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors* (XAPP1079) [Ref 37]
- *Simple AMP Running Linux and Bare-Metal System on Both Zynq-7000 AP SoC Processors* (XAPP1078) [Ref 36]

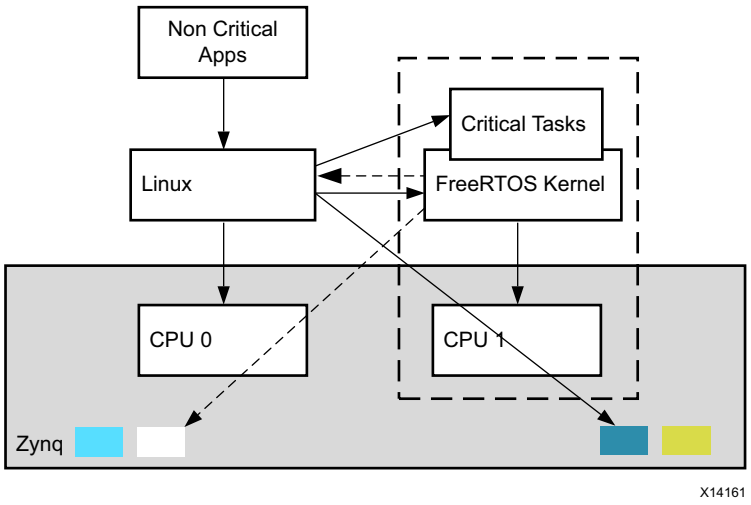

Figure 4-5: **AMP with Linux OS and RTOS**

Referring to Figure 4-5, AMP with Linux OS and RTOS operates as follows:

- The firmware runs in dedicated memory space at address 0x0000_0000.
  - The FreeRTOS kernel does not run within the Linux memory context.
  - Linux can read and write this address space just like it can read any other device memory.

- The MMU limits how firmware affects Linux.
  - The MMU defines what addresses (memory and AXI devices) the application can access normally.
  - The granularity is 1 MB. If two devices lie in a 1 MB region you cannot isolate one without the other.
  - The GIC is shared between firmware and Linux.
    - Interrupts are routed to firmware or Linux by the GIC.
    - Firmware is not prevented from reconfiguring the GIC.
- The Linux kernel is not prevented from corrupting firmware.
  - However, violations should only occur from explicit calls or through specific types of crashes.
  - System design and comprehensive testing may mitigate customer real-world concerns.
- Implementation approach.
  - Should be applicable to other flat RTOS.
  - Might be applicable to RTOS using virtual memory (such as kernel and user space).

For an overview of the Linux OS and FreeRTOS AMP design shown in Figure 4-5, page 149, refer to *PetaLinux Tools User Guide: Zynq All Programmable SoC Linux-FreeRTOS AMP Guide* (UG978) [Ref 17].

ARM Cortex-A9 TrustZone

The Zynq-7000 AP SoC supports the ARM Cortex-A9 TrustZone technology, according to the recommendations of the Trusted Base System Architecture specification. This technology enables the development of platforms that support a Trusted Execution Environment (TEE) and security-aware applications and secure services, or Trusted Applications (TA). A TEE is a small, secure kernel that is typically developed with standard APIs according to the TEE specification evolved by the Global Platform industry forum.

The TrustZone technology supports development of separate Rich Operating System and TEE environments by creating operating modes, in addition to the Normal domain, that are known as the Monitor mode and the Secure domain. The Secure domain has the same capability as the Normal domain but it operates in a separate memory space. A Secure Monitor acts as a virtual gatekeeper to control migration between the domains.

Development and Debugging Tools

The development and debugging tools available to support your application are described below.

Xilinx Software Development Kit

The Xilinx Software Development Kit (SDK) provides a variety of Xilinx software packages, including drivers, libraries, board-support packages, and complete operating systems, for developing a software platform.

TIP: *The Software Development Kit Design Hub in the Documentation Navigator provides links to additional information about Using the Software Development Kit and embedded design. For more information, see* *Related Design Hubs, page 225*.

The Xilinx SDK is a complete development environment, including:

- Platform-aware BSP generation
- Sample application-code generation, ranging from memory tests to TCP/IP echo servers
- Cross-compilation tools
- Application profiling
- Linker-script generation
- Debug interfaces through JTAG
- Target communication framework
- Remote debugging
- Heterogeneous multicore debugging
- Flash loaders
- Boot-file creation for secure and non-secure environments
- First-stage boot loader (FSBL) generation

SDK supports Linux-kernel debugging through JTAG and application thread-level debugging using the target communications framework (TCF) agent and GNU GDB infrastructure.

ARM DS-5 Development Studio

ARM DS-5 Development Studio is a complete development environment that supports Linux, Android, and RTOS debugging. For details on using the DS-5 with a Zynq-7000 AP SoC, refer to the *Xilinx XC702 DS-5 Getting Connected Guide* [Ref 101] and the *Zynq-7000 Platform Software Development Using the ARM DS-5 Toolchain* (XAPP1185) [Ref 42].

IAR

The IAR Systems' debugger supports OS and RTOS kernel-level debugging. For more information, see the IAR Integrated Solutions Partner Program web site [Ref 83].

Libraries and Middleware

This section summarizes how static and dynamic libraries and middleware can be used to improve application-programming productivity and reliability.

Libraries

Libraries are a convenient method of combining common and useful functions into a single unit that can be re-used by programs. A library abstracts implementation details and provides a tested and optimized behavior that can be easily incorporated into programs. Libraries support static and dynamic linking.

Libraries can be categorized into commonly used libraries and domain-specific libraries:

- Examples of commonly used libraries include GNU C, GNU C++ library, POSIX Pthread libraries.
- Examples of domain-specific libraries include Intel Threading Building Blocks (parallel processing), FFmpeg audio, OpenCV, direct rendering manager (DRM), kernel mode setting (KMS), ARM OpenMAX DL sample software library (audio/video processing), and MATLAB engine library (mathematical processing).

Benefits of Libraries

- Provide behavior and code reusability (the write-once, use-many approach).
- Speed up application development by abstracting underlying hardware and software.
- Provide easily maintainable code that reduces testing resources.

For example, the GNU C library is well-tested, and bugs are fixed regularly. Because of this, using the library saves substantial testing and development resources.

Xilinx Libraries for Stand-Alone Systems

The Xilinx Software Development Kit (SDK) provides a set of reusable libraries and drivers for Zynq-7000 AP SoC processing system (PS) and programmable logic (PL) components. These libraries and drivers include:

- libxil.a — Device drivers for peripherals.
- LibXil MFS — A memory file system.
- LibXil FFS — A generic FAT file system based on an open source implementation. It is primarily used with the SD/eMMC driver and a glue layer is implemented to link it to that driver.
- LibXil Flash — A library that provides read, write, erase, lock, unlock, and device-specific functions for parallel flash devices.
- LibXil Isf — An in-system flash library that supports the Xilinx in-system flash hardware and serial flash on SPI/QSPI.
- LibXil SKey — The LibXil SKey library provides a programming mechanism for user-defined eFUSE bits. The PS eFUSE holds the RSA primary key hash bits and user feature bits, which can enable or disable some Zynq-7000 AP SoC processor features.
- lwIP — A third-party, light-weight TCP/IP networking library.

Libraries are classified into two types, static and dynamic, according to the type of linkage object.

Static Libraries

Static libraries are a collection of object files that become, after linking, part of an application image. In Linux, these libraries are called archives. The file names of static libraries conventionally end with a `.a` suffix.

Advantages

- Simple to use; external dependencies are resolved at compile time.
- Shorter compilation time.
- Runs slightly faster. In theory, code in static ELF libraries that is linked into an executable should run 1% to 5% faster than a shared library or a dynamically loaded library.
- Simplifies makefile link steps, because the linker takes care of searching and extracting object files required by the application.

Disadvantages

- Changes to static libraries require rebuilding all dependent applications.
- Executable memory footprint becomes larger.
- If processes that share an object library run simultaneously, the code and data sections of the object library are copied into the address space of each process.

When to Use

Static linking is often the easiest way to distribute an application because it relieves the user from the task of distributing any dependent library with the target application.

Dynamic Libraries

Dynamic libraries are loaded when an application that uses them is executed. A single copy of the object library is shared among multiple applications, so that the library's code and data sections do not become part of application's address space. Shared objects are not copied into each application program area. Instead, a single copy of the object library is loaded into memory at run time when the first application requiring that library is executed.

In Linux, the naming convention for dynamic-library files has a `lib` prefix, the name of the library, and the string `.so` followed by a version number that is incremented whenever the interface or implementation changes.

Advantages

- Smaller executable memory footprint than for static libraries.
- If an object library is changed, dependent applications simply need to be re-executed.

Disadvantages

- Libraries are dependent on the run time loader. In Linux, the dynamic loader (ld.so) is part of the GNU C library and is responsible for loading all dependent shared libraries.
- Symbol relocation is performed at run time. Because of this, applications using a shared dynamic library may take slightly more time to execute than a static-library counterpart.

Running an executable on a target platform requires that the custom shared library path be added to the search path of the dynamic linker. The LD_LIBRARY_PATH environment variable specifies to the run time linker that the custom shared library resides at some particular location.

When to Use

Use a dynamic library when the library is shared between multiple applications. For example, use embedded run time libraries like the GNU C library in this manner.

One such use case can be found in a Xilinx Zynq-7000 AP SoC Tech Tip demonstrating use of the ARM NEON Library on a ZC702 platform. Refer to *Xilinx Zynq-7000 AP SoC Spectrum Analyzer part 2-Building ARM NEON Library Tech Tip* [Ref 62].

The Tech Tip describes the process of obtaining and building a set of filtering functions targeting the Zynq-7000 AP SoC ZC702 platform. Many applications that can take advantage of the processing capabilities of the Zynq-7000 AP SoC involve complex calculations used in filtering, video manipulation, and signal processing.

An open source project within the ARM community provides a library of common, useful functions accelerated by NEON that you can use when developing applications. This has evolved into the Ne10 project and the Ne10 library.

Middleware

Middleware is a set of libraries that runs between the operating-system and application layers. In embedded systems, middleware is system software that sits on top of the operating system or is sometimes included as part of the operating system. For example, common middleware such as a TCP/IP communication stack is typically part of modern operating systems.

Middleware reduces complexity by centralizing software stacks that otherwise would be redundant in an application layer. There are many type of middleware, including message-oriented middleware (MOM), remote procedure call (RPC), and networking protocols above the device-driver layer and below the application layer.

Examples

- TCP/IP stacks
- USB host stack
- Controller area network (CAN) stack
- Multimedia middleware stack

Advantages

- Fast development of robust and powerful applications
- Does only what is unique to the application
- Supports scalability and abstraction
- Reliability

Use Case

lwIP is an open source TCP/IP protocol suite available under the BSD license. Although it can be used with an operating system, lwIP is a stand-alone stack without operating system dependencies. The use-case demonstrates how you can integrate middleware, such as a lightweight IP (lwIP) network stack, into a stand-alone application.

The Xilinx software development kit (SDK) provides lwIP software customization. The lwIP stack supports IP, ICMP, IGMP, UDP, TCP, and other networking protocols. By using the lwIP stack, you can focus on developing your application core and leave the TCP/IP implementation to the lwIP stack.

For more information about how to integrate an open-source network stack into an application development, refer to *LightWeight IP (lwIP) Application Examples v4.0* (XAPP1026) [Ref 35].

Boot Loaders

A boot loader is the software that initializes the system in preparation for execution of the next level of software, such as an operating system. Usually, each operating system has a set of boot loaders specific for it. Boot loaders usually contain several ways to boot the OS kernel and also contain commands for debugging and/or modifying the kernel environment.

Details of the boot process depend on the complete system design and choice of OS, such as embedded Linux, RTOS, or bare-metal OS. The number of stages in booting also may vary based on the system design. This section describes the boot flow of the Zynq-7000 AP SoC, from power-up to application execution.

Boot Process

The Zynq-7000 AP SoC boot process can be a multistage boot, depending on the system requirements. This section describes three broadly divided stages of booting and their role in the booting process. Figure 4-6, page 157 shows the steps while booting embedded Linux and application code on a Zynq-7000 AP SoC.

When power is applied to the SoC, the boot process starts from the BootROM. This process loads and then starts executing the first-stage boot loader (FSBL) from on-chip memory (OCM). The FSBL configures the specific initialization. Then, based on the software architecture, the second-stage boot loader (SSBL), such as U-Boot in the case of embedded Linux, is initialized and executed. The FSBL and/or SSBL start the RTOS or embedded Linux and the application code.

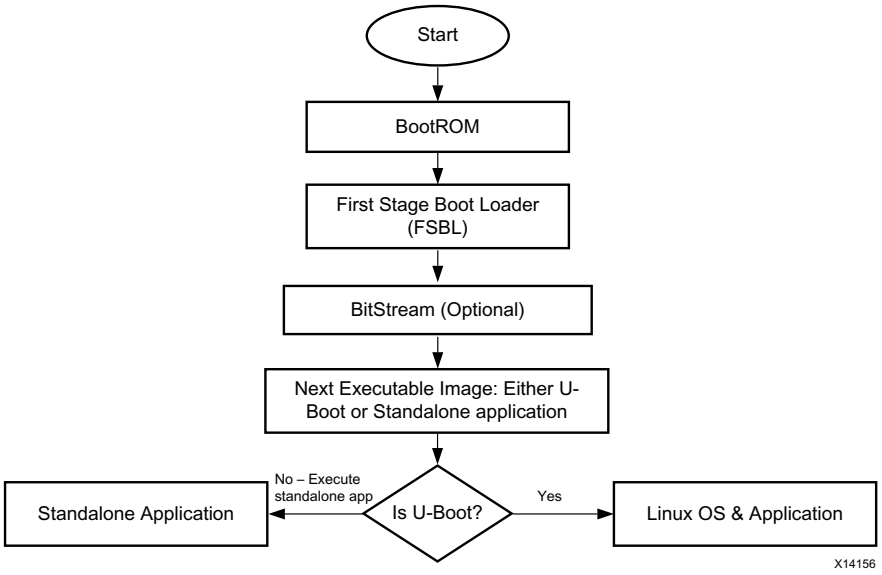

Figure 4-6: **Boot Flow**

Stage 0

At power-on or reset, the ARM core runs initialization code from the BootROM. The BootROM code cannot be changed; it is a factory pre-programmed code that comes with each Zynq-7000 AP SoC. The BootROM code determines the device on which the next-level loader is located by reading the boot-mode pins. Depending on the boot-mode setting, the FSBL is copied from either NAND, parallel NOR, serial NOR (Quad-SPI), or Secure Digital (SD) flash memories to the OCM.

Figure 4-7 shows the standard (non-secure) boot flow. The secure boot flow is described in Embedded Device Security, page 45

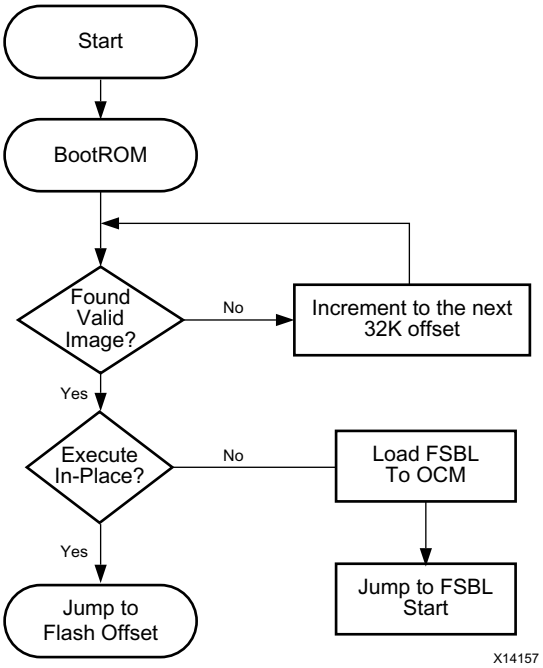

Figure 4-7: **Non-Secure Boot flow**

Stage 1

The FSBL gets execution control from the BootROM and either runs from the OCM or execute-in-place flash based on the boot mode settings. The FSBL initializes the PS and looks for a bit file in the boot device. If found, the FSBL writes the bit file to the PL. Whether or not a bit file is found, the FSBL loads application binaries and data files into memory until the complete image has been read from the boot device. Then the FSBL starts executing the first application binary that was loaded.

Typically, the FSBL initializes the external RAM and loads the second-stage boot loader (SSBL) or a stand-alone application. Based on the software architecture, if it is a full featured OS like embedded Linux the SSBL (such as U-Boot in the case of embedded Linux) is initialized and executed. The FSBL and/or SSBL start the RTOS or embedded Linux and the application. Then, the FSBL hands over control by jumping to the start address of the SSBL, in the case of a full-featured OS like embedded Linux, or to an RTOS or application.

Stage 1 can include the user application if it is small enough to fit in OCM memory.

The Zynq-7000 AP SoC supports both secure and non-secure boot methods. The boot-ROM code flow for both methods is described below.

For secure boot, the principal security objective of the SoC is to lay a foundation of trust. It does this using integrity, confidentiality, and authentication from the time power is applied to the time control is asserted. The SoC also provides a method for maintaining this trust. The SoC does this by using the PL's built-in Advanced Encryption Standard (AES-256) and hashed message authentication code (HMAC) engines, and the RSA authentication capability of the PS.

The SoC boots securely unless it is changed from a secure to a non-secure mode. The first decision point is the initial load of the FSBL. If the FSBL is encrypted, the BootROM code boots securely and hands control off to the FSBL securely, while disabling JTAG. If the FSBL is not encrypted, the AES 256 and HMAC engines are disabled and the JTAG ports can be accessed. To enable RSA authentication, the RSA Enable eFUSE must be programmed.

The Xilinx design tools allow specifying whether the software and bit-stream partitions are authenticated using the public-key RSA algorithm and whether subsequent images are encrypted and authenticated using the AES and HMAC engines. An unencrypted partition can also be specified. This allows use of both public- and private-key algorithms on a partition basis.

Alternatively, when configuration time is critical, a trade off can be made between security and boot time, because configuration speed is faster for unencrypted partitions than for encrypted partitions. A relatively large, open-source U-Boot or Linux image, for example, loads faster if the image is unencrypted. If any of the secure features are selected, the initial FSBL, at a minimum, must be encrypted with AES-256 and authenticated with the HMAC algorithm.

Figure 4-8, page 160 shows the FSBL flow for secure boot mode. Figure 4-9, page 161 shows the FSBL flow for non-secure boot mode.

For more information about secure boot of the Zynq-7000 AP SoC, refer to the following documents:

- *Secure Boot in the Zynq-7000 All Programmable SoC* (WP426) [Ref 33]
- *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 40]

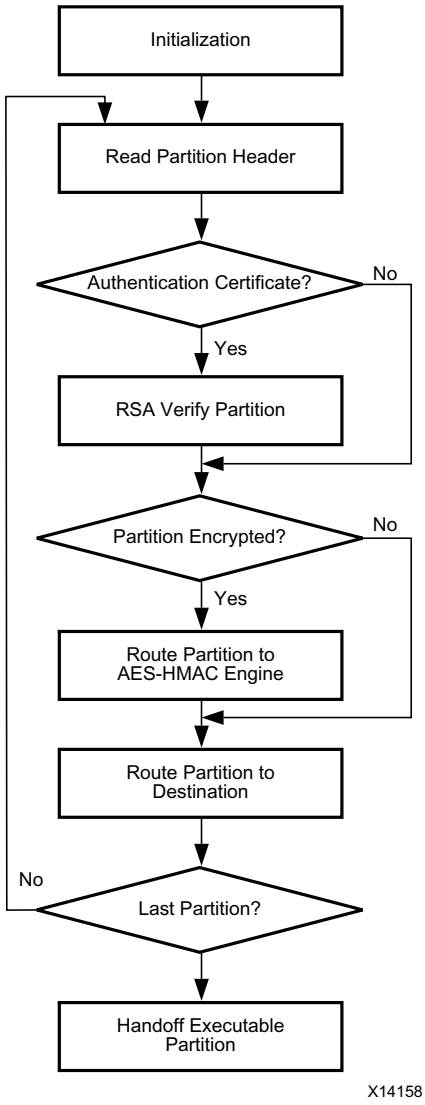

Figure 4-8: **FSBL Flow for Secure Boot**

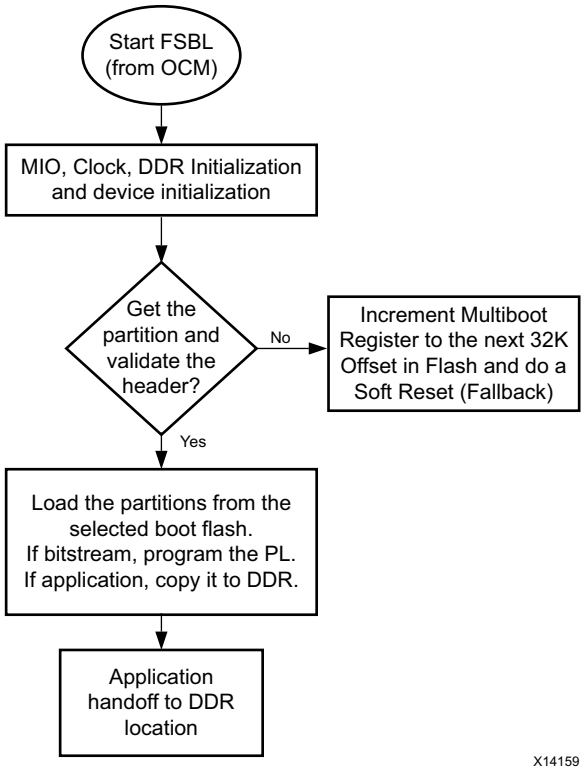

Figure 4-9: **FSBL Flow for Non-Secure Boot**

Stage 2

This stage could be a second-stage boot loader (SSBL) like U-Boot, or it could be an RTOS or application. In the case of a full-featured OS like embedded Linux, the SSBL U-Boot runs in CPU0 to initialize and set up the environment in which the OS will boot. The initialization includes configuring the MMU to use flat memory (that is, virtual and physical locations are the same). The boot loader then fetches the kernel image (per the boot-mode setting) and other information, such as boot arguments, into RAM. Then the loader passes control to the OS, such as embedded Linux.

In the case of Linux, the OS detects and enables the second processor core, configures and activates the MMU and data caches, and performs other actions to make a complete system available to applications.

There is no limit to what each stage can actually do, although some things can be done more easily in one stage than in others. For example, reading a file from an NTFS USB drive is possible from Linux, but it requires some development effort to do it from the boot loader. U-Boot can be adapted to such requirements.

Zynq-7000 All Programmable SoC Software Developers Guide (UG821) [Ref 7] covers the architecture details, programming model, OS considerations, and complete development steps for running Linux and applications on Zynq-7000 AP SoC.

Software Development Tools

Xilinx provides a variety of software development tools that can be used to build various software components used by the Zynq-7000 AP SoC, including:

- **Board Support Package (BSP).** This is a set of APIs used to access low-level hardware. These APIs are grouped based on the peripheral they access and functions they perform. The BSP also includes startup, CPU initialization, and EABI code.
- **Stand-alone (bare-metal) applications.** These are simple applications that do not support complex kernel features such as multi-tasking. Such applications use the BSP APIs to access low-level hardware.
- **FSBL (first stage boot loader).** This is a small application that performs PS hardware initialization, loads the fabric with a bit stream, optionally loads additional data, and loads the second-stage boot loader. The FSBL is an example of a stand-alone (bare-metal) application.
- **U-Boot (second stage boot loader).** This boot loader performs the necessary hardware initialization for the kernel to begin execution. When done, the Linux kernel is loaded and starts executing.
- **Linux kernel.** Linux is an open source operating system. The hardware-dependent parts of the Linux kernel (device drivers, etc.) are provided for the Zynq-7000 AP SoC. The hardware independent parts of the kernel (file systems, networking, etc.) are similar to other Linux machines. The kernel is configured as needed and targeted to the ARM processor.
- **User applications.** Typical user applications run in the Linux environment and are hardware independent. Hardware access is done through the driver APIs.
- **Device tree blob.** The device tree is a structure of nodes and properties describing components and features supported by the hardware (PS peripherals, ARM processors, and PL peripherals). The nodes provide information such as the hardware-component map address, interrupt number, default settings, etc. The device tree is compiled and provided to the Linux kernel at boot time. While booting, the kernel uses this information to load the corresponding device drivers.

Also, the software development tools are used to selectively debug and profile the software on the hardware platform.

The software development tools for Zynq-7000 AP SoCs include:

- **GUI-based tool chain.** These tools provide you with a graphical user interface. The tool and build features are selected through a set of menu options. After this is done, the compiler/linker is started in the background and the compilation progress is shown visually. The GNU tools are integrated into the GUI-based tool chain applications. The Xilinx software development kit (SDK) is an example of a GUI-based tool chain.

  With the IDE (integrated development environment), programmers can build a variety of software components, such as:

  - BSP
  - User applications on a stand-alone platform
  - User applications on a Linux platform

- **Command-line based tool chain.** These tools are standard GNU tools (such as gcc) that are configured for the ARM processor. CodeSourcery (a gcc toolchain from Mentor Graphics supporting the ARM architecture) is used in the development tools for the Zynq-7000 AP SoC. The compiler and linker options are supplied using the command line or with makefiles. The arm-linux-xilinx-gnueabi-gcc is an example of a command-line based tool chain.

  With the GNU-based tool chain, programmers can build a variety of software components, such as:

  - The Linux kernel and device drivers
  - Device tree blob
  - User applications
  - Reusable libraries

- **Hybrid tool chain.** These tools are a combination of GUI and command-line based tools. The configuration of all project components is done through GUI-based tools. However, each component build is done using command-line tools. PetaLinux is an example of a hybrid tool chain.

  The GNU tools that are configured for ARM processors are integrated into a hybrid tool chain. With a hybrid tool chain, programmers can build a variety of software components, such as:

  - The Linux kernel and device drivers
  - Device tree blob
  - User applications
  - Reusable libraries
  - Completely packaged image

GUI-Based Tool Chain

The Xilinx SDK is an integrated development environment (IDE) based on Eclipse. The Xilinx SDK provides an environment for creating, compiling, and debugging software applications.

The SDK runs the GNU-based compiler tool chain in the background. The supported utilities include the GCC compiler, a linker, the GDB debugger, and other utilities such as a JTAG debugger, flash programmer, and libraries. Also, the tool integrates drivers for all Xilinx IP cores, example applications, software service packages (such as lwIP), and bare-metal BSPs. Programmers can integrate any of these components to build the application. The SDK supports applications developed using assembly, C, and C++.

Features of the Xilinx SDK include:

- Support for Zynq-7000 AP SoCs and MicroBlaze™ processors.
- Based on the Eclipse C/C++ Development Tooling (CDT).
- Complete IDE that directly interfaces to Vivado tools.
- Complete software design and debug processes are supported including multicore, hardware, and software debug capabilities.
- The editor, compilers, build tools, flash memory management, JTAG, and GDB debuggers are integrated.
- Supported by the Xilinx edition of Mentor Sourcery CodeBench Lite.
- Custom libraries and device drivers.
- The SDK includes user-customizable drivers for all supported Xilinx hardware IP cores, a POSIX-compliant kernel library, and networking and file-handling libraries. The libraries and drivers can be customized based on feature needs, memory requirements, and hardware capabilities.
- Managed makefiles, management of hardware platforms, remote debug, cross probing support between hardware and software debug, flash programming, boot image (boot.bin) creation, linker script generation, repository management, and FPGA programming.
- Profiling support.

Features of the Xilinx SDK debugger include:

- Based on the Eclipse Target Communication Framework (TCF).
- Homogenous and heterogeneous multi-processor support.
- Linux application debug on the target platform.
- Hierarchical profiling.
- Bare-metal and Linux application development.

- Support for symmetric multi-processing and asymmetric multi-processing designs.
- Associate hardware and software breakpoints per core.
- NEON library support.

Applications are developed based on either a stand-alone or a Linux platform model.

Stand-Alone Platform

In a stand-alone platform, an application is built as monolithic, executable code. Multi-tasking is not supported, so device drivers are loaded with the application and the application invokes the driver APIs directly.

The stand-alone software platform is a single-threaded environment used when an application accesses processor functions directly. The stand-alone software platform supports program profiling and provides functions such as processor interrupt handling, exception handling, and cache handling.

Stand-alone platforms are used by small, dedicated systems that do not require an operating system. The advantages include:

- Application development is typically simple and quick.
- The overall complexity is minimized because context switching and multi-tasking is not used.
- Developers can have a more detailed and comprehensive understanding of the top-to-bottom system implementation.
- Ideal for the initial development stages of new hardware.
- A console (text) based application can be built using this platform.

The disadvantages include:

- A complex application system requiring multi-tasking cannot be built.
- Not ideal for GUI-based applications.

The steps for developing a stand-alone application include:

- Creating a new workspace and importing the hardware platform into the workspace (the hardware platform is exported from the Vivado design tools).
- Creating a new board support package (BSP) and application.
- Alternatively, importing an existing BSP and software application into the workspace. A board support package (BSP) in the SDK contains libraries and drivers that software applications can use when the provided APIs are used.
- Modifying the BSP settings.

- Building the BSP and application.
- Debugging software applications.
- Profiling software applications.

Figure 4-10 shows the development cycle using the SDK tool.

More details on SDK usage can be found in the following documents:

- *K7 Embedded TRD 2013.2* [Ref 49]
- *OS and Libraries Document Collection* (UG643) [Ref 5]

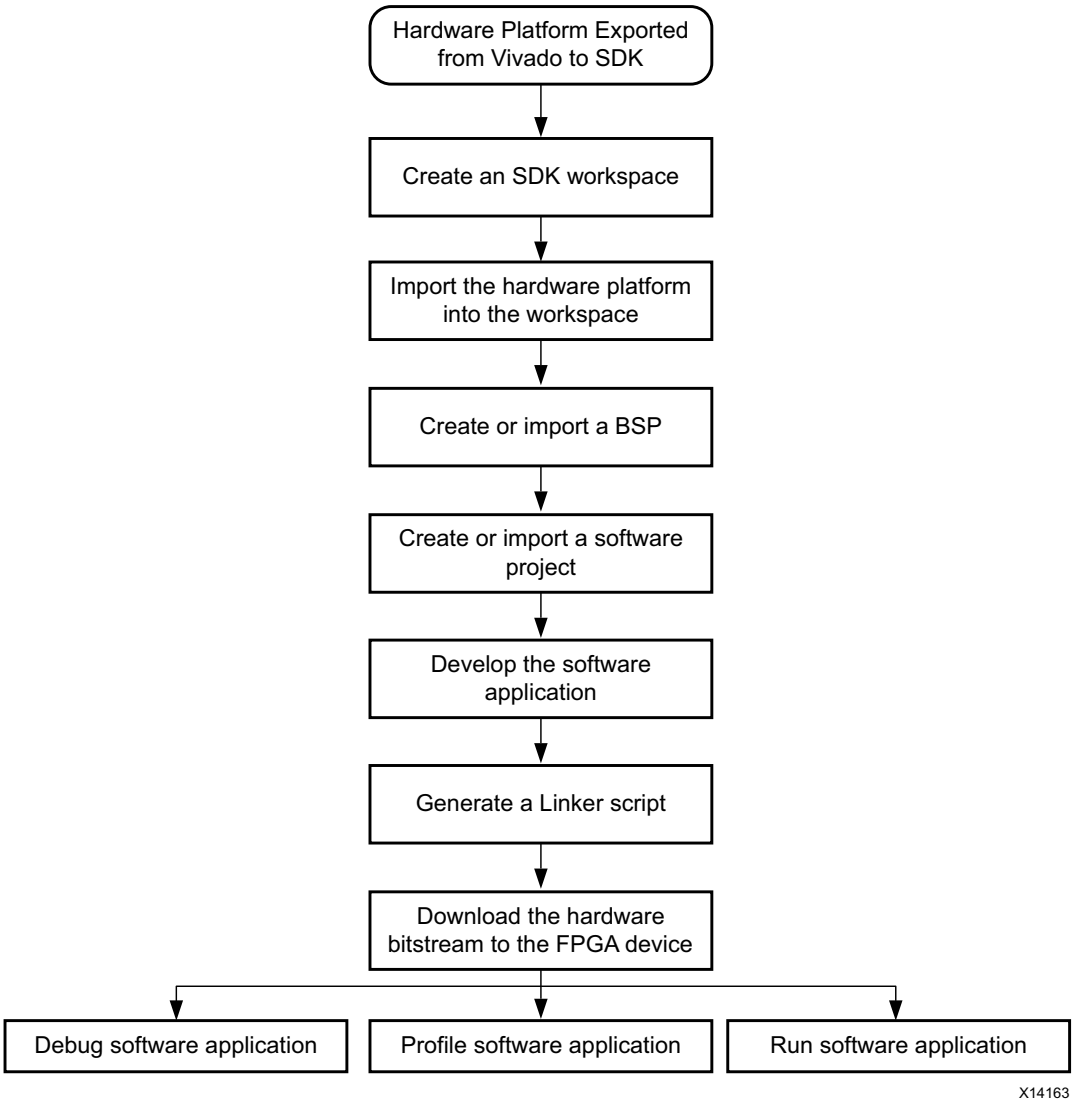

Figure 4-10: **SDK Software Development Flow**

Linux Platform

Linux is a multi-process operating system, and applications written for a Linux platform are code compatible with any system that runs Linux. The applications use the Linux library APIs and therefore access the hardware resources in an abstract manner.

Application development for a Linux platform is similar to other Linux application development processes that are hardware-platform independent. The procedure for creating applications in the SDK workspace is similar to the stand-alone platform except the Linux-platform BSP is selected.

GNU-Based Compiler Tool Chain

The Zynq-7000 AP SoC supports launch of the GNU tool chain in two ways:

- GNU open-source tool chain
- PetaLinux framework

These variants are described in the following sections.

GNU Open-Source Tool Chain

The open-source tool chain is based on Mentor Graphics' CodeSourcery tool chain. Sourcery CodeBench is a complete development environment for embedded C/C++ development on ARM, Power, ColdFire, and other architectures. Sourcery CodeBench Lite Edition includes:

- GNU C and C++ compilers
- GNU assembler and linker
- C and C++ runtime libraries
- GNU debugger

For ARM processors, the tool chain supports the ARM, Thumb, and Thumb-2 instructions for the active architectures, including version 7 of the ARM architecture.

The GNU open source tool chain is described in more detail on the Xilinx Zynq-7000 AP SoC Linux wiki page [Ref 63].

PetaLinux Framework

The PetaLinux Tools is a development environment that works with the Xilinx hardware-design flow for Zynq-7000 AP SoCs. Tailored to accelerate design productivity, the solution contains everything necessary to build, develop, test and deploy embedded Linux systems.

PetaLinux consists of three key elements:

- Pre-configured binary bootable images.
- Fully customizable Linux operating system for Xilinx devices.
- PetaLinux Tools, including tools and utilities to automate complex tasks across configuration, build, and deployment.

Because PetaLinux is fully featured, integrated, tested, and documented, it is the Xilinx recommended and supported Linux deployment mechanism for Xilinx customers. PetaLinux BSPs provide a complete, integrated and tested, Linux operating system for Xilinx devices including:

- BSP
- Boot loader
- Linux kernel
- Linux applications & libraries
- C & C++ application development
- Debug
- Thread and FPU support
- Integrated web server for easy remote management of network and firmware configurations

Although PetaLinux is based on a stable and thoroughly tested version of Linux, the PetaLinux Tools includes more than what is available from the Xilinx Git server. The PetaLinux Tools includes an installer, development tools, BSPs, platform management utilities, application and library frameworks, and documentation that are not found in the Xilinx open source Linux (OSL) offering.

All PetaLinux users are entitled to community (forum-based) support. Commercial licensees can get support directly from Xilinx.

The PetaLinux tool chain is described in more detail on the Xilinx PetaLinux Tools web site [Ref 48].

JTAG Debugger

The GUI-based SDK includes debugging capabilities, enabling programmers to debug code in the following ways:

- Single step
- Break points
- Memory watch
- Disassembly
- Call-stack
- Processor register dump

The SDK provides various ways to use the debug capabilities. The system debugger is based on the Vivado hardware server (which is based on TCF). The system debugger performs debug operations over JTAG or a TCP/IP link. The JTAG link is used for local debug and the TCP/IP link is used for communicating with the TCF agent in Linux. The debugger is also available as a command line debugger called XSDB. XSDB can be used without invoking the SDK.

The GUI-based SDK debugger is described in more detail in *K7 Embedded TRD 2013.2* [Ref 49]

The command-line XMD debugger is described in more detail in the *Embedded System Tools Reference Manual* (UG1043) [Ref 21].

JTAG Profiler

The GUI-based SDK also includes a software profiler. The profiler enables programmers to monitor function calls, time spent by the processor on each function, a graphical display of processor usage, etc. As with the debug capabilities, the profiler is based on the TCF-based XMD tool, which performs the profiling operations over the JTAG link.

Refer to the *AXI Interface Based KC705 Embedded Kit MicroBlaze Processor Subsystem Software Tutorial* (UG915) [Ref 12] for more information.

Hardware Design Flow

This chapter covers the following software design issues to be considered when designing with a Zynq®-7000 AP SoC:

- Using the Vivado IDE to Build IP Subsystems, page 171: You can use the Vivado® Design Suite to build IP subsystems using a variety of tools, which are described in this section.
- Rule-Based Connection, page 173: You can catch errors by running the design validation process, which runs design-rule checks on the block design and reports warnings and errors. This section describes the design validation process.
- Creating Hierarchical IP Subsystems, page 173: You can use the Vivado IP integrator tool to create hierarchical IP subsystems.
- Board Part Interfaces, page 173: The Board Part Interfaces tab in IP integrator shows all the interfaces that are present on a particular board.
- Generating Block Designs, page 174: When a block design or IP subsystem has been created, the source files, IP constraints, and structural netlist can be generated. After this is done, the design can be integrated into a higher-level HDL design, or taken through synthesis and implementation.
- Creating and Packaging IP for Reuse, page 174: The Vivado IP packager enables Vivado IDE users and third-party IP developers to easily prepare an IP design for use in the Vivado IP catalog. The custom IP can then be instantiated into a design using the Vivado Design Suite.
- Creating Custom Interfaces, page 176: The Vivado Design Suite requires that all memory-mapped interfaces use an AXI interface. The suite has a wizard that assists converting a custom IP interface to one that adheres to the AXI interface standard.
- Managing Custom IP, page 176: The Vivado IP Catalog contains built-in repository management features for adding IP cores from another source.
- Vivado High-Level Synthesis (HLS), page 177: The high-level synthesis (HLS) tool transforms a C, C++, OpenCL Kernel, or SystemC design specification into a register transfer level (RTL) implementation that in turn can be synthesized into a Xilinx All Programmable device.

Overview

As programmable devices become larger and more complex, and as design schedules become shorter, use of third-party IP and design reuse are becoming mandatory. Xilinx recognizes the challenges designers face and has created a powerful feature within the Vivado® Design Suite to help solve these challenges. This feature is called the Vivado IP integrator.

The IP integrator enables the creation of block designs. These block designs are essentially IP subsystems containing any number of user-configured IP and interconnect. IP integrator is the feature for doing embedded-processor design in Vivado design tools with Zynq®-7000 AP SoCs, MicroBlaze™ processor designs, and non-processor-based designs. It is used to instantiate high-level synthesis (HLS) modules from Vivado HLS, DSP modules from System Generator, and custom IP made available using the create-and-package IP flow. Designs can be created interactively on the IP integrator GUI canvas or programmatically through a Tcl programming interface.

For details, see the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 18].

Using the Vivado IDE to Build IP Subsystems

The Vivado IDE can be used to build IP subsystems with the IP integrator GUI, scripted tcl flow, or the designer assistance that is part of IP integrator.

Using the GUI to Create an IP Subsystem

The IP integrator has a powerful GUI environment for creating IP subsystems. IP cores can be instantiated and interconnected using various automation features. Custom IP can also be packaged for use in IP integrator. Designs should be reassembled in the GUI environment and taken through the flow. After the design is stable, a script can be written that creates and implements the design. For more information about features of the GUI, refer to the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 18].

Using a Scripted Flow to Create an IP Subsystem

Every action in the IP integrator GUI canvas results in an equivalent tcl command. These Tcl commands are also logged in the Vivado Design Suite journal file. A script can be written based on the journal file, and that script can be re-used to create and implement the design later.

Designer Assistance

To expedite the creation of a processor-based design, use the Block Automation and Connection Automation features of IP integrator. The Block Automation feature should be used to configure the processor and related IP cores, whereas the Connection Automation feature should be used to make repetitive connections to different pins or ports in the design. IP integrator is also board-aware and supports Xilinx® evaluation boards. This means that when an evaluation board is used as the target hardware, IP integrator knows all the interfaces present on that particular board. A design's I/O ports can be connected to existing interfaces on the target board using the Connection Automation feature. Designer assistance also helps with clocks and reset connectivity. Several kinds of tabs, such as the Signals tab and the Board Part Interface tab, help make connections in the block design. Designer assistance helps expedite interconnectivity and eliminates unintended design errors.

Block Automation

For some complex IP and processor-based designs, IP integrator has a feature called Block Automation. This feature supports quick assembly of processor- or IP-based subsystems with commonly used components. After the basic building blocks for an embedded design are assembled, this basic system can be extended by adding IP cores from the catalog.

Connection Automation

When the Block Automation is done and a basic system has been built, external I/O pins need to be connected. The Connection Automation serves this purpose. Connection Automation helps make connections to the I/O pins and it helps make connections to different sources on the design itself. Combined with board awareness, the Connection Automation feature helps connect the block design ports to external interfaces on the target board and to create physical constraints for these ports. Use of this feature, therefore, is highly recommended.

Rule-Based Connection

IP integrator runs basic design-rule checks in real time as the design is being assembled. However, there is a potential for something to go wrong during design creation. As an example, the frequency of a clock pin may not be set correctly. Such errors can be caught by running design validation. Design validation runs design-rule checks on the block design and reports any warnings or errors regarding the design. The warnings and/or errors can then be cross-probed from the messages view into the block diagram. Design validation is recommended to catch design errors that may go unnoticed until later in the design flow.

Running design validation also runs parameter propagation on the block design, one of the most powerful features of IP integrator. The feature lets an IP core auto-update its parameterization based on how it is connected in the design. IP cores can be packaged with specific propagation rules, and IP integrator runs these rules as the diagram is generated.

Creating Hierarchical IP Subsystems

The IP integrator can be used to create hierarchical IP subsystems. This feature can be useful for designs having a large number of blocks that could otherwise become hard to manage on the GUI canvas. Multiple hierarchical levels are supported, so blocks can be grouped based on design function. The use of hierarchy within IP integrator can keep designs modular and neat on the IP integrator canvas.

The visual aspects of design objects can be changed. For example, clocks and resets can be colored differently, and various layers can be enabled and disabled.

Board Part Interfaces

As mentioned above, IP integrator is board-aware. When a block design is targeted to a particular Xilinx evaluation board, the Board Part Interfaces tab shows all the interfaces that are present on that particular board. The interfaces present in the Board Part Interfaces Tab can be connected to create a target design with or without a processor. This is a powerful mechanism for creating a block design. When an interface is selected for connection, all IP cores that can connect to that particular interface are displayed. For example, on a Kintex® KC705 board, if the 8-bit LEDs are selected, three IP choices are presented (the GPIO, the IO Module, and the MicroBlaze MCS IP) that can connect to this interface. Based on the IP selection, a connection can be made to LEDs on the board or, using the Designer Assistance, to the rest of the block design.

Generating Block Designs

When a block design or IP subsystem has been created, the source files, IP constraints, and structural netlist can be generated. After this is done, the design can be integrated into a higher-level HDL design, or taken through synthesis and implementation.

Using Out-of-Context Synthesis for Block Designs

Hierarchical design flows enable the partitioning of designs into smaller, more manageable modules that can be processed independently. In the Vivado Design Suite, these flows are based on the ability to synthesize a partitioned module out-of-context (OOC) from the rest of the design. The most common use with IP integrator is for synthesizing an OOC module and creating a design checkpoint (DCP) file. This block design, if used as a part of the larger Vivado design, does not need to be re-synthesized each time other parts of the design (outside of IP integrator) are modified. This results in considerable run-time improvements and should be considered if run-times are a matter of concern, specifically during the early stages of design exploration.

Creating Remote Block Designs

IP integrator can create stand-alone block designs that can be re-used by multiple Vivado Design Suite projects. After the design is created and put under revision control, multiple design teams can re-use the same block design for creating multiple projects. This is an important re-use feature of IP integrator that should be considered in a team-based environment.

Creating and Packaging IP for Reuse

The Vivado IP packager enables Vivado IDE users and third-party IP developers to easily prepare an IP design for use in the Vivado IP catalog. The custom IP can then be instantiated into a design using the Vivado Design Suite. Use of the following packaging flow results in a consistent user experience, whether using Xilinx IP, third-party IP, or customer-developed IP within the Vivado Design Suite.

Packaging Custom IP and IP Subsystems

The Vivado Create and Package IP function supports the creation of custom IP for delivery in the Vivado IP Catalog. The industry standard IP-XACT format is used for packaging the IP core. The location of the packaged IP can be added to the Repository Manager section of the Vivado Design Suite Project Settings. When a repository of one or more IP cores is added, the IP is shown in the IP Catalog. Users can select and customize IP cores from this catalog.

The flow for using the Vivado IP Packager is:

1. Use the Vivado IP Packager to create and package the IP HDL and associated data files. The IP Packager currently does not support SystemVerilog sources as the top level. A Verilog or VHDL wrapper around the SystemVerilog source is required.
2. Provide the Packaged IP to a team member or customer.
3. Have the end-user add the IP location to the Repository section of the Vivado Design Suite Project Settings.
4. The IP core is shown in the IP Catalog, and the end-user can select and customize the IP core in a way similar to Xilinx-delivered IP.

The Create and Package IP function provides two ways of creating custom IP. The first way is to use the Create and Package IP function to create a new IP core with one or more AXI interfaces. When the interfaces have been created, insert the custom IP and connect to these interfaces. The second way is to package the IP core and include it in the IP catalog for later use in IP integrator. This second way can be used when all HDL files for the custom IP, including the AXI interface, are available.

The Create and Package IP function allows IP end-users to have a consistent experience, whether using Xilinx IP, third-party IP, or custom IP. For more information on creating and packaging IP, see the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118) [Ref 22].

Updating the IP Catalog

The Vivado Design Suite IP catalog is a unified repository that supports searching, reviewing detailed information, and viewing documentation for an IP core. After an IP core has been packaged, users can point to the IP repository containing the newly packaged IP core in the Vivado tool and add the IP core to the IP catalog. The newly packaged IP core is then ready to be used in the IP integrator.

Creating Custom Interfaces

The Vivado Design Suite requires that all memory-mapped interfaces use an AXI interface. The suite has a wizard that assists converting a custom IP interface to one that adheres to the AXI interface standard. The Create and Package IP flow can generate three AXI interface types:

- AXI4: For memory-mapped interfaces that allow bursts of up to 256 data-transfer cycles with a single address phase.
- AXI4-Lite: A lightweight, single-transaction, memory-mapped interface.
- AXI4-Stream: An interface that removes the requirement for an address phase and allows unlimited data-burst sizes.

The *Create and Package IP* flow can create a template AXI4 peripheral that includes HDL, drivers, a test application, bus functional model (BFM), and an example template. After the peripheral has been created, design files can be added to complete the custom IP. See the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 22] and *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 18] for details.

Managing Custom IP

The Vivado IP Catalog contains built-in repository management features for adding IP cores from another source. To make the custom IP visible, place the custom IP in a location that is accessible from the host machine. Then, launch the Vivado Design Suite and run the IP Settings functions from the IP Catalog to register the new user repository location and include the new IP core in the IP catalog.

There are two types of repositories:

- Standard Xilinx Repositories: The Vivado Design Suite ships standard Xilinx repositories. These repositories are always enabled, and they cannot be changed in the Xilinx IP Catalog.
- Configured User Repositories: User repositories are visible from the active machine that contains one or more IP cores.

The Repository Manager allows adding or removing user repositories and establishing precedence between repositories. IP is distinguished through a unique ordered list of elements corresponding to the vendor, library, name, and version (VLNV).

If multiple repositories are referenced and have the same IP VLNV in multiple locations, the Vivado IDE displays the IP core in the repository with highest precedence. The Xilinx IP repositories are always enabled, and they always have lowest precedence.

When the repository setup for a given project is changed, the Repository Manager stores the changes within the project settings. Consequently, changes are visible when the project is reopened on any machine (assuming the repository paths are also available).

Vivado High-Level Synthesis (HLS)

The high-level synthesis (HLS) tool transforms a C, C++, OpenCL Kernel, or SystemC design specification into a register transfer level (RTL) implementation that in turn can be synthesized into a Xilinx All Programmable device.

HLS performs two types of synthesis on the design:

- Algorithm Synthesis: This synthesizes the functional statements into RTL statements over potentially multiple clock cycles.
- Interface Synthesis: This transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system.

Like the many decisions made during manual RTL design, the number of available implementations and optimizations is large, and the combinations of how they impact each other is very large. HLS abstracts these details and helps create an optimal design in the shortest time.

Typically, HLS is used after a software bottleneck has been identified by profiling the software application within the SDK or third-party tools. When the bottleneck function(s) have been identified, the next step is to move those function(s) into hardware using HLS. Finally, the HLS design can be exported as an IP core to be used in IP integrator. HLS also generates a simulation model for the IP core that can be used to verify the IP function. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 9] for more information.

Summary

In general, the following steps should be followed to capture an embedded design in IP integrator:

1. Add processor IP, such as the Zynq-7000 AP SoC or the MicroBlaze processor.
2. When the processor IP is instantiated, designer assistance is available. Use designer assistance to configure the processor and peripherals.
3. Customize the processor further, if needed. This step is needed if more function and control over clocks, resets, I/O ports, etc., is desired.
4. Add peripherals, and connect them using designer assistance, when available.
5. Add connectivity IP for external interfaces, such as GPIO, Ethernet, etc.
6. Add custom accelerators for the Processing Logic, if needed.
7. Connect and review clock and reset domains, using the Signals tab or the Make Connection wizard.
8. Run design-rule checks by validating design. Resolve any errors or warnings flagged during design validation, and run design validation until no further errors are flagged.
9. Synthesize, implement, and generate bit-stream for the design.
10. Export the design to the SDK for software development.

Software Design Flow

This chapter describes software design flow for the Zynq®-7000 AP SoC and provides guidance for various software development roles, referred to as *personas*. The design flow for each persona includes an overview of the tool flow and available solutions—from Xilinx and its partners—at each stage. References are provided to external documents and other sections in this guide if more information is needed.

Unlike ASICs and ASSPs, the Zynq-7000 AP SoC has programmable logic (PL, the hardware-configurable FPGA portion of the device) and a processing system (PS, the two ARM Cortex-A9 cores). The hardware design and configuration of PL can be done using the Xilinx® Vivado® tools. Refer to Chapter 5, Hardware Design Flow. The hardware design files from the Vivado tools are sent to the Xilinx SDK for software development. Figure 6-1 shows the hardware and software development flow for the Zynq-7000 AP SoC.

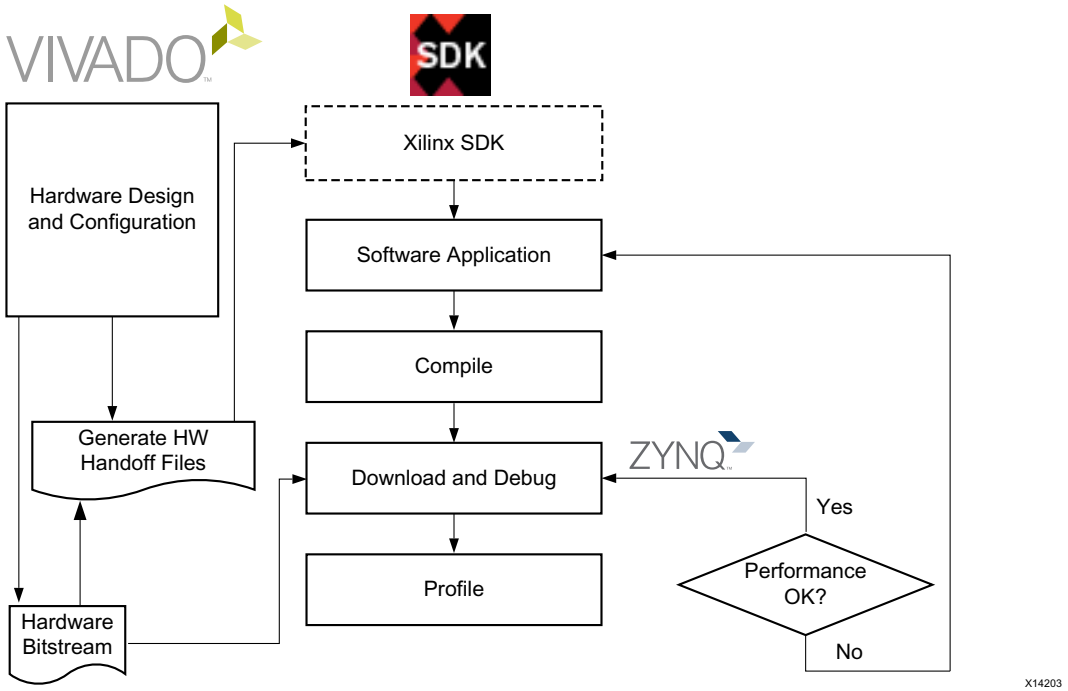

Figure 6-1: Hardware and Software Development Flow

Referring to Figure 6-1, page 179, the files sent from the hardware flow to the software flow (referred to as *handoff files*) contain information such as the hardware specification, PS peripheral information, register memory map, and a bitstream for the PL. The handoff files insulate software developers from the Zynq-7000 AP SoC reconfigurable hardware. The handoff files make the hardware appear to software as an ASSP. Software developers can use the handoff files to design firmware, drivers, and a board-support package (BSP).

Typically, the software design flow of Zynq7000-AP SoCs involves development of one or more of the software layers shown in Figure 6-2.

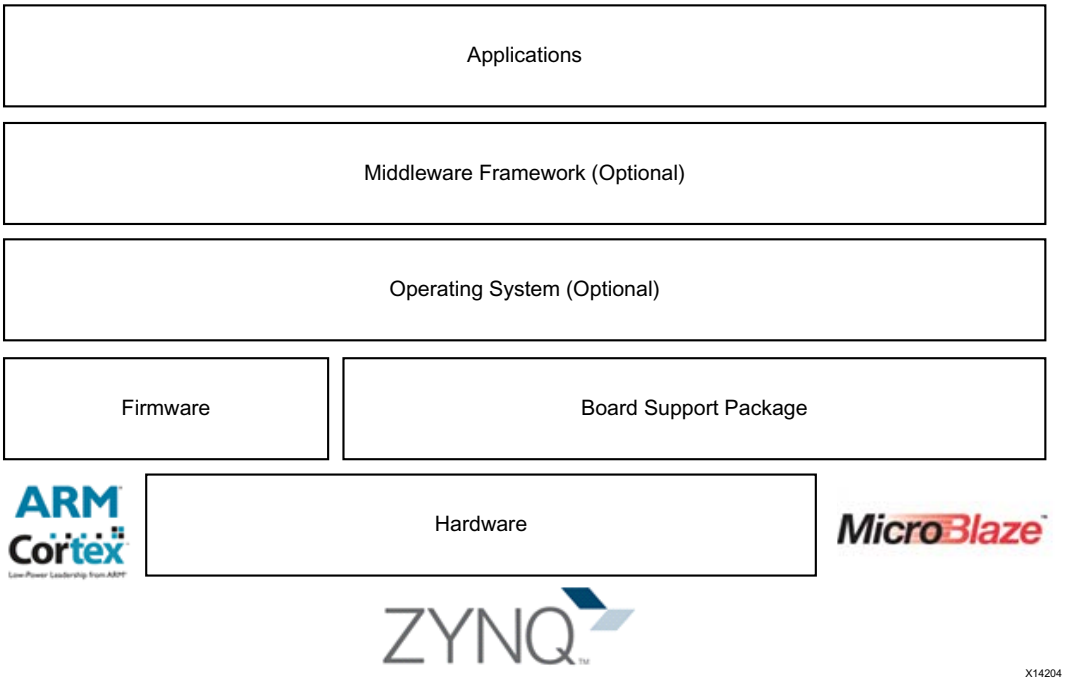

Figure 6-2: Software Development Layers

The remaining sections cover software development flow of three personas: board bring-up developer, driver developer, and application developer. Each persona works on one of the layers in Figure 6-2. A persona can refer to the corresponding section in this chapter to understand software development flow pertaining to their area of work.

Board Bring-Up Development

Board bring-up activities include developing low-level firmware, setting up the boot sequence, and basic tests for the interfaces and the peripherals. This section describes the Xilinx development solutions available for basic board bring-up activities, divided into the following phases:

1. PS initialization
2. PL configuration
3. Memory and peripheral testing
4. Hardware and software debug

PS Initialization

Xilinx tools auto generate the `ps7_init.tcl` and `ps7_init.c` files when Vivado exports a design. The `ps7_init.c` file is an initialization file auto generated by Vivado and used by the first stage boot loader (FSBL) to initialize the PS. When bringing up the board, the first step is to connect to the target Zynq-7000 AP SoC through the Xilinx debugger and run the `ps7_init.tcl` script (provides the same results as `ps7_init.c`) to initialize the PS peripherals, clocks, DDR, PLL, and MIO. To verify the PS initialized successfully, execute memory reads and verify commands from the XSDB debugger. System peripherals can be read and written to verify basic functionality.

The FSBL can load a second stage boot loader, such as U-Boot, or it can load and boot an operating system. In bare-metal implementations, FSBL directly loads the application code.

Refer to the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7] for more information on the FSBL, including normal and secure boot.

On third-party debuggers, the FSBL can be used to initialize DDR memory and load the application code, which can be debugged using the debugger. For example, the FSBL can be loaded in OCM, executed for a few seconds, stopped, and then the debug tool can be used (via JTAG) to load the application code in DDR for debug. Most debuggers support script mechanisms for automating this process.

The Xilinx SDK repository provides bare-metal drivers that can be used for board bring-up.

U-Boot

U-Boot is an open-source universal boot loader that can act as a primary or secondary boot loader. U-boot can be used to load a stand-alone application, a bitstream, or a Linux OS kernel. On Zynq-7000 AP SoCs, U-Boot is used as a secondary boot loader, but it can be configured and rebuilt to act as a primary boot loader. The FSBL is designed to bring up a secondary boot loader like U-Boot. U-Boot can be loaded from an external device, and it can be used to initialize PS peripherals not initialized by the primary boot loader. U-Boot or any other secondary boot loader can be removed from the final product to reduce boot time.

The flow charts in Figure 6-3 outline the boot sequence and the files involved in the boot process.

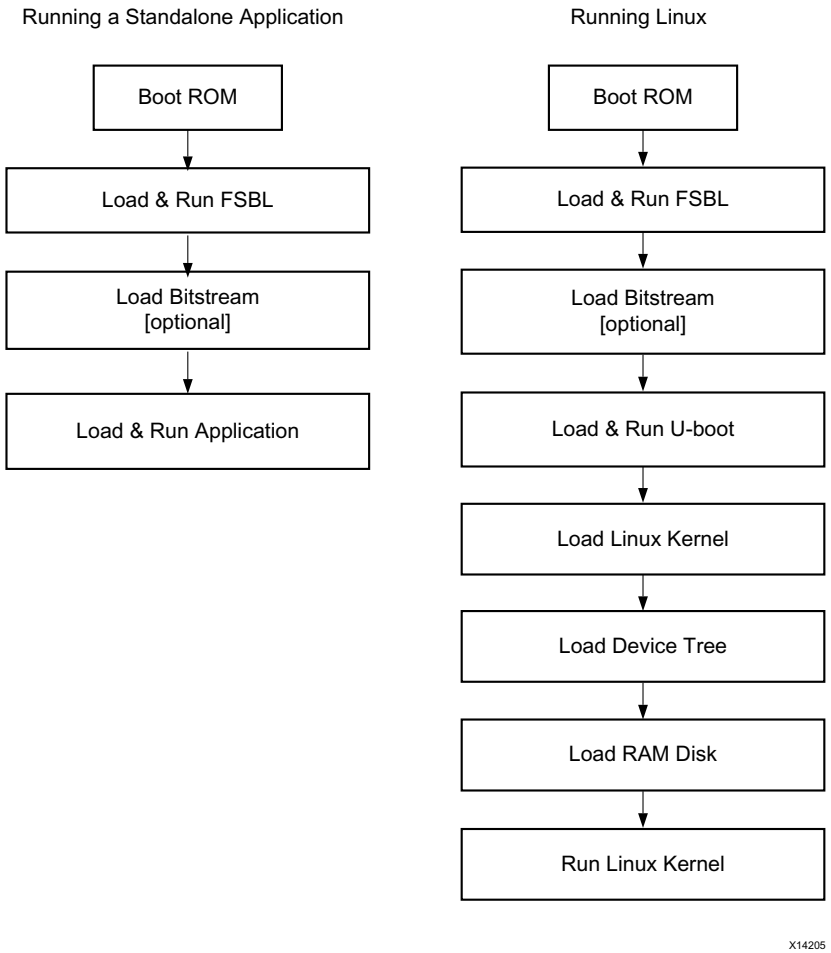

Figure 6-3: Boot Sequence and Boot Files

The open-source Xilinx U-Boot project is available at the Xilinx Git repository. Xilinx U-Boot is also released as a part of the PetaLinux offerings. Refer to the Xilinx U-Boot wiki page [Ref 57] for more information on configuring and building U-Boot.

PL Configuration

The PL is configured either by software running on the ARM core or through JTAG. Xilinx supports PL configuration using the FSBL, U-Boot, or Linux.

On production boards, the FSBL can configure the PL using the PCAP interface. During the development and debug phase, the PL can be configured through JTAG from either the Vivado tools or Xilinx SDK. After the PL is configured, the processor can access PL peripherals in the same manner as PS peripherals. The debugger can read and write PL peripheral registers at this stage.

Memory and Peripheral Testing

The Xilinx SDK has a memory test that can verify the DDR memory and the signal integrity of the memory controller and DDR interface. The bare-metal memory test application can be used to exercise a DDR memory power-on self-test. The DRAM test application template is located in your installation directory at `SDK\2014.2\data\embeddedsw\lib\sw_apps\zynq_dram_test`.

Example drivers are available for many PS peripherals. They can be found in the `SDK\<Version>\data\embeddedsw\XilinxProcessorIPLib\drivers\<Peripheral>` folder of your installation directory. The folder also contains sample test applications for Xilinx Soft IP cores. These tests can be used to test peripherals during the board bring-up phase. Also, the tests can be modified to develop a comprehensive power-on self-test or built-in test.

Hardware and Software Debug

There may be cases when DDR initialization fails. If this occurs, a connection to the debugger can be made by debugging the FSBL stage. Any application downloaded to DDR can be debugged after that. The Xilinx SDK supports heterogeneous multicore debugging, allowing debug of both MicroBlaze™ processors and the ARM cores. Also, cross triggers can be used when it is unclear whether the problem is in software or hardware. Refer to the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) [Ref 15] for more information on setting up cross triggers.

Driver Development

Driver developers create software drivers for SoC and on-board peripherals, establishing interfaces for higher software layers such as the OS or bare-metal applications. Peripheral drivers must be created for the ARM Cortex-A9 and MicroBlaze cores. This section covers points to consider when developing bare-metal and Linux drivers for the configured hardware, and it highlights available Xilinx solutions.

Bare-Metal Drivers

The Xilinx SDK supports various file types used in driver development (both Xilinx IP drivers and custom drivers) and used during the software development cycles. The file types and directory structure are shown in Figure 6-4.

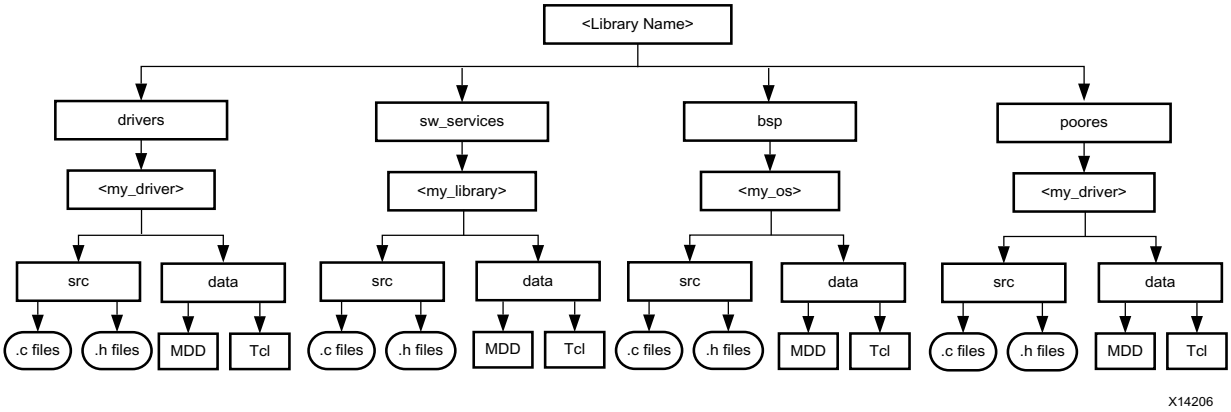

Figure 6-4: Directory Structure of Drivers, OSs, and Libraries

Handoff File Details

The software development cycle — from handoff to debug — is shown in Figure 6-5.

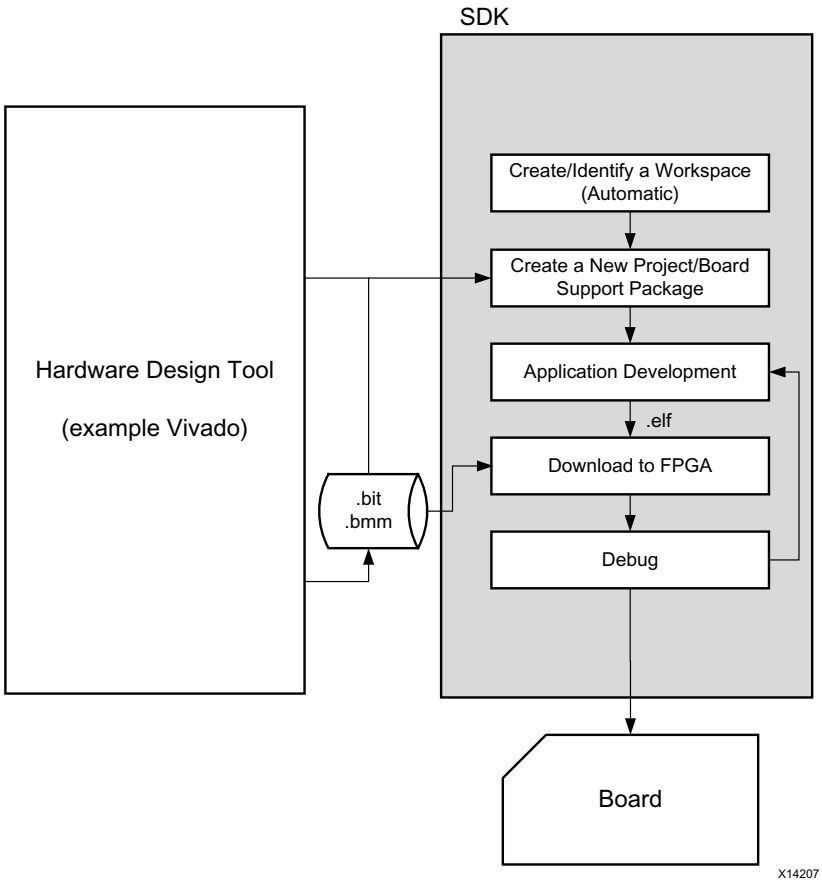

Figure 6-5: Software Development Cycle with the SDK

The software development cycle begins with handoff files created during the hardware design flow. After preparing the hardware design with the Vivado tools, a hardware handoff folder is created for use by the SDK in driver and application development. Beginning with release 2014.2, Vivado hardware export creates a single, compressed hardware definition file (`*.hdf`) and writes it to the `SDK_Export` folder. This file contains hardware details, such as base addresses and version information. From the SDK workspace, a hardware platform project is created and the project within the workspace is configured by pointing it to the `*.hdf` file. The SDK (beginning with release 2014.2) supports import of traditional XML files in addition to the `*.hdf` files.

When building the BSP project, the SDK generates an `xparameters.h` file, populating it with hardware details (such as base addresses and interrupt IDs) in the form of `#define` constants. Software drivers (stand-alone platform) and applications can reference these constants in the code.

Driver File Organization

The drivers for Xilinx IPs (PS peripheral IPs and soft IPs) are shipped with the SDK. These drivers are located in the `data/embeddedsw/XilinxProcessorIPLib/drivers` folder in your SDK installation folder. Each driver is located in a separate folder. For example, the IIC PS driver version 2.0 is located in the `data/embeddedsw/XilinxProcessorIPLib/drivers/iicps_v2_0` folder of your SDK installation.

Each driver folder contains four subfolders:

- `data`: This folder contains the MDD file and Tcl script file.
- `doc`: This folder contains help files that provide details of APIs and data structures implemented in the driver.
- `examples`: This folder contains C source files demonstrating examples on how to use the driver. These examples are ready to build and launch on the hardware. The examples can be used as-is when following the procedure for building stand-alone applications. The examples can also be modified as needed.
- `Src`: This folder contains the driver source code (C and H files).

The source file repository is organized as shown in Figure 6-6.

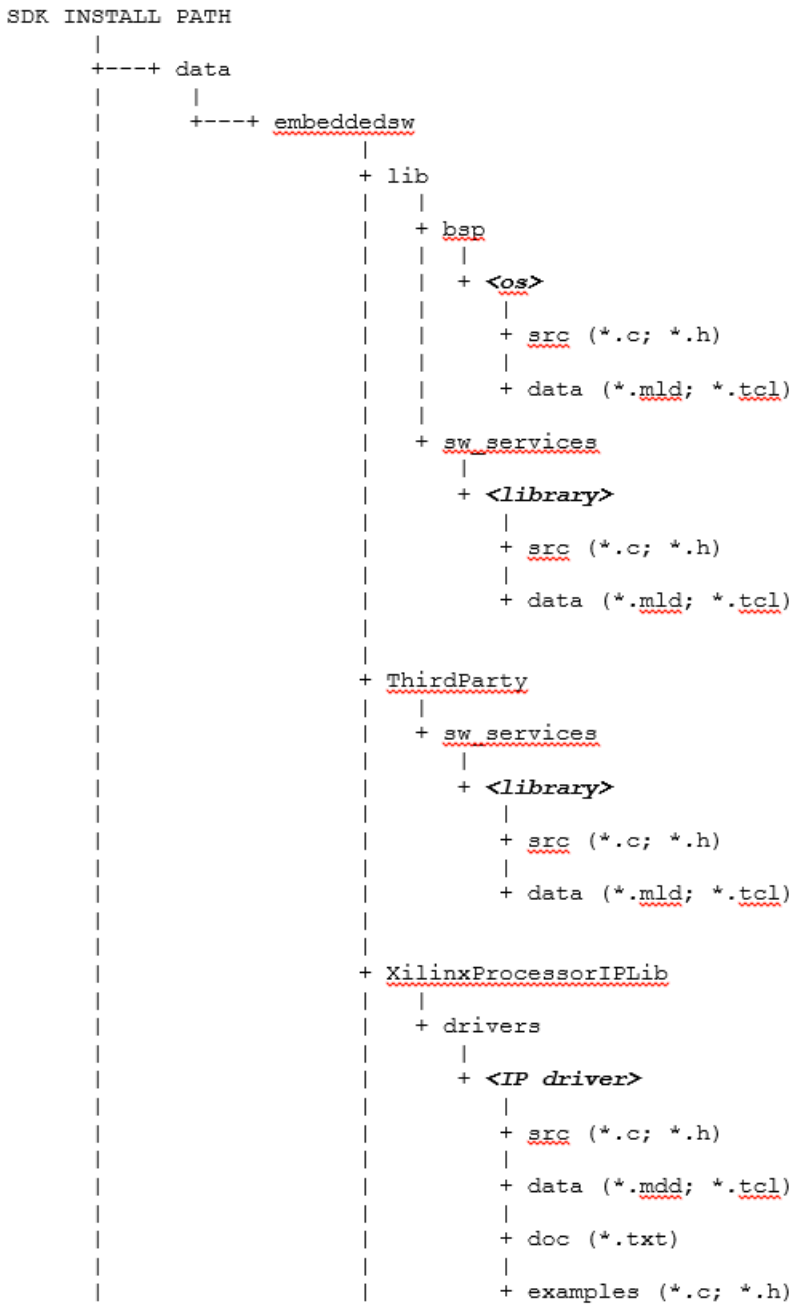

Figure 6-6: Source File Repository

To load driver source code into the project workspace, the SDK reads the MDD file in each version folder. The SDK selects the corresponding driver version based on parameter values in the MDD file. When the SDK finds a driver with multiple versions, it takes the highest version as default. However, the programmer can select the required version from a drop box.

The MDD is used to assist with the following tasks:

- Integrating custom drivers into the SDK.
- Modify existing drivers and integrate the variants into the SDK.
- Modify the BSP, such as adding new software services.

IP Blocks and Drivers

The IP blocks and drivers can be characterized as shown in Table 6-1. As can be seen in the table, the SDK can identify and load any driver source code for either type of IP block.

Table 6-1: IP Block and Driver Combinations Supported by the SDK

	Soft IP (PL)	PS IP (PS)
Xilinx drivers	Supported	Supported
Custom drivers	Supported	Supported

PS IP

PS IP blocks are hardware blocks that are part of the Zynq-7000 AP SoC PS. The PS includes several peripherals (USB, I2C, Ethernet, etc.) plus the ARM Cortex-A9 processor.

The application requires driver support to access the peripherals. Bare metal and Linux drivers are provided. The SDK provides pre-built bare-metal drivers and Linux kernel (Xilinx supported) drivers for all PS IP peripherals.

Soft IP

Soft IP blocks are hardware blocks built as part of the design and loaded into the Zynq-7000 AP SoC PL. Some examples of soft IP blocks in the PL are VDMA, AXI-Ethernet, and third party IP blocks.

The application requires driver support to access the peripherals. Bare metal and Linux drivers are provided. Bare-metal drivers are provided by the SDK and Linux drivers are provided by the Linux kernel source tree from Xilinx.

The following steps must be followed so that a third-party IP driver can be used in the SDK:

- Create a driver for the IP block.
- Create the associated data definition file (MDD) and data generation file (Tcl) for use by the SDK.
- Set the driver path as an external repository of the SDK project. The SDK can now recognize and load the driver normally.

The bare-metal driver and application development flow is shown in Figure 6-7, page 189.

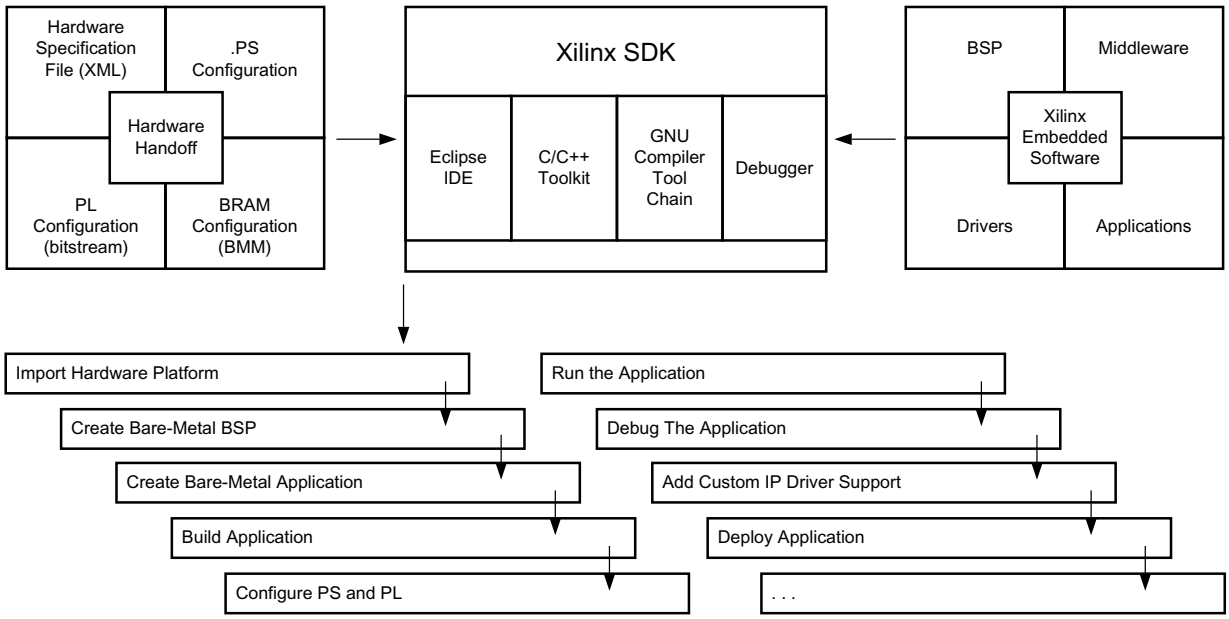

Figure 6-7: Bare-Metal Application Development Overview

For more information, refer to the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

Xilinx Driver Flow

Soft IP drivers are shipped with the tool installation. The drivers can be directly imported into the BSP from the standard driver location.

There are two methods for modifying and using the standard drivers:

- Method 1:
  - Modify the driver source code in the default driver path.
  - Reload and build the BSP. This action will cause the modified driver to be loaded.
- Method 2:
  - Create a driver folder structure in a local path.
  - Copy the standard driver and assign a different version number.
  - Modify the source code and the corresponding MDD file.
  - In the SDK, point to the local repository path so that the SDK uses the modified driver.

For more information on MDD parameters, refer to the "Microprocessor Driver Definition (MDD)" information in the *Generating Basic Software Platforms Reference Guide* (UG1138) [Ref 23].

Custom Driver Flow

Custom IP software can be in the form of libraries or drivers. The Xilinx MDD and MLD files provide the capabilities needed to create drivers for custom IP blocks, enabling developers to:

- Develop new drivers and integrate them into the SDK
- Customize existing drivers
- Customize a BSP format and folder structure tailored to the OS tool chain
- Customize the OS based on the hardware

The MLD format describes the parameters that can be used to customize libraries and operating systems. For more information on user-written libraries and operating systems that must be configured by the Libgen tool, refer to the "Microprocessor Software Specification (MSS)" chapter in the *Generating Basic Software Platforms Reference Guide* (UG1138) [Ref 23].

MDD files contain directives for customizing or creating software drivers. For more information on the MDD format and the parameters that can be used to customize or create drivers, refer to the "Microprocessor Driver Definition (MDD)" chapter in the *Generating Basic Software Platforms Reference Guide* (UG1138) [Ref 23].

The method for modifying and using custom drivers is the same as method two for Xilinx drivers, described in Xilinx Driver Flow, page 189.

Linux Driver Development

Linux on Zynq-7000 AP SoCs requires drivers for both PS IP (peripherals) and soft IP (custom logic implemented in the PL). Drivers can be added to Linux as a part of the kernel or as loadable kernel modules. Xilinx provides a complete Linux source targeted to the Zynq-7000 AP SoC. For more information and to download a copy, see The Official Linux Kernel from Xilinx web page [Ref 98]. The Xilinx Wiki's Linux drivers page lists information on drivers from Xilinx. The Xilinx Wiki's Linux page also provides information on Linux-related offerings from Xilinx.

Xilinx releases pre-built images for Zynq-7000 AP SoC evaluation boards (ZC702, ZC706, and ZedBoard) with the release of Xilinx tools.

Device Tree

Information about hardware added or removed (such as an IP block in the PL) can be sent to the kernel using a device tree. The device tree can also be used to convey specific information about the hardware.

The Xilinx Wiki's build device tree page describes how to build a device tree BSP using the Xilinx SDK. The SDK requires the following two files when creating a device tree BSP:

- Hardware handoff files from the Vivado tools.
- Device tree files (`.tcl` and `.mld`) from the device tree release in the SDK repository.

Root File System

Xilinx provides the initial root file system for Zynq-7000 AP SoCs: ramdisk/initrd. Other supported file systems can be mounted on the root file system after system boot. The Xilinx Wiki has more information on how to build and modify the root file-system.

Linux Kernel Debug

The Xilinx SDK's system debugger can be used for Linux kernel debug through JTAG. The system debugger is a heterogeneous multicore debugger that enables simultaneous debug of MicroBlaze and ARM cores. Processor cores can be controlled individually by the system debugger when the Linux kernel is running in SMP mode. PL address space can be debugged, and during debug PL memory space is shown in the memory view.

The SDK help has information on how to use the system debugger to debug a Linux kernel on a Zynq-7000 AP SoC. The SDK help also describes how Linux kernel symbol files can be added during debug configuration and the steps for mapping the compilation path from the Linux host machine to the Linux source tree in a Windows environment.

By default, when debugging using *attach to running target* mode, the PL registers will not be accessible. The environment variable HW_SERVER_ALLOW_PL_ACCESS must be set using the SDK shell, and the hw_server started using the following sequence:

1. Kill all running instances of hw_server.
2. Launch a terminal command prompt from the SDK and set HW_SERVER_ALLOW_PL_ACCESS.
3. Launch hw_server from the same shell.

Alternatively, the GDB debugger can be used for Linux kernel debug through JTAG. The ARM DS-5 also supports Linux kernel debug on Zynq-7000 AP SoC devices.

Application Developer

Applications are developed to run on bare metal or an operating system such as Linux or FreeRTOS. This section describes how applications can be developed, debugged, and analyzed for Linux and bare-metal platforms running on Zynq-7000 AP SoCs. Use cases are provided to illustrate how the SDK tools assist in developing, debugging, and analyzing the software application.

Bare Metal

The SDK supports software application development on the exported hardware design. The SDK extracts hardware information from the design exported by the Vivado tools. Based on that information the SDK provides a complete application development environment. Application development requires a board-support package, including hardware initialization, drivers, and a set of libraries, such as TCP/IP Stack (lwIP), and FAT File System (FFS).

The Xilinx SDK provides two template types:

- Hardware (PS only) templates for Zynq-7000 AP SoCs. Using this template, an application can be developed without creating a design with the Vivado tools. Templates are available for the ZC702, ZC706, ZedBoard, and MicroZed boards.
- Application templates. Predefined applications, such as *hello world* and an empty application can be generated for any hardware design using the available peripherals. Other applications, such as lwip, can be created with minimal effort in the SDK.

The SDK can also be used in batch mode, where you can create and build applications through a command line interface, without invoking the SDK IDE.

For more information, refer to the *Xilinx Software Development Kit Help* (UG782) [Ref 6].

The Xilinx SDK comes with a set of libraries you can use to facilitate stand-alone application development. For more information, see Xilinx Libraries for Stand-Alone Systems, page 153.

Use Case: Develop, Debug, and Integrate

The dialog box in Figure 6-8 shows the creation of a C/C++ application. For more information, refer to the SDK Help [Ref 6]: Xilinx C/C++ New Project Wizard.

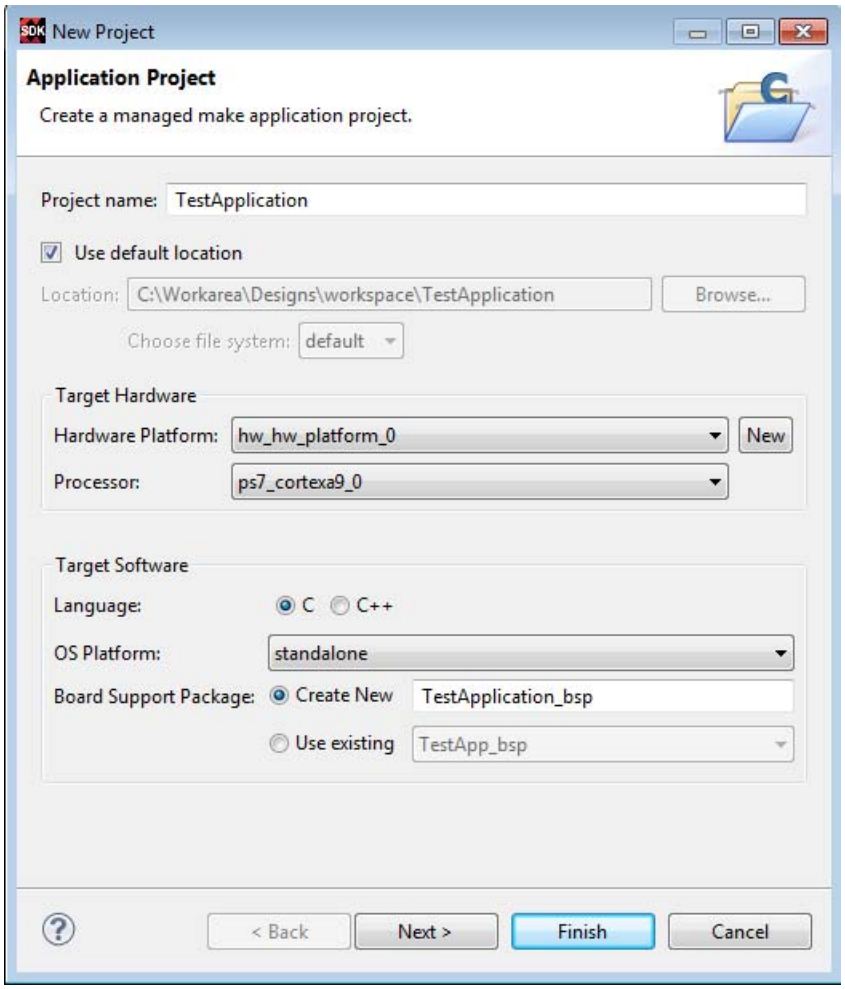

Figure 6-8: C/C++ Application Dialog Box

After an application is created, it must be built. For more information, refer to the SDK Help [Ref 6]: Building Projects web page.

For more information on running and debugging applications, refer to the SDK Help [Ref 6]: Running, Debugging, and Profiling Projects.

Linux Application Developer

Linux applications use the Linux software stack, which includes the Linux kernel, libraries, file system, and the driver APIs.

Some of the architectural choices that affect the development of Linux applications are:

1. Symmetric multiprocessing (SMP) or asymmetric multiprocessing (AMP) system.
2. Resource sharing in AMP systems. AMP systems require memory mapping for separate cores. For more information, refer to the sample AMP configurations.
3. SMP system interrupt affinity.
4. Third party libraries. Refer to Libraries and Middleware, page 152, for more information. Licensing for open-source libraries and software used on proprietary projects should be considered.
5. Secure execution, such as TrustZone execution.
6. Driver support for custom IP.
7. Inter-processor communication mechanism. For more information, refer to *How a MicroBlaze can peaceably coexist with the Zynq-7000 AP SoC* [Ref 82].

For more information on architectural design considerations, see Chapter 4, Software Design Considerations.

Xilinx provides documents, basic software blocks, and reference designs that can be used during the architecture design and evaluation stage. For more information, refer to the Xilinx Linux wiki page [Ref 54].

Application Development Flow

The Xilinx SDK provides a set of tools and a suggested development flow for developing Linux applications. This is described in detail in the "Software Application Development Flows" chapter of the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

Alternatively, in-house or third-party solutions can be used to develop Linux applications. The following support is required by the tools for developing applications for Zynq-7000 AP SoCs:

- A build system, including a cross-development tool chain on the host system.
- A Linux kernel, device tree, software stack, and libraries.
- A Zynq-7000 AP SoC Linux target or emulation platform.

Xilinx Tool Flow for Linux Application Development

Xilinx provides Linux drivers, BSPs, and source code as a part of the PetaLinux tool flow and Xilinx Linux distribution. The PetaLinux flow is designed for FPGAs and devices similar to the Zynq-7000 AP SoC. It offers configuration and Linux deployment capabilities for the Zynq-7000 AP SoC, including a complete tool chain, BSPs, and Xilinx evaluation boards. For more information on configuring PetaLinux, refer to the Xilinx Wiki's PetaLinux page.

PetaLinux is covered in more detail in OS and RTOS Choices, page 142.

Figure 6-9 shows the Xilinx tools used in Linux application development.

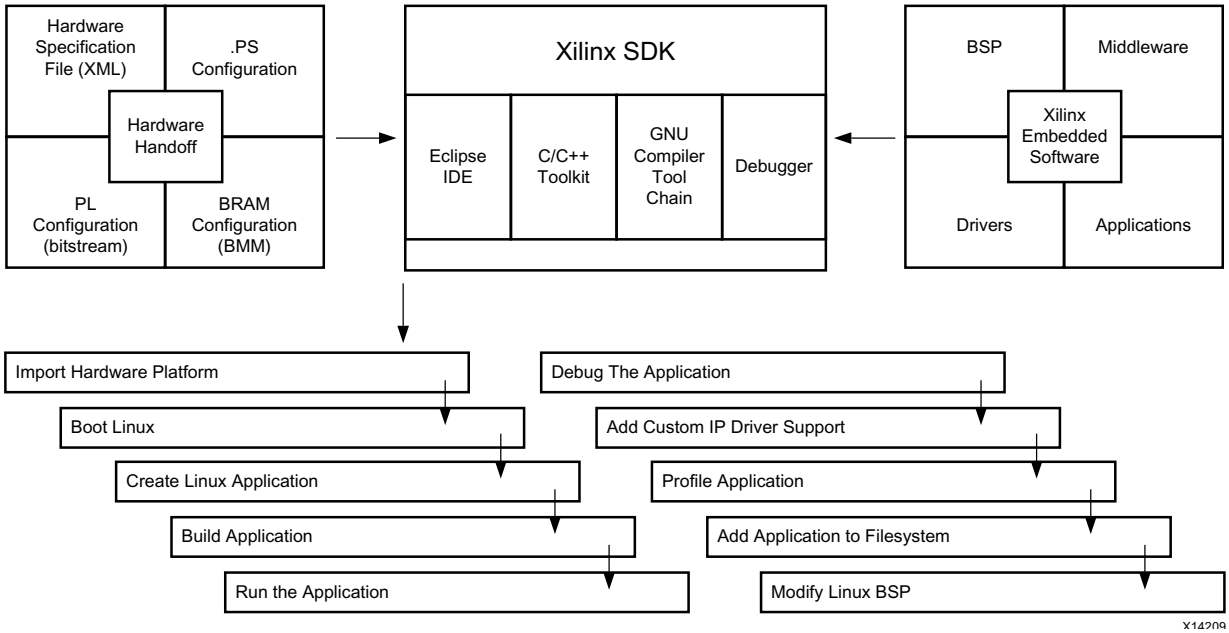

Figure 6-9: Linux Application Development using Xilinx Tools

Application Development Using the Xilinx SDK

The Xilinx SDK supports development and debug of Linux applications for Zynq-7000 AP SoCs. It supports the ARM Linux GCC tool chain (Sourcery CodeBench Lite) and offers various Xilinx tools for Linux application development. The Xilinx SDK also comes bundled with Linux application templates and supporting libraries to help developers evaluate the Zynq-7000 AP SoC with minimal setup.

For more information on Linux application development flow, refer to the "Software Application Development Flows" chapter of the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

Figure 6-10 shows how a Linux application development project can be created in SDK, using an OpenCV Example Application template. SDK provides the libraries required for OpenCV applications.

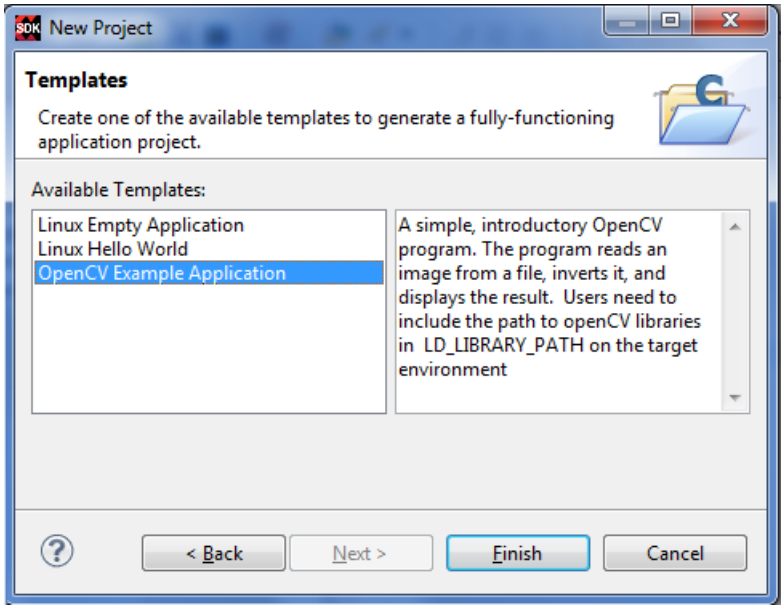

Figure 6-10: Creating a New Linux Application in the Xilinx SDK

Figure 6-11 shows how different build settings and libraries can be selected when building Linux applications using SDK.

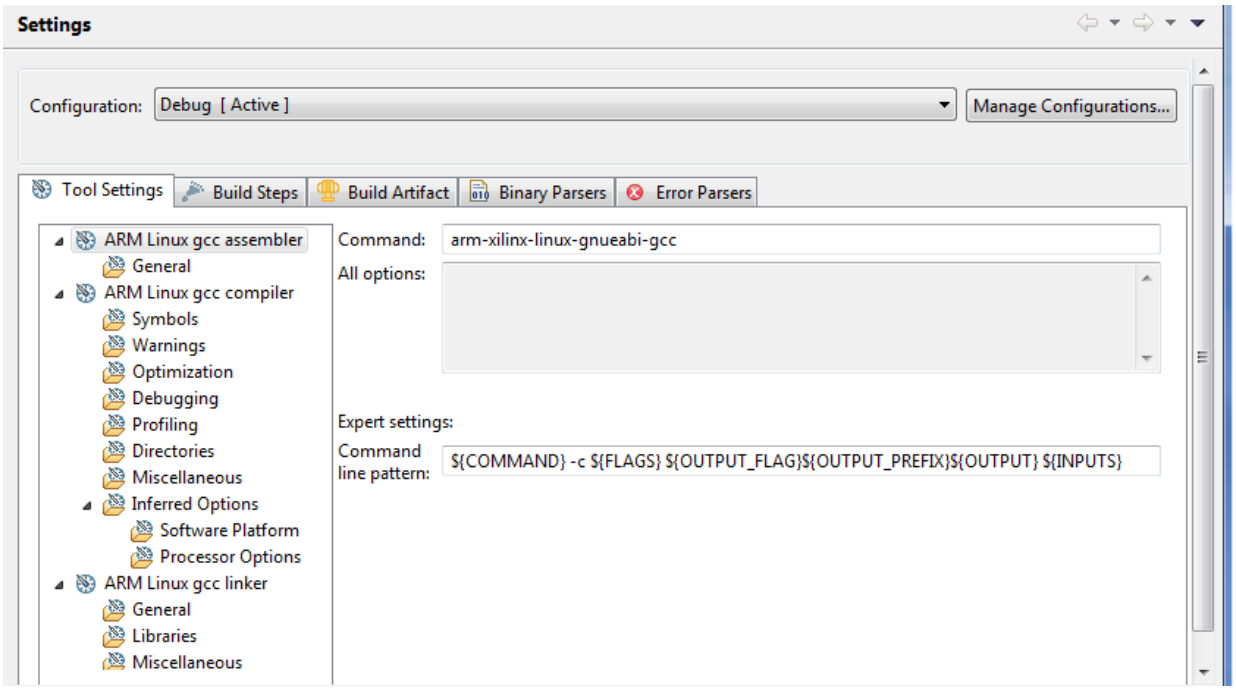

Figure 6-11: Xilinx SDK Build Setting and Libraries

SDK enables users to connect to a Zynq-7000 AP SoC Linux target using an IP address. After the connection is made, the application executable can be copied to the Linux target and run.

Linux Application Debug

Linux applications can be debugged on Zynq-7000 AP SoCs using GDB or the Xilinx system debugger. The Xilinx SDK also enables local-target or remote-target Linux application debug. Linux application debug is supported by the Xilinx SDK using gdbserver for GDB, or a TCF agent for the system debugger. A TCF agent process must be running on the Linux target to debug Linux applications using the system debugger.

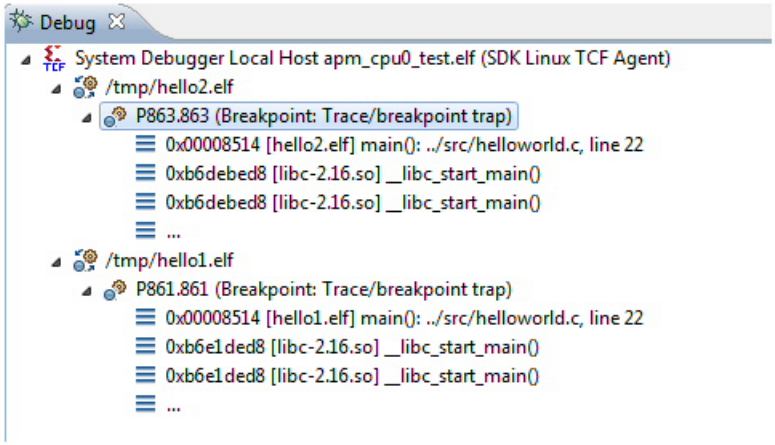

Figure 6-12: Xilinx SDK Build Setting and Libraries

Linux applications can also be developed and debugged using other IDEs, such as ARM DS-5. If a command-line approach is preferred, a build system developed in-house can be used, or one can be obtained from an OS vendor. The Zynq-7000 AP SoC ecosystem is supported by a variety of OS partners. A list of OS partners is can be found in OS and RTOS Choices, page 142.

Xilinx SDK Tools and Packages

Xilinx provides tools and software packages for developing Zynq-7000 AP SoC software. The development utilities are released as:

- Xilinx tools released in the SDK IDE.
- Software packages, including a bare-metal BSP and drivers, plus source code.
- Software released through Xilinx Git.

Tools Released in the Xilinx SDK

Tools and utilities released in the Xilinx SDK are described in the following sections.

Boot Image Creation

The Xilinx SDK provides facilities for creating a boot image. Bootgen can stitch the boot-loader FSBL, the bitstream, and application executable files (including U-Boot) to generate a boot image file in `.bin` or `.mcs` format. The SDK also has a *Create Boot Image* wizard option for adding the partition images and creating a bootable image. For more information, refer to the following:

- "Creating a Zynq Boot Image for an Application" in *Xilinx Software Development Kit Help* (UG782) [Ref 6].
- "Using Bootgen" appendix (available at this link) in the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].
- Vivado Design Suite QuickTake Video: How to Create Zynq Boot Image Using Xilinx SDK

Programming Utilities

Program Flash

The program flash utility allows users to download .bin and .mcs files to on-board flash devices. Figure 6-13 shows the flash devices supported by this utility.

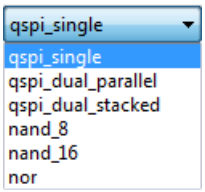

Figure 6-13: Flash Devices Supported by the Program Flash Utility

Program FPGA

The PL bitstream can be programmed in different ways. One method is to use the *Program FPGA* utility in the SDK GUI.

Debuggers

The Xilinx SDK provides heterogeneous debuggers that support ARM and MicroBlaze debug. The debug capability is supported by the GUI and command line interface. The command line interface supports Tcl scripting, enabling command execution in JTAG mode. The following debuggers are supported by the Xilinx SDK:

- System Debugger.
- XSDB: Xilinx System Debugger (command line flow).
- XMD: Xilinx Microprocessor Debugger.
- GDB debugger.

To ensure that the MicroBlaze core is visible to the debugger, MDM must be enabled when creating the hardware design in Vivado.

The SDK system debugger provides hardware-software debug features that assist in debugging code running on Zynq-7000 AP SoCs. The debugger also facilitates hardware-software cross triggers.

For more information, see Chapter 7, Debug.

Performance Analysis

Performance analysis is an important step in identifying code bottlenecks. The Xilinx SDK provides code profilers and a performance analyzer. For more information, see Chapter 2, Profiling and Partitioning.

Application Templates

Figure 6-14 shows the application templates bundled in the SDK for bare-metal programming.

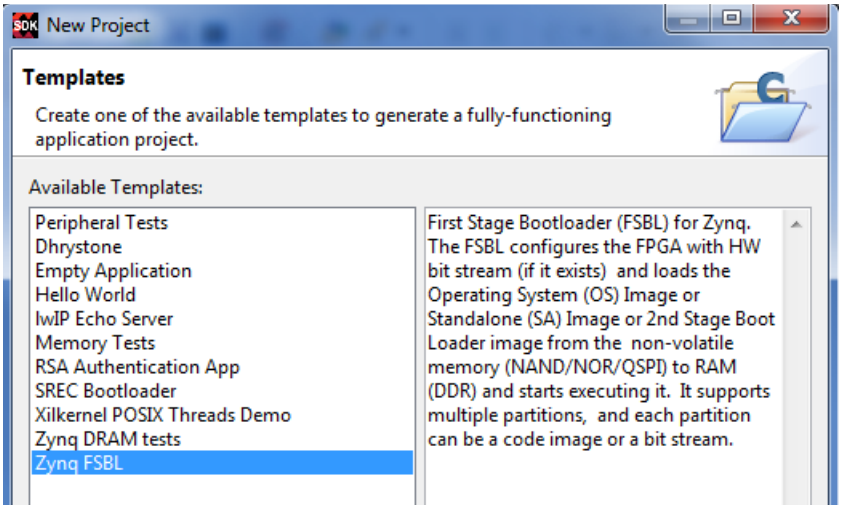

Figure 6-14: SDK Application Templates

Xilinx Git Software Solutions Release

Xilinx releases system software for Zynq-7000 AP SoCs and MicroBlaze targets on the Xilinx Git page. Some of the packages released on the Git page include:

1. Linux xlnx: The Xilinx Linux kernel
2. U-Boot-xlnx: The Xilinx U-Boot repository
3. Device tree: The Xilinx SDK Linux device tree generator
4. meta-xilinx: Xilinx device and board support for Yocto/OE-core.

Xilinx Software Development Tools

Software development tools provided by Xilinx include the SDK and the Xilinx Yocto Project.

Xilinx SDK

The Xilinx SDK is an Eclipse-based IDE and contains Xilinx tools and software packages. It enables embedded software development for Xilinx FPGAs and Zynq-7000 AP SoC devices. The SDK contains essential software packages, such as pre-defined hardware configurations for Xilinx evaluation boards, peripheral and IP device drivers, a bare-metal BSP repository, firmware with required supported libraries, and application templates. The packages are released with the latest source code so that they can be modified as-needed. The SDK also contains other tools that are used to create a boot image, program the processor, and test performance.

The Xilinx SDK is free and can be downloaded from http://www.xilinx.com/tools/sdk.htm.

PetaLinux

PetaLinux offers a solution for the development and deployment of Linux-based software on All Programmable devices. The PetaLinux tool set includes an installer, development tools, Linux-based BSPs for Xilinx evaluation boards, and library frameworks. The Xilinx Wiki contains a detailed description of PetaLinux. The wiki page also contains documentation on the overall Xilinx Linux offerings and Xilinx open-source software.

Xilinx Yocto Project

The Xilinx Yocto Project provides templates, tools and methods for creating custom Linux-based systems for embedded products built on MicroBlaze and Zynq-7000 AP SoCs. Included are meta files, development boards, and a QEMU emulator. The Xilinx Wiki Yocto page provides links to Xilinx Yocto releases. This includes meta-xilinx, a Yocto BSP layer supporting Linux kernel build, U-Boot and Poky distributions.

The Xilinx Yocto project also enables up-streaming of relevant drivers and software packages from non-Xilinx entities. Customers and OS vendors are encouraged to develop compliant software and upstream drivers, which helps enhance the product and overall Xilinx community.

For more information on the Xilinx Yocto Project, refer to the following:

- Xilinx Yocto wiki page [Ref 58]
- Yocto Project web page [Ref 102]

Debug

This chapter covers the following information about the debug methodology:

- Overview, page 203: This section provides an overview of the debug methodology for developing applications using a Zynq®-7000 AP SoC.
- Software-Only Debug, page 204: Embedded software debug is done using the Xilinx® Software Development Kit (SDK). The SDK debug perspective provides a comprehensive debug environment.
- Simulation-Based Debug, page 209: Simulation emulates the final design's behavior in a software environment. Simulation helps verify design functionality by injecting stimulus and observing the output result.
- Board Debug, page 210: After a design is implemented, it must be tested on hardware. Software debug of hardware can be done using the SDK tool. Debug can be useful in finding problems in the FPGA or fabric side.
- Hardware and Software Co-Debug, page 211: Xilinx provides a solution allowing you to simultaneously debug the processing system (PS) and the peripherals in the programmable logic (PL).
- Virtual Platforms, page 212: Virtual platforms are software systems that mimic the behavior of hardware. They are fast, functional models of an embedded platform.

Because of the programmable logic (PL) density and the processor system complexity, the debug methodology described in this section is critical to the development of applications using a Zynq®-7000 AP SoC.

The test and debug capability of Zynq-7000 AP SoCs combined with the ARM CoreSight technology enables you to debug the processing system (PS) and PL using intrusive and non-intrusive debug methods. Users can debug a complete system, including the PS and PL together. In addition to debugging software, users can debug hardware points in the PS and user-selected hardware points in the PL.

The test and debug capability is based on the ARM CoreSight v1.0 architecture and defines four classes of CoreSight technology components: access and control, trace source, trace link, and trace sink. Also, the Debug Access Port (DAP), which is not a CoreSight component, provides access to CoreSight components and other system features.

Overview

The software tools allow both intrusive and non-intrusive debug. There are break and watch points for intrusive debug. Non-intrusive debug is done using ARMs Program Trace Macrocell (PTM), which captures instruction flow. Flow is compressed by capturing only changes in the program flow. Data can be sent off-chip through 2 to 32 trace output port pins that are part of the MIO/EMIO. The other option is to load the trace data into the Embedded Trace Buffer (ETB) and use third-party tools to upload the trace data through JTAG.

Software-only debug can be done, or the Vivado® Integrated Logic Analyzer (ILA) cores can be added to the design for combined hardware and software debug. The PL JTAG port is daisy chained with the ARM Debug Access Port (DAP). The Vivado logic analyzer cores are accessed through the PL JTAG. ARM and third-party tools attach to the ARM Debug Access Port (DAP).

For more information on PS and PL debug configurations, refer to the "JTAG and DAP Subsystem" (available at this link) and "System Test and Debug" (available at this link) chapters of the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 4].

PS Debug

The Zynq-7000 AP SoC PS provides the following capabilities for system-wide trace:

- Debug and trace visibility of whole systems with a single debugger connection
- Cross triggering support between SoC subsystems
- Multi-source trace in a single stream
- Higher data compression than previous solutions
- Standard programmer's models for tools support
- Automatic topology discovery
- Open interfaces for third party soft cores
- Low pin-count options

PL Debug

Xilinx provides the fabric trace monitor (FTM) for PL test and debug. It is based on the ARM CoreSight architecture, and is a component of the trace source class in the CoreSight system within Zynq-7000 AP SoCs. The FTM receives trace data from the PL and formats it into trace packets to be combined with the trace packets from other trace source components, such as program trace macrocell (PTM) and instrumentation trace macrocell (ITM). With this capability, PL events can be traced simultaneously with PS events. The FTM also supports cross-triggering between the PS and PL, except for the trace dumping feature. Also, the FTM provides general-purpose debug signals between the PS and PL.

The PL test and debug features are:

- ARM CoreSight compliant
- 32-bit trace data from the PL
- 4-bit trace ID from the PL
- Clock domain crossing between the PL and PS
- FIFO buffering for trace packets to absorb PL trace data bursts
- FIFO overflow indication by generating an overflow packet
- Trace packets are compatible with ARM trace port software and hardware
- Trigger signals to and from the PL
- General-purpose I/Os to and from the PL

Software-Only Debug

Embedded software debug is done using the Xilinx® Software Development Kit (SDK). The SDK debug perspective provides a comprehensive debug environment. The debug window shows the session state, including a call stack, source code, disassembled code, process memory, register information, and the XMD console. Breakpoints can be set and execution controlled using standard debugger commands.

A typical application development and debug process overview includes the steps shown in Figure 7-1.

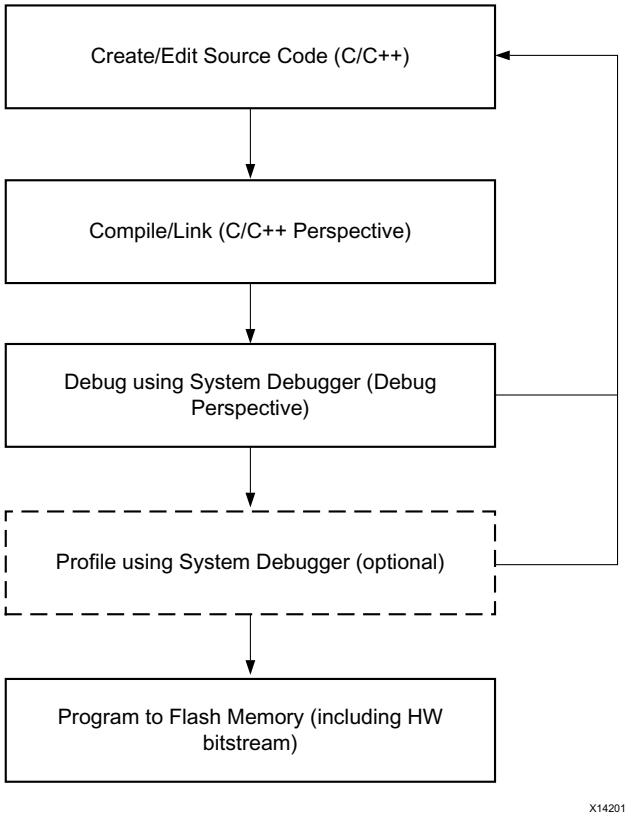

Figure 7-1: **Application Development and Debug Overview**

System Debugger and Target Connections

When debugging applications, the System Debugger connects to the target board through JTAG using the Target Communication Framework (TCF). When debug starts, the System Debugger runs a hardware server agent on the local host. Alternatively, it can run on a remote host when one machine is running the debug application and another machine is connected to the board, such as in a remote lab environment. The System Debugger target-connections window allows management of the board under debug.

System Debugger

The Xilinx System Debugger (XSDB) is the preferred software debug method for Zynq-7000 AP SoCs. System Debugger provides heterogeneous debug of dual ARM and multiple MicroBlaze™ processors in the same session, on either bare metal or Linux-based systems. It also includes hierarchical profiling, ARM NEON library support, and the XSDB command interface. After a board is connected, the status of each CPU in the system can be identified. The debugger also provides a way to attach target applications (`.elf` files) to each CPU, assign symbols for Linux debug, view and set registers, view and set breakpoints, and

monitor memory locations. Commands can be issued to the system through XSDB, and disassembled application code can be viewed using halts and breakpoints.

For more information, including how to connect to a board, refer to the *Xilinx Software Development Kit Help* (UG782) [Ref 6].

Xilinx System Debugger (Command Line)

XSDB is the command interface for loading and controlling CPUs. It can be invoked either from the command line or using an interactive GUI built into the SDK. It includes a Tool Command Language (Tcl) interface that supports scripting of repetitive or complex tasks. XSDB also provides ways to read registers, memory locations, and set code breakpoints. The XSDB supports dynamic help for command usage.

Profiling

The System Debugger in the SDK also supports code profiling that displays function call execution time, so that the most time-consuming parts of an application can be identified. Function calls can be isolated and the called sub-functions profiled, and the user can cross-probe to the source code for reference. Profiling can be used to determine where software applications might be converted into PL or DSP functions for faster system performance.

GNU Tool Chain Support

In addition to System Debugger, the SDK supports application debug using the GNU tool chain, including the GNU Debugger (GDB) and gprof for application profiling.

Drivers and BSP Debug

Software applications must link to or run on top of a software platform using the provided APIs. Therefore, before creating and using software applications in the SDK, a board support package (BSP) must be created. A board support package is a collection of libraries and drivers that forms the lowest layer of an application software stack.

The SDK includes a stand-alone board support package for application development. This package is a simple, semi-hosted, and single-threaded environment that provides basic features, such as standard input/output and access to processor hardware features.

Board support packages and their libraries can be configured using the SDK Board Support Package settings dialog box. Multiple board support packages can be created in the SDK workspace. This allows users to work simultaneously with different board support package configurations, and to re-target applications from one package to another.

Drivers can be used to create applications. The applications and drivers can be debugged using Xilinx System Debugger in stand-alone debug mode or attached to a running target.

The debugger can be configured for each core, and user-selectable options can run initialization files before downloading and running the stand-alone application through JTAG.

For more information, refer to the *Xilinx Software Development Kit Help* (UG782) [Ref 6] and the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7].

OS Debug

Xilinx System Debugger can be used in the attach-to-running-target mode for debugging the operating system and supporting drivers. The System Debugger configurations allow users to add the kernel symbol files and to provide the compilation path mapping to enable source-code debugging. The kernel source must be compiled with following flags set.

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
```

The debugger can also debug code running in PL address space, and the memory view supports PL memory space during debugging.

When debugging using attach-to-running-target mode, the default is that the PL registers will not be accessible. The environment variable HW_SERVER_ALLOW_PL_ACCESS needs to be set through the SDK shell and then the hw_server as follows:

1. Kill any instance of hw_server that is running.
2. Launch a terminal command prompt from the SDK, and set `HW_SERVER_ALLOW_PL_ACCESS`.
3. Launch hw_server from the same shell.

Alternatively, the GDB debugger can also be used for Linux kernel debugging. Like system debugger, GDB is a remote debugger connected to a server that translates between TCP and JTAG. The ARM DS-5 Development Studio also supports Linux kernel debugging for Zynq-7000 AP SoCs.

For more information, refer to the "Attach and Debug Linux Kernel Using Xilinx System Debugger" topic in the *Xilinx Software Development Kit Help* (UG782) [Ref 6].

Linux Application Debug

Developers can debug Linux applications by selecting the System Debugger's Linux application debug mode. The Xilinx SDK also enables local target or remote target debug for Linux applications. Linux application debug is supported by the Xilinx SDK using the TCF agent. A TCF agent process must be running on the Linux target to debug Linux applications using System Debugger.

The GDB debugger can also debug Linux applications. The remote ARM Linux application debugger supports debug by communicating with the gdbserver process running on the

Zynq-7000 AP SoC target. An example System Debugger window is shown in Figure 7-2.

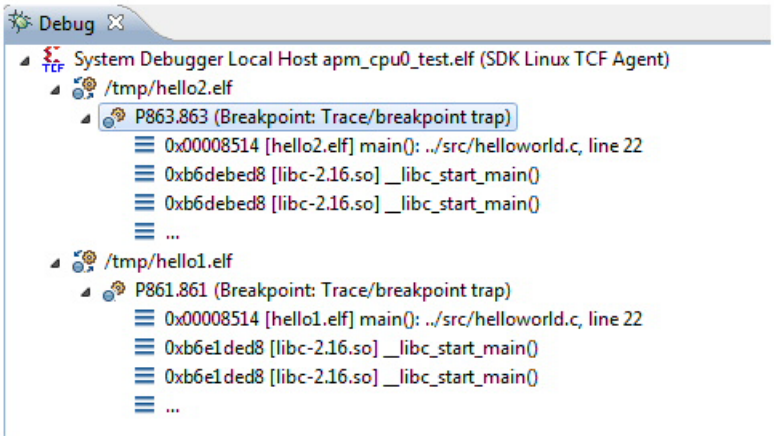

Figure 7-2: **SDK System Debugger Window**

Developers can use IDEs other than the Xilinx SDK, such as the ARM DS-5 Development Studio, for Linux application development and debug. Some developers prefer using a command-line tool when developing Linux applications. Those developers can use tools built in-house or obtained from OS vendors. The Zynq-7000 AP SoC ecosystem is supported by a variety of OS partners. A list of OS partners is found in OS and RTOS Choices, page 142.

Trace

The Cortex-A9 includes a program trace module (PTM) that is compatible with the ARM CoreSight program-flow trace capabilities. This can be used on either Cortex-A9 processor and provides full visibility into the processor instruction flow. The Cortex-A9 PTM has visibility over all code branches and program flow changes with cycle-counting, enabling profile analysis. The PTM block and the CoreSight design kit provides the ability to non-obtrusively trace the execution history of multiple processors. The instruction execution results can be stored in an on-chip buffer, or sent off-chip using a standard trace interface for improved visibility during development and debug.

The ARM DS-5 Development Studio with DSTREAM supports ETB and PTM instruction trace functions for Zynq-7000 AP SoC devices. Lauterbach Trace32 also supports trace data from PTM, FTM, ETB, and AXI Monitor.

For more information on SDK software debug, refer to the following:

- *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 7]
- *Xilinx Software Development Kit Help* (UG782) [Ref 6]

Simulation-Based Debug

Simulation emulates the final design's behavior in a software environment. Simulation helps verify design functionality by injecting stimulus and observing the output result.

System Simulation

Zynq-7000 AP SoCs can be simulated at the RTL level using the Zynq-7000 AP SoC bus functional model (BFM). The BFM supports functional simulation of Zynq-7000 AP SoC applications. It enables functional verification of PL by mimicking the PS-PL interfaces, and the PS OCM/DDR memories. The BFM is delivered as a package of encrypted Verilog modules. The BFM operation is controlled using a sequence of Verilog tasks contained in a Verilog syntax file. For more information on the Zynq-7000 AP SoC BFM, refer to the product specification for the *Zynq-7000 Bus Functional Model, Data Sheet v2.0* (DS897) [Ref 30].

If the design is created using IP integrator, the Vivado tools automatically use the Zynq-7000 AP SoC BFM when the design simulation files are generated. The PL logic is simulated using the IP RTL and the Zynq-7000 AP SoC BFM. A typical use case generates transactions from the PS using GP ports. This can be done using one of the Verilog tasks defined as part of the Zynq-7000 AP SoC BFM.

When using a PL-side DDR memory, the MIG slave interface will be connected to the Zynq-7000 AP SoC BFM with the AXI-interconnect. If a Micron memory module is used, the associated Micron simulation model must be connected to the test bench during design simulation.

Verifying Custom IP

Custom IP packaged using the Vivado tools can be verified using simulation and on hardware. When creating custom IP using the Create and Package IP feature, the **Verify the Peripheral IP using the AXI BFM Simulation Interface** option should be used. Selecting this option ensures that a template IP integrator design is created with AXI BFM slave and master cores, thus simulating the core functionality. The test bench used by this template design uses the API's defined in the AXI BFM IP. For more details on the AXI Bus Functional Model with respect to APIs and response codes, refer to the *LogiCORE IP AXI BFM Cores v5.0 Product Guide (AXI)* (PG129) [Ref 31].

The default test bench template should be modified when testing IP cores generated by the CIP wizard. For more information about the process of mapping HDLs with custom IP templates, refer to this link in the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118) [Ref 22]. Also, if ports are exposed, the template IP integrator design and the template test bench must be modified accordingly before design simulation.

Board Debug

After a design is implemented, it must be tested on hardware. Software debug of hardware can be done using the SDK tool. Debug can be useful in finding problems in the FPGA or fabric side.

FPGA-Only Debug

The PL can be debugged using the Vivado Integrated Logic Analyzer (ILA). The ILA can be included in the RTL code or inserted in the PL after synthesis. It can be configured to monitor design interfaces and nets. A trigger condition can be specified during run-time that exposes a specific problem. Nets marked for debug and connected to the ILA can be monitored to locate the problem source.

Processor-Only Debug

The SDK is used to debug software issues. The SDK allows a user to view the code, view variable values, follow control flow, and set-up and analyze break points.

Before using the SDK, the design must be exported to hardware. The export process creates a design directory containing files from the Vivado tools describing the hardware configuration. The files contain information about all design IP blocks and their memory map. Using this information, the SDK can create the IP drivers. Different software applications can be created in the SDK using the information provided by the export function.

After a software application has been built in the SDK, the FPGA can be programmed using the design bitstream and the application can be launched on the hardware. System Debugger can then be used to debug the software application.

Hardware and Software Co-Debug

Zynq-7000 AP SoCs give you outstanding design potential, allowing you to combine processor function with custom FPGA logic. With this high potential comes greater complexity and the need for a powerful set of tools. Xilinx provides a solution allowing you to simultaneously debug the PS and the peripherals in the PL.

Hardware and software co-debug is done using the Vivado ILA and the Xilinx SDK. The ILA enables nets to be marked for debug and analyzed. The SDK allows setting breakpoints or watchpoints, stepping through program execution, viewing program variables and the stack, and viewing system memory contents. Co-debug is used to debug system-level issues when the software and hardware are combined.

Cross Triggering

Embedded Cross Trigger (ECT) is the cross-triggering mechanism to debug interactions between hardware and software. Using ECT, a CoreSight component can interact with other components by sending and receiving triggers. ECT contains two components:

- **CTM:** Cross Trigger Matrix
- **CTI:** Cross Trigger Interface

Cross triggering can be used to co-debug an application running on the processor and the PL hardware. Vivado Hardware Manager and the SDK are used for this purpose. Cross-trigger ports in the Zynq-7000 AP SoC enable this feature using handshake signals with ILA. There are specific interfaces, TRIGGER_IN and TRIGGER_OUT, for each ARM core. Designers can choose either or both of the ARM cross trigger interfaces at the PS-PL interface. Also, the designer must use the ILA IP in the PL and set cross triggers in the SDK GUI. Processor software can be interrupted when a hardware trigger condition occurs and, similarly, hardware signals can be probed when a software breakpoint occurs. For more information on the procedure to set cross triggers and debug using Vivado, refer to the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) [Ref 15].

Custom IP Hardware Validation

The CIP wizard enables custom IP design template generation in IP integrator that can be debugged on hardware. The *Create and Package IP* feature provides you with the capability to generate an example design connecting JTAG to the custom IP using an AXI Master core. The design incorporates AXI BFM master and slave cores based on the peripheral IP created and stitches them with the peripheral IP. After the corresponding bitstream is programmed on the board, you can modify the generated test vector Tcl file to test the peripheral IP function tested on the board. The test vector file contains a series of writes and reads that are used to validate the IP.

Virtual Platforms

Virtual platforms are software systems that mimic the behavior of hardware. They are fast, functional models of an embedded platform. They execute the same software binaries as hardware and run on traditional desktops, laptops, or servers. Virtual platforms enable system design to be done earlier and at a higher level of abstraction. They enable software development to progress before hardware (board or HDL) is available. They also support unlimited fault injection and test, thereby catching and eliminating bugs in early development phases. Virtual platforms can be easier and faster to modify than hardware because hardware parameters can be changed quickly.

Zynq-7000 AP SoC Virtual Platform

The Zynq-7000 AP SoC virtual platform uses models of PS components found in a typical SoC, including:

- The Cortex-A9 MPCore processor
- Peripherals around the processor, such as UART, Ethernet, USB, CAN, and others
- Component models frequently used in the programmable fabric, such as PCIe technology and video

The virtual platform can connect the models to a variety of interfaces, including Ethernet, USB, serial console, and graphical displays.

The virtual platform enables developers to create and integrate new device models within the Zynq-7000 AP SoC PL. Models can also be created and integrated within the board containing the Zynq-7000 AP SoC. Unlike other solutions that require a hardware implementation of custom IP to be connected to the virtual target, the Xilinx virtual platform can be modified entirely in software to represent a custom hardware design.

Xilinx virtual platforms are beneficial to all stages of product development, from pre-architecture product definition, system design, development, integration, test, to final delivery.

The benefits include:

- Time to Market
  - **Early software development:** Software teams can start developing device drivers and applications prior to hardware availability.
  - **Fast software development:** Unlike simulators for hardware design and verification, the virtual platform runs fast enough to satisfy software developers.
  - **Smarter software development:** Virtual platforms enable software developers to explore new techniques and tricks that can significantly reduce development time.

  - **No retargeting:** Register and instruction-accurate models ensure that the virtual platform will run the same production binaries used by the hardware implementation, simplifying software migration to the physical platform.
  - **No hardware and software dependencies:** The need for developers to share scarce development equipment is eliminated.
- Product Quality
  - **Improved system design:** Concurrently experiment with alternative software and hardware configurations to yield the best system design.
  - **Complete system visibility and control:** Any register and memory location can be monitored and debugged, even at points that are inaccessible on a physical device.
  - **Improved testing iterations:** Software and hardware can be debugged and tested in parallel for more comprehensive test coverage.
  - **Enhanced software regression testing:** Nightly software builds and tests can be run on the virtual platform.
  - **Enhanced Fault injection and Metric driven verification:** A virtual platform enables hardware fault injection, even at points that are inaccessible on a physical device.

Cadence Virtual System Platform

Although traditional virtual platforms enable new development approaches that result in changes in product design, they often fall short when product teams move from a development platform to their product-specific hardware. Extensible virtual platforms offer:

- **Convenient, well-defined model:** The Zynq-7000 SoC virtual platform is out-of-box ready. Software teams can use it immediately to start developing Linux, RTOS, and bare-metal applications.
- **Extensibility:** Xilinx and Cadence enable developers to extend the virtual platform with new devices (instantiated within the Zynq-7000 SoC programmable fabric or on the board) or system models. The extensible virtual platform has the flexibility and configurability of physical hardware. Using Cadence tools, it supports models written in C or System-C, as well as models written in SystemVerilog or VHDL.

SDSoC Environment

This chapter covers the following information about designing in the SDSoC environment:

- Overall Usage Flow, page 216: You should understand the overall usage flow for using the SDSoc design environment, as well as supported OSs and hardware platforms.
- Profiling, page 218: SDSoC integrates profiling capabilities on top of the ones available in Xilinx® SDK, allowing you to profile the application without code instrumentation.
- Performance Estimation, page 219: The SDSoC environment provides tools to quickly estimate the performance improvement that can be expected when you identify functions for hardware acceleration.
- Generating and Running a Complete Software-Hardware System, page 219: After running the performance estimation flow, you can rebuild the system and see the actual performance of the generated system.
- Optimizing Performance Using C-Callable RTL IP Library, page 219: You can use the C-Callable RTL IP library to optimize performance.
- Optimizing IP Performance Using HLS, page 220: You can use Vivado high-level synthesis (HLS) to compile application code that is sent to hardware in the SDSoC environment, and then improve code using common strategies used by the SDSoC tool.
- Optimizing System Performance, page 220: With SDSoC, you can use various techniques to optimize your system performance.
- Debugging the System, page 222: SDSoC allows projects to be created and debugged using the SDSoC environment. SDSoC shares the debugging infrastructure and methodology with the Xilinx SDK System Debugger.
- Performance Measurement and Analysis, page 223: SDSoC shares many of the advanced system-level performance measurement and analysis capabilities with Xilinx SDK. It provides optional functions for measuring performance and for identifying performance bottlenecks.
- Expert Use Models, page 223: Expert users can override design decisions made by SDSoC.

Introduction

SDSoC provides a C/C++ programming experience with an easy-to-use Eclipse-based development environment, including design tools for Zynq®-7000AP SoC development by software engineers and system architects. SDSoC includes a full-system optimizing C/C++ compiler that provides automated software acceleration in programmable logic combined with automated system connectivity generation between the software and the hardware.

The SDSoC programming model is designed to be intuitive to software engineers. An application is written as C/C++ code, with the programmer identifying a target platform and a subset of the functions within the application to compile into hardware. The SDSoC C/C++ compiler and linker then compile the application into hardware and software to realize the complete embedded system implemented on a Zynq device, including a complete boot image with firmware, operating system, and application executable. Figure 8-1 shows the complete system flow.

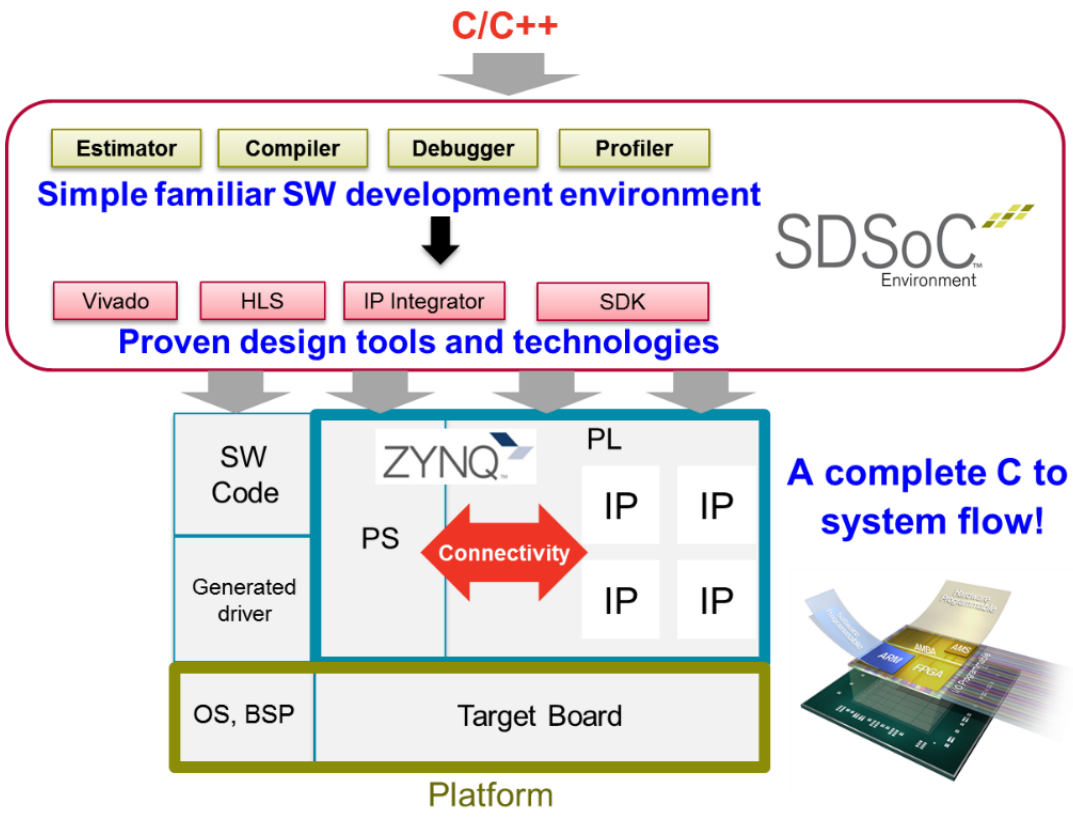

Figure 8-1: **Complete C/C++ Application to Zynq System Flow with SDSoC**

SDSoC unifies the hardware and software design flows presented in this guide into a single C/C++ based flow and automates many of the time-consuming tasks, especially the system connectivity generation and the hardware-software integration. Figure 8-2 illustrates the

difference between the traditional hardware-software design flow and the design flow using SDSoC. By automating many of the design tasks, SDSoC enables a software engineer and system architect to quickly implement an application and perform rapid design iteration.

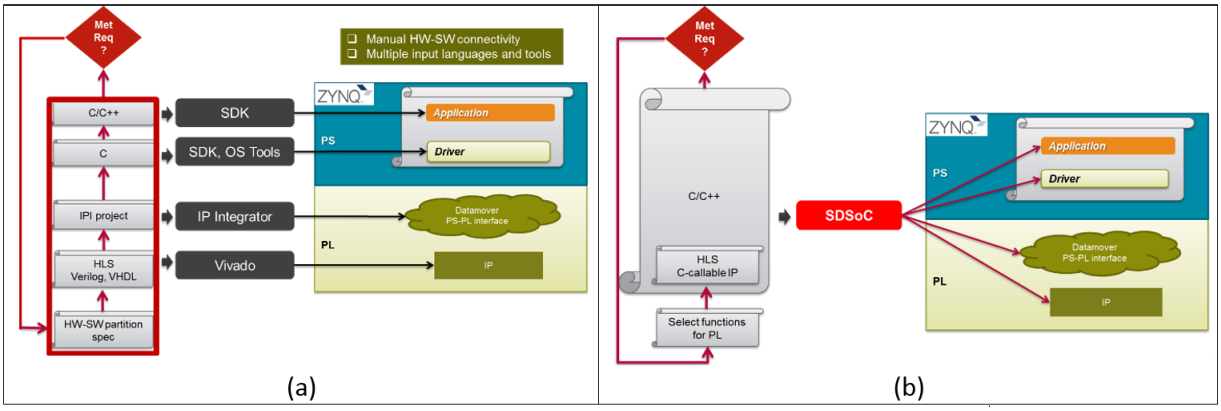

Figure 8-2: The Traditional Design Flow(a) and the Design Flow with SDSoC (b)

The hardware/software interface is defined by the interactions and data flow among functions mapped to hardware and the rest of the program. Each function mapped to hardware runs as an independent thread, but the compiler ensures that the program semantics are preserved. Rather than recoding an application to invoke device drivers for each hardware function, SDSoC compiler automatically maps the original application onto the generated hardware system using its own library of efficient data movers to communicate data between the processing system and the programmable logic.

The easiest way to understand the mapping is to realize that the SDSoC compiler analyzes the program structure and its data flow and generates a hardware and software system that efficiently implements the program's data flow, including optimizations to communicate data directly from one hardware block to another, rather than always going through the processing system main memory.

Applications can link with C-callable IP libraries to support IP reuse. When application code links with C-callable libraries provided as part of the platform, the interface to these libraries essentially encapsulates the platform-specific portions of the application. In this way, applications can achieve considerable portability across platforms.

Overall Usage Flow

Figure 8-3 shows the overall usage flow using the SDSoC design environment. For a given user application, the first step is to identify the compute-intensive parts of the application using profiling or by other means. These parts should either be already present or refactored into functions. In the SDSoC design environment, these functions can then be assigned to hardware. SDSoC supports a fast performance estimation without generating

full bit-stream to enable rapid design optimization. Based on the performance estimation, optimizations can be done with the following goals:

- To migrate more computation to hardware
- To optimize the performance of functions moved to hardware; for example, using high-level synthesis (HLS) pragmas
- To optimize the data transfer performance between the processing system and programmable logic; for example, using different data movers or different memory allocation

After achieving the desired performance, SDSoC can generate the SD card image for the full system to run the application on the target platform. Based on running the generated system, further optimizations can be done.

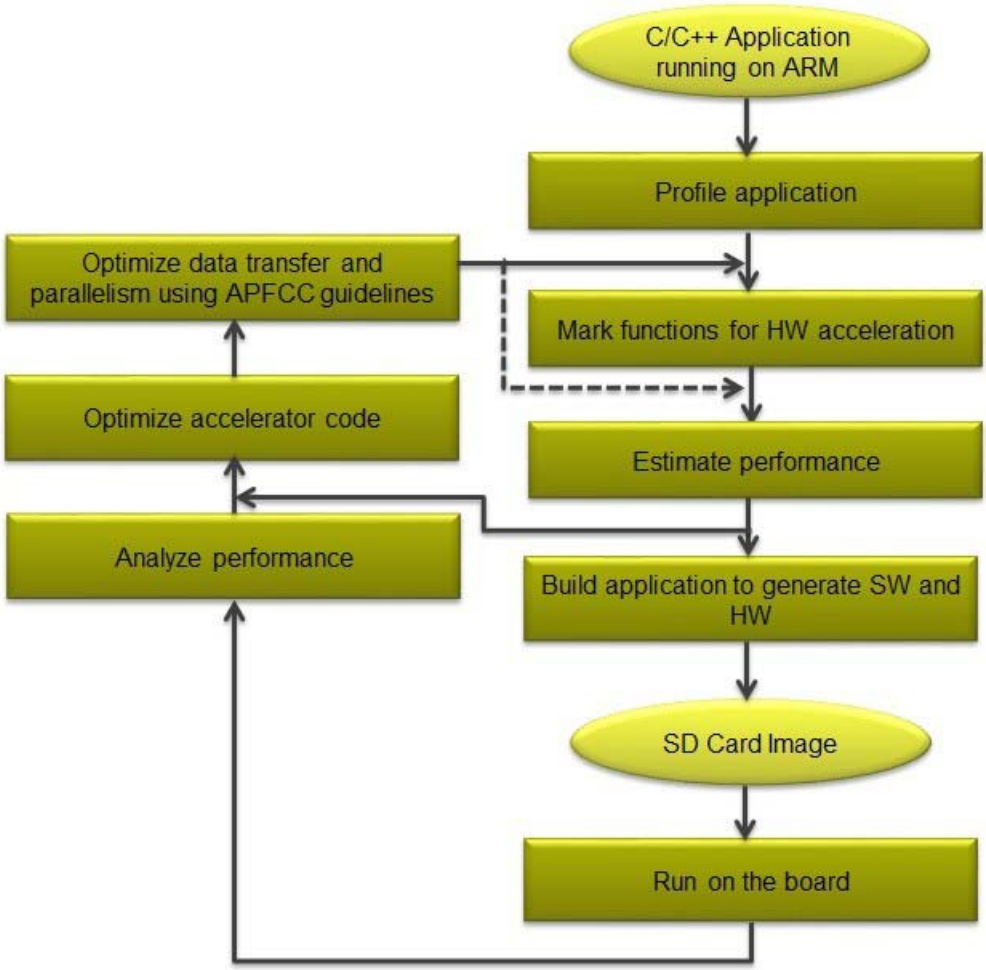

Figure 8-3: SDSoC User Flow

By default, SDSoC targets the Linux operating system. The SD card image that is generated includes the Linux kernel, file system, and user application. SDSoC also supports FreeRTOS and a standalone (bare metal) mode as choices of OS. For FreeRTOS and standalone, the user application is compiled with support from Xilinx Standalone system (LibXil). For more details on options to control the OS choice, refer to *Introduction to SDSoC* (UG1027) [Ref 25] and *SDSoC Getting Started Guide* (UG1028) [Ref 26].

SDSoC comes with support for a few hardware platforms out of the box. This list of supported platforms includes popular boards such as ZC702, ZC706, Zed board, and MicroZed. You can select the target platform for an application either in SDK while setting up the project, or by specifying the `-sds-pf` command line option within a Makefile. You can view the list of all available platforms by running the command line tool sdscc with the option `-sds-pf-list`.

SDSoC also allows users and third-parties to generate platforms starting with their own Vivado designs. Such platforms can be selected by pointing to the top-level platform directory either in SDK or the command line. For more details on creating SDSoC platforms, refer to *SDSoC Platforms and Libraries Guide* (UG1146) [Ref 27].

Profiling

The importance of performance analysis and profiling has been well documented in previous chapters. SDSoC integrates those principles through hooks and features inside the tool. Profiling capabilities build on top of the ones in Xilinx SDK, and therefore, application developers can use the non-intrusive TCF profiler to profile the application without any code instrumentation. Based on the profile information, a first pass at moving functions to hardware can be started. For more details, refer to Chapter 6, Software Design Flow.

SDSoC also supports the standard Eclipse based tools, such as gprof, for profiling the user application; refer to Chapter 7, Debug for references. An SDSoC user can modify the build properties of the application and select the `-pg` and `-g1` options to obtain the profiling information. Once these options are selected, the user can build the project and execute it on the board to obtain the profile information file (`gmon.out`). The SDSoC environment includes the standard Eclipse-based graphical view of the `gmon.out` file, which shows a hierarchical view of functions with the percentage of time spent in each of them.

Performance Estimation

After functions have been identified for hardware acceleration, the SDSoC environment provides tools to quickly estimate the performance improvement that can be expected by these choices. For example, a full build cycle for a typical application might take 40 minutes, whereas the performance estimation flow can give an estimate of the application speedup in the order of minutes. The performance estimation flow can be run by changing the build configuration to "SDEstimate" and building the project. More details on this flow can be found in the *Introduction to SDSoC* (UG1027) [Ref 25] and Lab 4 in the *SDSoC Getting Started Guide* (UG1028) [Ref 26].

Generating and Running a Complete Software-Hardware System

Once you are satisfied with the estimated performance obtained by performing iterations using the performance estimation flow, you can change the build configuration to "SDRelease" and rebuild the project. You can also copy the generated SD card image to an SD card and run on the board to see the actual performance of the generated system.

Optimizing Performance Using C-Callable RTL IP Library

As described in Introduction, page 215, the data flow defined by a software application can be automatically mapped onto hardware and software in an application-specific SoC. The SDSoC compiler maps and schedules operations onto hardware functions based on program data flow. An application can also link to a library that provides a function call mapping onto an IP block defined in a hardware description language like Verilog. From the point of view of the application code and the SDSoC compiler, the scheduling and mapping are identical whether the function is synthesized at compile time using HLS or mapped onto an HDL IP block. Any call to a hardware function will be mapped into hardware by the SDSoC compiler.

A traditional "software driver" approach to targeting an IP block peripheral typically exposes only the control interface; the hardware system captures the data flow. Using a full-system compiler like SDSoC, application code can instead directly invoke the data processing functions as it would any other functional library. The data transport, such as using DMA, is abstracted and automatically invoked to implement the function call efficiently.

For more information, refer to the "SDSoC Libraries" chapter in the *SDSoC Platforms and Libraries Guide* (UG1146) [Ref 27].

Optimizing IP Performance Using HLS

The application code that is sent to hardware in the SDSoC environment is compiled through the Vivado high-level synthesis (HLS) compiler. This is described in Chapter 5, Hardware Design Flow. Several optimization strategies are presented in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 9]. Some common strategies to improve code are used by the SDSoC tool, such as adding a pragma for enabling pipelining before a compute intensive loop and increasing the memory bandwidth for the inner loop by specifying the array_partition pragma for the variables accessed inside the inner loop. For more high-level details for optimization strategies, refer to *Introduction to SDSoC* (UG1027) [Ref 25].

Optimizing System Performance

System performance improvements through acceleration is one of the key principles integrated in the SDSoC. Chapter 2, System Level Considerations and Chapter 3, Hardware Design Considerations touched upon the different ways to improve performance from both Software and Hardware perspective. This section describes some additional techniques used by SDSoC and how the it implements some of the previously described techniques such as storing data in system memory and data pipelining.

Using Physically Contiguous Memory

The SDSoC system-optimizing compiler incorporates memory allocation trade-offs in software and hardware functions. On Linux based systems, memory allocated by `malloc()` is paged. The kernel keeps track of the pages for the user and presents a unified view to the user application. However, when memory allocated by `malloc()` is transferred to the accelerator, there is a performance penalty to collect individual pages and transfer them using the scatter-gather mode in the AXI-DMA. SDSoC provides a user-accessible function `sds_alloc()`, which behaves like `malloc()` but allocates physically contiguous memory. This results in faster performance when using data movers like scatter-gather DMA, but it also allows SDSoC to choose data movers like simple-mode of the AXI-DMA with significantly better performance due to lower overhead.

Using Local Caches and Streaming Data

The data transferred to the code compiled through Vivado HLS can be sent once to reside in BRAMs in the programmable logic for the duration of the computation, if the amount of data is small. If the data is too large to fit in BRAMs in the programmable logic, or if it is not used multiple times during the computation, then it can be sent as a datastream to the generated accelerator. You can control these choices for caching or streaming data being transferred between the processor and accelerator, as detailed in *Introduction to SDSoC* (UG1027) [Ref 25].

Using Shared Memory

Instead of transferring data from the processor's memory to the accelerator's local memory or streaming it through the accelerator, it is also possible for the processor and accelerator to access a common region of shared memory. You can achieve this by adding code pragmas to the function that is mapped to hardware. Using this technique requires the user application to allocate the memory using `sds_alloc()` instead of `malloc()`. More details about this can be found in chapter 10 of *Introduction to SDSoC* (UG1027) [Ref 25].

Using Direct Accelerator-Accelerator Data Transfers

If multiple functions are selected for hardware acceleration, the SDSoC environment can automatically pipeline them. This means that it passes the data transferred between them directly in hardware, without copying it to the processor first, so long as there is no use on the processor side.

Using Multiple Instances of the Same Hardware Function

Multiple calls from a single source location to a function mapped to hardware are mapped to a single accelerator. However, SDSoC provides an `async` code pragma, which directs the compiler to generate multiple accelerator instances to improve the performance of the application by processing more data in parallel.

Pipelining Data Communication with Computation

Multiple successive calls to an accelerator can be pipelined by the SDSoC compiler under user pragma control. This allows the overlap of accelerator computation with the data transfer of the subsequent call to the accelerator. Consider Figure 8-4, page 222, which shows sequential calls to an accelerator with two inputs and one output.

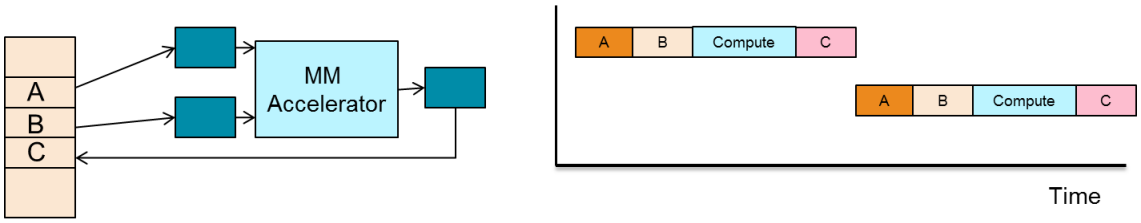

Figure 8-4: **Sequential execution of calls to the accelerator**

Figure 8-5 shows two calls to the same accelerator executing in a pipelined fashion. Multiple buffers, automatically inserted by the compiler, are used so that the data transfer for the second call can start as soon as the data transfer for the first call is done. This is achieved in the SDSoC environment by using the `async` pragma as described in Chapter 7 of *Introduction to SDSoC* (UG1027) [Ref 25].

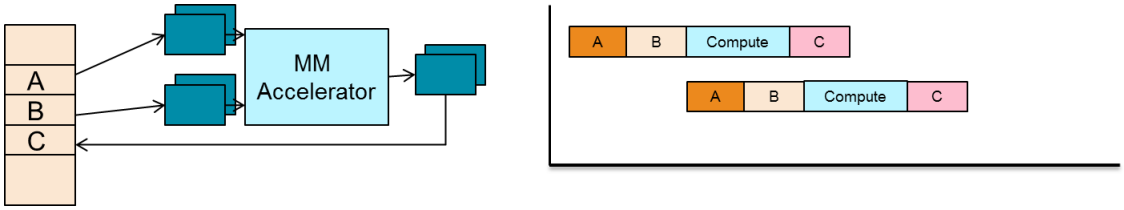

Figure 8-5: **Pipelined execution of calls to the accelerator**

Debugging the System

SDSoC shares the debugging infrastructure and methodology with the Xilinx SDK System Debugger, including enhancements to support debugging applications for the standalone platform and having the ability to look into IP registers in the physical address space. Refer to Chapter 6, Software Design Flow.

SDSoC allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE using user-created Makefiles and debugged either on the command line or using the SDSoC IDE. For details about debugging, refer to *Introduction to SDSoC* (UG1027) [Ref 25] and the *SDSoC Getting Started Guide* (UG1028) [Ref 26].

Performance Measurement and Analysis

For simple performance measurement and to identify performance bottlenecks, SDSoC provides the function `sds_clock_counter()`, which returns a cycle-accurate timestamp from a free-running counter in hardware. You can use this counter to instrument your code and collect performance data. You also have access to Vivado_HLS reports under the `_sds` folder in the build directory, from which you can gain more insight about the generated hardware and changes you could make to improve performance.

SDSoC shares many of the advanced system-level performance measurement and analysis capabilities with Xilinx SDK; refer to Chapter 6, Software Design Flow. The SDSoC compiler optionally inserts an AXI Performance Monitor (APM) into the generated system to monitor traffic between the Zynq device Programming Logic (PL) and Processing System (PS). By observing activity at the PS/PL boundary using the APM, a designer can determine whether a system is saturating the capacity of a particular resource, such as the cache-coherent ACP port.

A hardware function observing an idle data transport interface for long periods between computations might indicate a compute-bound system that would benefit from accelerator micro-architecture optimization to increase throughput and decrease latency. Conversely, a data transport interface that is saturated might indicate a situation where traffic can be redirected to other available transport interfaces on the PS. For more information, refer to the *Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis* (UG1145) [Ref 24].

Expert Use Models

You can override every design decision made by SDSoC as needed, either by inserting pragmas into application code or by using IDE/command-line parameters.

Specific controls for more advanced users provide the following capabilities.

- Data mover selection: To control the mapping of data transfers on to different IP blocks, such as simple mode DMA, scatter-gather DMA, and 2-D DMA.
- Zynq-7000AP SoC processing system interfaces to CPU and DDR selection: To control the mapping onto cache-coherent or high-throughput AXI interfaces.
- Hardware function "port" interface selections: To control the hardware interfaces on hardware accelerator; for example, as "shared memory" between hardware function and controlling software thread, or as "copy-in, copy-out" for caching data in local BRAMs.
- Run-time variable size data transfers: For transfers that obey "copy-in, copy-out" semantics, limiting transfer sizes when possible can improve system performance.

- Hardware function microarchitecture using Vivado HLS: For compute-bound systems, you can employ Vivado HLS pragmas and directives to improve throughput and latency.
- Blocking vs. non-blocking function invocation: By default, functions preserve the original function call semantics, but you can instruct the compiler, using pragmas in the code, to return control "immediately", such as, to implement software task pipelining.
- Mapping subsets of hardware functions into different bitstreams that the application will time-multiplex during run-time: To time multiplex the programmable logic fabric to realize more hardware functions than can fit in the target device concurrently.

For more details regarding each of these controls, refer to the “SDSoC Pragma Specification” section in *Introduction to SDSoC* (UG1027) [Ref 25].

Advanced users, hardware designers, and system software teams might want to build their own SDSoC platform, including partial hardware design and software components like OS, drivers, file system, and libraries. They can also package their own HDL IP as C-callable IP libraries. There is a well-defined, low-overhead methodology for exporting a Vivado hardware system to support SDSoC, as well as to create C-callable libraries for HDL IPs. For more information, refer to the *SDSoC Platforms and Libraries Guide* (UG1146) [Ref 27].

SDSoC is a tool that brings a lot of techniques described in earlier chapters under one umbrella to aid you in effectively architecting, designing, and debugging a system. It provides a means to implement methodology principles described earlier and also highlights new principles that become visible when seen under one umbrella. The biggest benefits of the tool are reaped by the Software designer. This is a must have in a Software designer's toolbox to accelerate development of systems with hardware components.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Xilinx Documentation Navigator

The Xilinx Documentation Navigator is a free tool that you can use to access documentation while using Xilinx products. The Documentation Navigator is available as part of the Vivado Installer. When it is installed on your system, you can access it by going to **Start > Programs > Xilinx Design Tools > DocNav** and clicking the **DocNav** icon.

For detailed information about using the Xilinx Documentation Navigator, refer to this link in *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 11].

Related Design Hubs

Design hubs provide quick access to documentation, training, and information for specific design tasks. The following design hubs are applicable to embedded development and the methods described in this guide:

- Partial Reconfiguration Design Hub
- Software Development Kit (SDK) Design Hub

References

The references below provide additional information.

Xilinx User and Reference Guides

1. *7 Series FPGAs Clocking Resources* (UG472)
2. *7 Series FPGAs GTX/GTH Transceivers* (UG476)
3. *7 Series FPGAs GTP Transceivers* (UG482)
4. *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585)
5. *OS and Libraries Document Collection* (UG643)
6. *Xilinx Software Development Kit Help* (UG782)
7. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)
8. *ZC702 Evaluation Board for the Zynq-7000 XC7X020 All Programmable SoC User Guide* (UG850)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
11. *Vivado Design Suite User Guide: Getting Started* (UG910)
12. *AXI Interface Based KC705 Embedded Kit MicroBlaze Processor Subsystem Software Tutorial* (UG915)
13. *Zynq-7000 All Programmable SoC ZC702 Base Targeted Reference Design (Vivado Design Suite 2014.2) User Guide* (UG925)
14. Zynq-7000 All Programmable SoC PCB Design Guide (UG933)
15. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940)
16. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
17. *PetaLinux Tools User Guide: Zynq All Programmable SoC Linux-FreeRTOS AMP Guide* (UG978)
18. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
19. *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC User Guide* (UG1019)
20. *Zynq-7000 All Programmable SoC Secure Boot Getting Started Guide* (UG1025)
21. *Embedded System Tools Reference Manual* (UG1043)
22. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

23. *Generating Basic Software Platforms Reference Guide* (UG1138)
24. *Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis* (UG1145)

SDSoC Documentation

The following SDSoC documents are currently available on a limited access basis. To request access to these files, contact your Xilinx sales representative.

25. *Introduction to SDSoC* (UG1027)
26. *SDSoC Getting Started Guide* (UG1028)
27. *SDSoC Platforms and Libraries Guide* (UG1146)

Other Xilinx References

Data Sheets

28. Zynq-7000 All Programmable SoC (Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics (DS187)
29. *Zynq-7000 All Programmable SoC Overview* (DS190)
30. *Zynq-7000 Bus Functional Model, Data Sheet v2.0* (DS897)

Product Guides

31. *LogiCORE IP AXI BFM Cores v5.0 Product Guide (AXI)* (PG129)

White Papers

32. *Get Smart about Reset: Think Local, Not Global Whitepaper* (WP272)
33. Secure Boot in the Zynq-7000 All Programmable SoC (WP426)

Application Notes

34. *Designing High-Performance Video Systems with the Zynq-7000 All Programmable SoC* (XAPP792)
35. *LightWeight IP (lwIP) Application Examples v4.0* (XAPP1026)
36. *Simple AMP Running Linux and Bare-Metal System on Both Zynq-7000 AP SoC Processors* (XAPP1078)
37. *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors* (XAPP1079)

38. *PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 AP SoC* (XAPP1082)
39. *Using the Zynq-7000 Processing System (PS) to Xilinx Analog to Digital Converter (XADC) Dedicated Interface to Implement System Monitoring and External Channel Measurements* (XAPP1172)
40. *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175)
41. *System Monitoring Using the Zynq-7000 AP SoC Processing System with the XADC AXI Interface* (XAPP1182)
42. *Zynq-7000 Platform Software Development Using the ARM DS-5 Toolchain* (XAPP1185)

Web Pages

43. AXI Performance Monitor web page
44. AXI Timer/Counter web page
45. LogiCORE™ AXI Traffic Generator web page
46. Xilinx Security Solutions web page
47. Xilinx Zynq-7000 AP SoC Ecosystem web page
48. Xilinx PetaLinux Tools web page

Wiki Pages

49. K7 Embedded TRD 2013.2 wiki page
50. Xilinx Linux Drivers wiki page
51. Xilinx Linux GPIO driver wiki page
52. Xilinx Linux I2C driver wiki page
53. Xilinx Linux SPI driver wiki page
54. Xilinx Linux wiki page
55. Xilinx Multi-OS Support wiki page
56. Xilinx PetaLinux wiki page
57. Xilinx U-Boot wiki page
58. Xilinx Yocto wiki page
59. Zynq-7000 AP SoC Boot - Booting and Running Without External Memory Tech Tip wiki page
60. Zynq-7000 AP SoC Boot - Locking and Executing out of L2 Cache Tech Tip wiki page

61. Zynq-7000 AP SoC Low Power Techniques part 1 - Installing and Running the Power Demo Tech Tip wiki page
62. Xilinx Zynq-7000 AP SoC Spectrum Analyzer part 2-Building ARM NEON Library Tech Tip
63. Xilinx Zynq Linux wiki page
64. Xilinx Zynq Power Management wiki page

Answer Records

65. Xilinx Answer Record 46778
66. Xilinx Answer Record 47511
67. Xilinx Answer Record 50991
68. Xilinx Answer Record 52847
69. Xilinx Answer Record 55572
70. Xilinx Answer Record 57744
71. Xilinx Answer Record 58387

Information from Other Companies

72. ARM DS-5 Development Studio Streamline Performance Analyzer documentation: http://ds.arm.com/ds-5/optimize
73. ARM Infocenter Accelerator Coherency Port web page: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407e/CACGGBCF.html
74. *ARM Security Technology: Building a Secure System using TrustZone® Technology*: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
75. *Cortex-A9 Technical Reference Manual*: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e/DDI0388E_cortex_a9_r2p0_trm.pdf
76. *Design Calculations for Robust I2C Communications:* http://www.edn.com/design/analog/4371297/Design-calculations-for-robust-I2C-communications
77. Discretix: http://www.discretix.com/
78. ENEA Software AB Solutions: http://www.enea.com/solutions/
79. eSOL Embedded Engineering and Enabling Solution: http://www.esol.com/
80. Express Logic ThreadX RTOS: http://rtos.com/products/threadx/xilinx_arm
81. GreenHills INTEGRITY RTOS: http://www.ghs.com/products/xilinx_zynq.html

82. *How a MicroBlaze can peaceably coexist with the Zynq-7000 AP SoC*: http://www.eetimes.com/document.asp?doc_id=1280680
83. IAR Integrated Solutions Partner Program: http://www.iar.com/Products/RTOS/Integrated-RTOSes/
84. iVeia Adeneo Embedded Board Support Packages: http://www.adeneo-embedded.com/Products/Board-Support-Packages
85. Mentor Graphics Nucleus RTOS: http://www.mentor.com/embedded-software/nucleus/?sfm=auto_suggest
86. Micrium µC/OS RTOS: http://micrium.com/products/
87. MontaVista Carrier Grade Edition 7: http://www.mvista.com/product-carrier-grade-edition7.html
88. *PCA9548A Low Voltage 8-Channel I2C Switch With Reset*: http://www.ti.com/lit/ds/symlink/pca9548a.pdf
89. SD Association, Latest Simplified Specifications: https://www.sdcard.org/downloads/pls/simplified_specs/
90. Open Kernel Labs OKL4 Microvisor: http://www.ok-labs.com/products/okl4-microvisor
91. QNX Neutrino RTOS: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html#overview|
92. Quadros RTOS: http://www.quadros.com/products/operating-systems
93. Real Time Engineers FreeRTOS Interactive webpage: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/entries/23277857-Updated-Xilinx-FreeRTOS-port-for-Zynq-to-SDK-14-4-release
94. Sciopta RTOS: http://www.sciopta.com/news/ZYNQ-7000.html
95. Sierraware Hypervisor and SierraTEE Trusted Execution Environment: http://www.sierraware.com/arm_hypervisor.html
96. SYSGO PikeOS BSPs for ZC702, Zed, and Zynq-7000 All Programmable SoC BSP: http://www.sysgo.com/products/board-support-packages/pikeos-bsp-list/
97. SYSGO PikeOS Product Information: http://www.sysgo.com/news-events/press/press/details/article/sysgos-certified-pikeos-supports-xilinxs-zynq-7000-all-programmable-soc/
98. The Official Linux Kernel from Xilinx webpage: https://github.com/Xilinx/linux-xlnx
99. Timesys LinuxLink: http://www.timesys.com/embedded-linux/linuxlink
100. Wind River VxWorks, Linux, and Workbench IDE: http://www.windriver.com/
101. *Xilinx XC702 DS-5 Getting Connected Guide:* http://www.arm.com/files/pdf/zc702_ds5_2.pdf
102. Yocto Project webpage: http://git.yoctoproject.org/cgit/cgit.cgi/meta-xilinx/about/

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related videos:

1. Vivado Design Suite QuickTake Video: How to Create a Zynq Boot Image Using Xilinx SDK
2. Vivado Design Suite QuickTake Video Tutorials

Please Read: Important Legal Notices