WP262 (v2.0) November 21, 2007

Designing Multiprocessor Systems in Platform Studio

By: Vasanth Asokan.

Embedded processing requirements are growing at a rapid pace and system architects are turning towards multiprocessor designs to solve the problems of burgeoning complexity and inadequateness found in a uniprocessor system. The advent of FPGAs with high logic density and high performance hard blocks, have made powerful chip multiprocessing (CMP) solutions a reality. The real challenge now lies in the rapid exploration and creation of designs in this solution space. The Xilinx Platform Studio (XPS) and Embedded Development Kit (EDK) is a comprehensive solution for designing embedded programmable systems. Platform Studio tools and IP make it very easy to design powerful CMP systems. They provide the flexibility to create uniquely crafted, customized solutions on FPGA logic real estate that can meet both price and performance targets. This white paper describes various multiprocessing hardware and software concepts in Platform Studio, as they apply to Xilinx solutions based on the PowerPC™ and MicroBlaze™ embedded processors.

Introduction

There are various factors which can require a system to be designed using more than one processor. In most cases, it is very simply performance. In others, it is functionality, modularity, and such concerns. Following is a broad summary of the typical scenarios.

Multiple Independent Functions

The design may have multiple, independent set of processing tasks to be performed. An attractive way to solve the problem could just mean creating various processing modules that are completely independent, each dedicated to its own processing task. Each processing module is assigned a unique processor and peripheral set.

Control Plane Offload

A common scenario in a system is the presence of a clearly distinct set of real-time and non real-time tasks, such that a solution based on a single processor to handle both may cease to be responsive. In these cases, a slave processor is dedicated to perform the real-time control task in a timely fashion. Other regular and non-special tasks are left to be performed by a master processor which also usually serves as an interface to the host system. The master processor monitors the slave and occasionally sends control commands and may also send or receive data.

Data Plane Offload

Another common design scenario is the presence of intensive number crunching or protocol processing tasks in conjunction with more regular end applications. In these cases, a slave processor is used to offload the data intensive or number crunching tasks, while the master processor performs overall co-ordination, setting up of computations and host interface. The slave processor may contain specialized functions or interfaces to allow it to meet computation performance requirements. Some examples of this scenario include, network offload, streaming media processing, security algorithm, etc.

Interface Processing

On a system which acts as a bridge or switch between multiple interfaces, a slave processor can be dedicated to the processing of data at each interface, while one or more master processors perform the higher level bridging and switching tasks. This is a typical network processing design.

Stream processing

For handling stream-oriented computation, processors may be arranged to act upon the data stream in a pipeline fashion. Each stage in the multiprocessor pipeline, acts upon one portion of the computation before passing it on to the next processor. All the processors act as peers. This solution is typically used to increase the throughput of a solution.

Symmetric Processing

In certain cases, a single processor may just not provide enough performance and it may be hard to find clean boundaries at which the solution can be partitioned across multiple processors. Traditional Symmetric Multiprocessing (SMP) is a useful solution in which the performance of an application is scaled up by adding more processors.

An OS layer manages parallel tasks and automatically schedules them across multiple processors. Linux is an example of an OS that supports SMP.

Reliability and redundancy

A processing system may be replicated multiple times to provide for reliability and redundancy. Triple Mode Redundancy (TMR) is a related concept. TMR is outside the scope of this paper. Please refer to
http://www.xilinx.com/ise/optional_prod/tmrtool.htm

Apart from the SMP scenario (due to lack of cache coherency support), all the other scenarios are feasible on Xilinx FPGAs with Platform Studio tools. The unique capability of Xilinx processing solutions is the flexibility to customize each of the processing subsystems to the application needs. For example, not all processors may need a cache, or a floating point unit. By assigning specific functions to specific processors, a tailored solution that meets all design goals can be created.

Design Flow

The topology and use model of a multiprocessor system is dictated and constrained by the various requirements of the end application. The constraints could arise equally from either hardware considerations or software considerations. For e.g., running Linux might be a necessary software requirement, while responding to a controller in a real-time fashion or running the system at a frequency higher than 66 MHz might be hardware requirements. Exploring the huge design space offered by FPGA logic, tools and IP, provides the required flexibility to the system designer. In most cases, the architecture of the system follows very logically from the constraints placed. In a few cases, there may be more than one way to architect the solution.

A typical design flow is shown in Figure 1. The designer(s) takes the application requirements, applies various constraints and trade-offs and designs a solution with some hardware and software architecture. He then creates a prototype and takes it to the FPGA using Platform Studio tools. Qualifying and refining of the solution then follows, through multiple iterations of profiling and verification. Trade-offs and redesign are applied during this stage. The power of the FPGA and Platform Studio tools, make this iterative process very fast. Designing the original design requires a few hours, but making refinements can be done in a matter of minutes – all the while having working hardware generated for the reconfigurable platform. Adding or removing a processor to the system is as easy as clicking a few buttons on a GUI. The powerful simulation, debug, and profiling infrastructure, offered by Platform Studio help guide design space exploration.

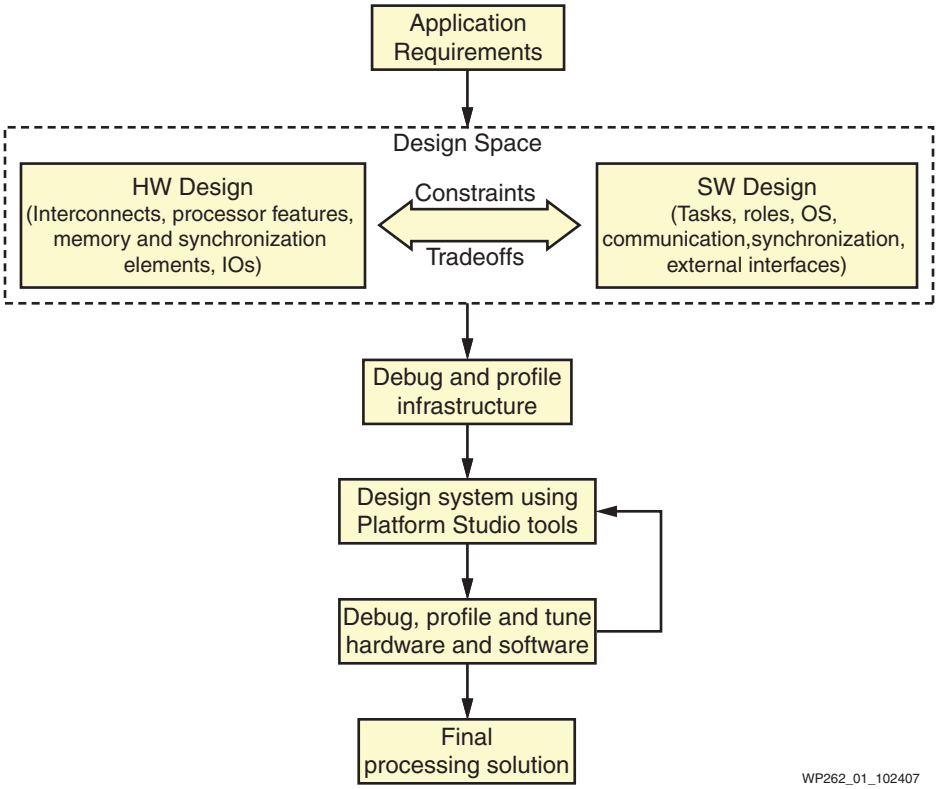

Figure 1: **System Design Flow**

Background

The Xilinx Embedded Development Kit (EDK) bundle is an integrated software solution for designing embedded processing systems. This pre-configured kit includes the award winning Platform Studio tool suite as well as all the IP and documentation required for designing Xilinx Platform FPGAs with embedded PowerPC hard processor cores and/or MicroBlaze soft processor cores.

Virtex-II Pro™ and Virtex-4 FX FPGA devices provide embedded PowerPC 405 (PPC405) hard core based on the IBM PowerPC processor family. The PowerPC processor is capable of achieving 400 MHz and 600+ DMIPS performance. It is a RISC core (32-bit Harvard architecture) with a 5-stage pipeline. It contains large, 16 KB, 2-way set-associative instruction and data caches. It also provides a Memory Management Unit (MMU) that enables robust RTOS implementations. It supports enhanced instruction and data On-Chip Memory (OCM) controllers that interface directly to embedded Block RAM memory.

The MicroBlaze processor core is a 32-bit Harvard RISC architecture with a rich instruction set optimized for embedded applications. The processor is a soft core, meaning that it is implemented using general logic primitives rather than a hard, dedicated block in the FPGA. The MicroBlaze solution gives the user control of a number of features such as the cache sizes, interfaces, and execution units such as a hardware floating point unit. The configurability allows the user to trade-off features for size, in order to achieve the necessary performance for the target application at the lowest possible cost point. MicroBlaze also features an optional MMU and thus supports a variety of RTOS implementations.

Hardware Design

This section describes the hardware design considerations in multiprocessor systems.

Overall Architecture

The number of topologies that are possible with multiple processors in them is large. Not only can the number of processors be different, but there are a lot of possibilities with regards to their arrangement and function. The presence or absence of various elements such as external memory and peripherals can also open up various options for the system architecture.

The busing paradigm used by each processing subsystem forms the core of any system architecture. Xilinx processors primarily adhere to the PLBv4.6 CoreConnect specification from IBM. PLBv46 is a powerful shared bus interconnect with many advanced features. PLBv46 is described in detail in the Xilinx CoreConnect technology pages. All peripherals of a particular processor are connected to this primary PLBv46 system bus. Apart from the PLBv46 bus, each processor is capable of connecting to on-chip local memory (BRAMs) via local memory bus interfaces. In the PPC405 case, the local memory bus is called the On-chip Memory Bus (OCM), whereas in the MicroBlaze case, it is the Local Memory Bus (LMB).

Figure 2 shows a very basic, completely independent dual processor system architecture with each individual processing subsystem following the bussing paradigms just described.

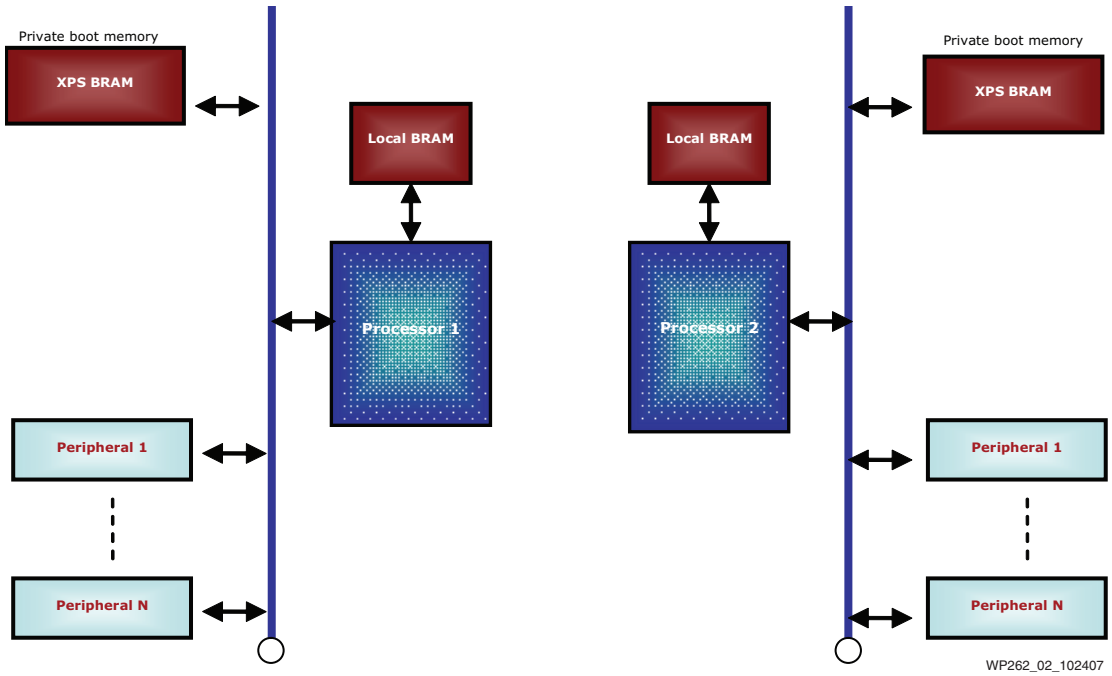

Figure 2: **Independent Dual Processor Architecture**

There are no hardware components linking the individual subsystems and hence there is no way for the processors to communicate at run-time. In the generic architecture above, a user can equivalently instantiate a PPC405 or a MicroBlaze in the place of each of the processors. Such completely independent processing systems are just a trivial enumeration of multiple single processor systems put together in the same

FPGA, but can still be useful to partition a large function set across multiple processors.

Figure 3 shows a more generic dual processor architecture where the two processors can communicate with each other and co-ordinate to accomplish a certain function.

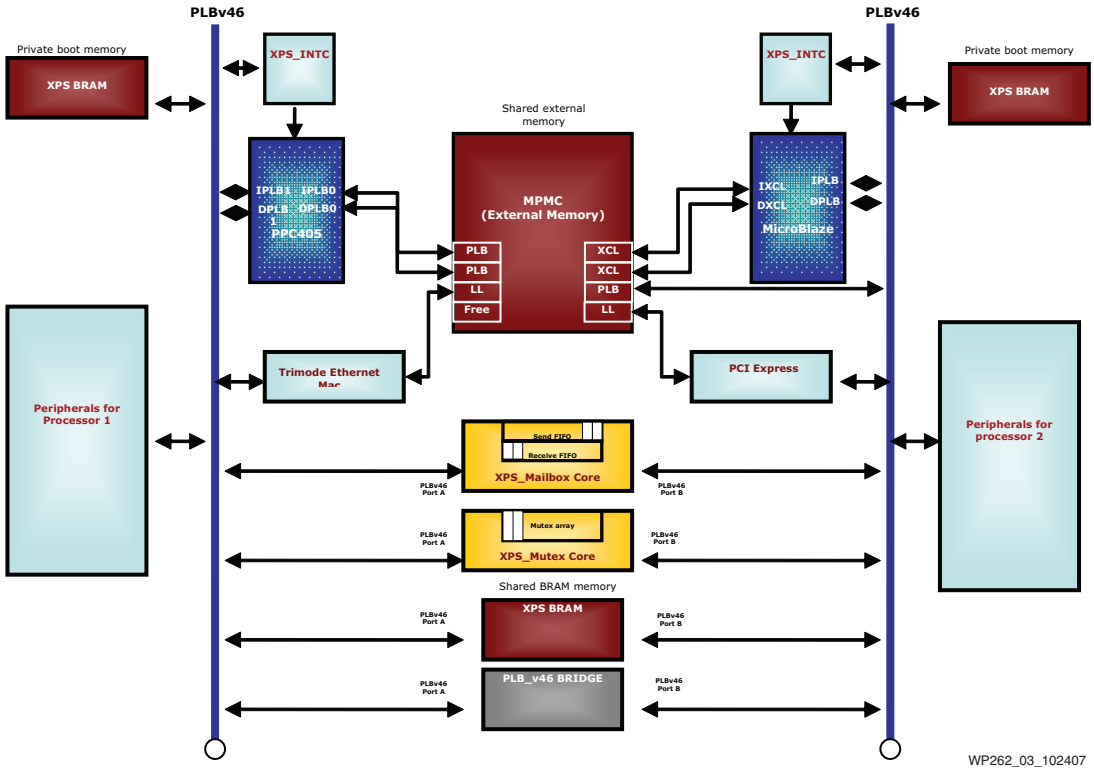

Figure 3: **Generic Dual Processor Architecture**

The key concepts in this architecture are as follows:

- The architecture is an extension of the completely independent processing architecture by connecting shared components between the two processors. The shared components allow the two processors to communicate with each other in various ways.
- The example intentionally shows a PPC405 as the first processor and a MicroBlaze as the second processor to illustrate certain specific characteristics of each processor. However, any one processor can be equivalently be replaced by the other with very minimal adaption, thus providing great flexibility in processor choice.
- Shared components are multi or dual ported in nature. The multi-ported nature of these components allows each PLBv46 system bus to be independent of the other both in terms of static as well as dynamic load. By isolating each processing subsystem, it is ensured that the system bus is not locked out for a processor or peripheral due to a currently executing transaction for another processor. All the multiported peripherals arbitrate access on the various ports internally.
- The key shared peripheral is the external memory controller - MPMC. MPMC is a unique memory controller supported by Platform Studio that offers many different interfaces (via ports) to the same external memory. Apart from PLBv46, MPMC port types include the high performance MicroBlaze Xilinx CacheLink (XCL) interfaces (Instruction XCL and Data XCL) as well as the point to point

cache link interfaces of PowerPC405 (IPLB0/DPLB0). This topology allows both the MicroBlaze and PowerPC processors to access external memory with minimal latency and high bandwidth at the same time. MPMC currently provides a maximum of eight ports, thus allowing three to four processors to connect to a single external memory.

- It is also possible to share internal BRAM memory between processors. Sharing on-chip BRAM can provide an extremely fast way to pass kilobyte sized data between the processors. In some cases, deterministic access to BRAM is an important required. This can be achieved by connecting the BRAM memory to the local memory interfaces of each processor (instead of the PLBv46 interface shown in Figure 2).
- Apart from shared memory, there are two other cores - the XPS Mailbox and XPS Mutex which provide other simple forms of communication. These cores are further explained in the Communication and Synchronization Setup section.
- Figure 3 also shows a PLBv46 to PLBv46 bridge connecting the second processor to the first processor's system bus. This might be required to share a peripheral that is not multi-ported in nature. For instance, some systems might wish to share a UART or SPI or I2C peripherals. Such a situation requires connecting the peripheral to a particular processor's system bus and providing the PLBv46 bridge from the other processors' system bus.

While, Figure 3 illustrates the recommended overall architecture for a system with multiple processors, there are however, a few other options available to architect the system. For instance, in a system in which logic area and resource usage is a key concern, all the processors could be located on the same system bus. While this makes the system less deterministic and increases run-time load on the bus, it offers area savings by eliminating a new system bus as well as removing the need for multiple ports on IPs. There are other derivative architectures possible, such as having the high performance processor on a separate system bus and multiple low performance processors on a shared system bus. Hierarchical topologies can also be created by connecting processing sub-systems to each other via multiple levels of bridges. An exhaustive listing of all such topologies is beyond the scope of this document. However, most of the concepts discussed in the rest of the document apply broadly across all the architectures.

Memory Map

Both the MicroBlaze and PowerPC processors use memory mapped I/O to interface with peripherals. The memory map of a processor is determined by the address ranges that are assigned to each peripheral and to the peripherals that are connected directly/indirectly to the processor.

As long as each processor uses a separate system bus, all memory and peripheral elements are cleanly isolated with the shared elements being clearly defined. Software on one processor will not see the peripheral subsystem of another processor. Within each processing subsystem, there are some requirements of the memory map of a processor to be able to run executables. The primary constraint is that there must be private memories mapped to the fixed reset and interrupt locations for each processor. Such private memories can be connected via either the local memory interfaces of the processor or the PLBv46 interface. Once such private memories are connected, Platform Studio address generation tools automatically take care of generating a clean memory map for each processor with the right address ranges assigned for each peripheral and memory.

Apart from the private memory sections, each processor can use any memory segment to run the rest of its software. Memory segments that are shared must be done so in a non- conflicting manner. The various sections of an executable and their sharing implications are discussed in greater detail in the Software Design section.

In derivative topologies with more than one processor on a shared system bus, all memory and peripheral elements on the bus are equally accessible by all processors sharing the bus. Although a peripheral might be accessed ever by one processor only, the peripheral is physically visible to the other processor. Though Platform Studio address generation tools automatically assign distinct address spaces to each peripheral, it is the responsibility of software to ensure non-conflicting access by each processor at run-time.

Interprocessor Communication and Synchronization

After multiple processors are connected to the interconnect infrastructure, the system designer begins designing the primary modes of communication and synchronization between the processors. On Xilinx processors, the most common communication schemes are Shared Memory and Mailbox based message passing. These communication schemes are described in the subsequent sub-sections.

Shared Memory

Shared memory is the most common and intuitive way of passing information between processing subsystems. A shared memory system has the following properties:

- Any processor can reference any shared memory location directly.
- Communication occurs via processor load and store instructions.
- Location of data in memory is transparent to the programmer. Data could be distributed across multiple processors, the details of which would then be abstracted away by some software API.
- Access to the shared memory segment must be synchronized by some hardware/software protocol between the two processors.

Shared memory is typically the fastest asynchronous mode of communication, especially when the information to be shared is large (> 1000 bytes). Shared memory also allows possible zero-copy or in-place message processing schemes. Shared memory can be built out of on-chip local memory or on external memory.

Sharing external memory is done using the MPMC memory controller as shown in Figure 3. Each port of the memory controller is mapped to the same address range and though each port, all the processors automatically share the entire external memory. It is the responsibility of the software designer to define the distinct shared and non-shared memory regions, based on some partitioning, and write the software protocol that uses the memory regions to pass information between the processing subsystems.

On-chip memory can be shared using PLBv46 based memory controllers as shown in Figure 3. Apart from PLBv46, local memory interfaces such as OCM and LMB can also be used to create high-performance, guaranteed latency, shared memory segments. This scheme works by connecting memory controllers via local memory interfaces to the on-chip BRAM blocks. Because on-chip memory BRAM blocks on Xilinx FPGAs are dual ported in nature, this scheme has the limitation that the memory can be shared between a maximum of two processors. The interfaces connected on either side can be LMB and/or OCM. This mode of sharing can be used between any pair of

processors - two PowerPC processors, two MicroBlaze processors, or a PowerPC processor and a MicroBlaze processor as shown in Figure 4.

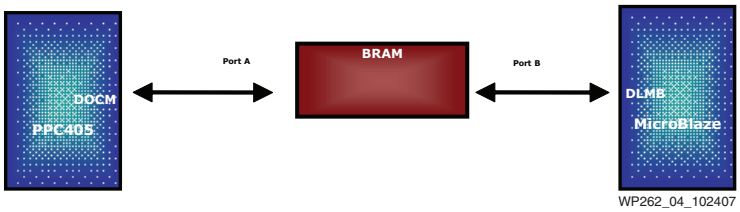

Figure 4: **Shared Dual Ported Local Memory**

When using shared memory in conjunction with caches on the processors, the user must be aware of coherency considerations. Neither the MicroBlaze processors nor the PowerPC processors provide cache coherency support and the software must enforce coherency. This is described further the Software Design section. Note that using local memory interfaces to on-chip BRAM has the same performance as a cache hit, therefore such regions of memory are typically not marked as cacheable. Thus by using local memory for sharing, a multiprocessor system garners all the benefits - guaranteed latency, performance, and memory coherency.

Mailboxes

Mailboxes are a method to pass messages between one or more senders and a receiver. The mailbox forms a channel through which messages are queued in a FIFO fashion from one end by senders, then dequed at the other by the receiver. The mailbox can be considered a simplified, TCP/IP-like message channel between the processors. The reception of the message at the end of the receiver may be done in a synchronous or asynchronous fashion. In the synchronous method, the receiver actively keeps polling the mailbox for new data. In the asynchronous method, the mailbox sends an interrupt to the receiver upon the presence of data in the mailbox. Typical usage of a mailbox is to pass pointers (to data) between the processors or to send actual data.

Xilinx provides the XPS Mailbox inter-processor communication core which provides the features described above. Each mailbox core has a pair of mailbox FIFOs, one for transmit and one for receive from a particular processor. The depth of the FIFOs is configurable by the user. The FIFOs are implemented either using distributed RAM or BRAM resources. Each mailbox has a pair of interfaces to connect to processors communicating via the mailbox. Though, it is possible to connect multiple processors to each interface, the recommended usage is to use a single mailbox between a pair for processors. Figure 5 shows the Xilinx XPS mailbox communication scheme between two processors.

The Xilinx XPS Mailbox hardware core is usually suited for small to medium sized messages, usually less than a few 100 bytes. The sender processor needs to copy the entire message from local or external memories and write it onto the FIFO. Hence it may not be suited for large messages. Similarly the receiver processor needs to copy the entire message back into its own memory. Involving the processor in this copying of messages wastes valuable processor cycles. Xilinx Mailboxes may be augmented in the future with DMA capabilities which will prevent the processor from having to do the message copies and thus be able to accommodate larger sized messages.

The arrival of a message on a mailbox FIFO is indicated to the receiver by sending an interrupt on the IRQ line coming out of the mailbox. The interrupt is deactivated when the mailbox has no more messages. The interrupts can be disabled at the interrupt

controller. Thus, the communication between sender and receiver can either be synchronous or asynchronous.

The arrival of a message on a mailbox FIFO is indicated to the receiver by sending an interrupt on the IRQ line coming out of the mailbox. The interrupt is deactivated when the mailbox has no more messages. The generation of interrupts is optional and can be turned off. Thus, the communication between sender and receiver can either be synchronous or asynchronous.

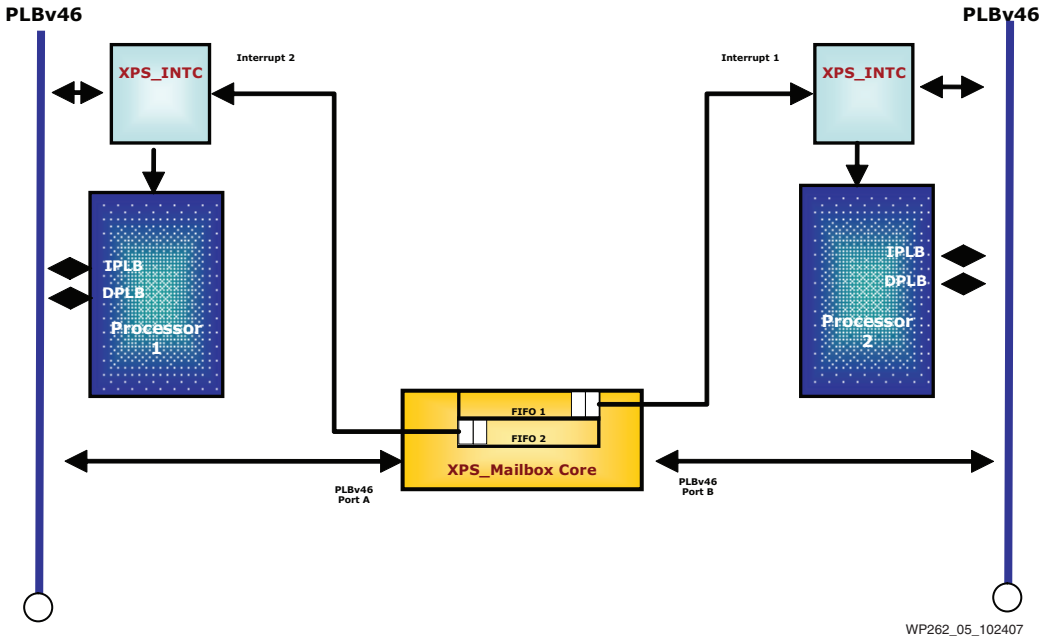

Figure 5: **XPS Mailbox for Message Passing**

Interprocessor interrupts (IPI) is a well known concept in desktop based processor architectures. It is defined as the ability of a processor to interrupt another processor in the system, thus delivering an event. The asynchronous messaging capability of the XPS Mailbox scheme may be used to generate interrupts between the processors. Sending an interrupt from one processor to the other requires only writing a single message on the mailbox. The mailbox core generates an interrupt for the receiver. In the interrupt handler, the receiver dequeues the message to simultaneously acknowledge the interrupt and dequeue the message.

The MicroBlaze and PowerPC processors also offer such FIFO style communication capabilities through the FSL interface. FSL is a unidirectional point to point FIFO link. The FSL capability of the PowerPC processor works through the Auxiliary Processing Unit (APU) interface connected to an FSL bridge core. Setting up a link from a master processor to a slave allows the master to send configurable width message words in a FIFO fashion. The MicroBlaze and PowerPC processors have special instructions that allow a program to write to a given FSL channel with optional non-blocking and/or control semantics. The point to point nature and the very minimal hardware requirements are the greatest advantages of FSL. In addition, the latency of sending a message through this interface is very low. When hardware or bandwidth constraints rule out the XPS mailbox scheme, the alternative FSL based scheme may be used for the same purpose. The XPS Mailbox also features an option to use FSL protocol at either of its two interfaces.

Synchronization

Processing nodes often need to synchronize with each other when accessing shared resources (peripherals, memory, etc). There are well known software semaphore and mutex lock techniques for providing synchronization between multiple threads on a uniprocessor system. These primitives are provided by an operating system running on the uniprocessor. The same software techniques cannot be used on an asymmetric multiprocessing system, because there is no common OS for all the processors.

Although there are some software protocols for achieving simple synchronization constructs, they may either be limited in features or require atomic read-modify-write support from the processor.

The MicroBlaze processor does not provide support for atomic read-modify-write instructions. Hence, Platform Studio provides a hardware synchronization module called the XPS Mutex to provide the ability to create mutual exclusion regions among multiple processors.

The XPS Mutex module provides a configurable number of memory mapped mutex registers which have a value component and a processor ID component. The mutex works on the test and set principle. Upon reset, the mutex value becomes zero, representing an unlocked mutex with an unassigned processor ID. To acquire the mutex, processors perform a write of the software-assigned ID of the processor to the corresponding mutex register and a value of zero. The mutex arbitrates simultaneous access to the mutex and stores the ID value written by the winning processor in the mutex value register. If the mutex is already locked, the mutex value remains unchanged. Each processor tests the successful acquisition of the mutex by reading back the mutex value and comparing it to its own processor ID. The processor that originally acquired the mutex is free to release it at any time by performing a write operation to the mutex register with its own processor ID and a value of one. Figure 6 shows how the XPS mutex must be hooked up to all the processors that use it.

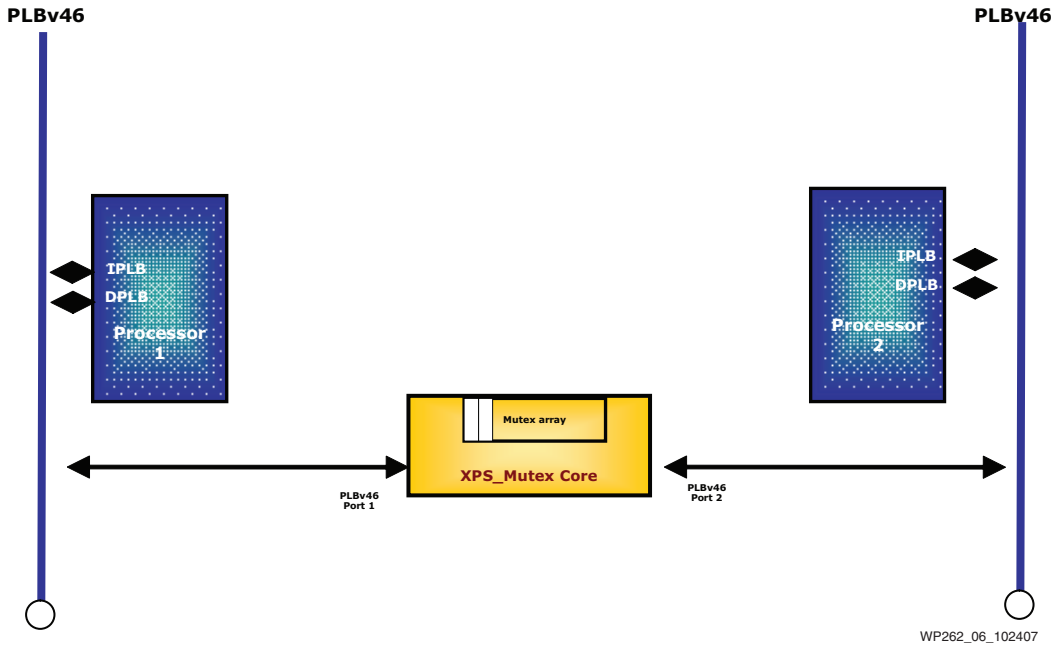

Figure 6: **XPS Mutex for Synchronization**

Software usage of the mutex module is described in further detail in the Software Design section.

Debugger Hookup

It is essential to be able to debug every processor in the design in an independent and system-aware manner. Both the MicroBlaze and PowerPC processors can be connected to Platform Studio debugger utilities via the JTAG interface.

The PowerPC processor is connected via an on-chip module called the JTAGPPC Controller. Each JTAGPPC Controller provides an interface with up to a maximum of two PowerPC processors. Debugger tools such as the Xilinx Microprocessor Debugger (XMD) can target either of the PowerPC processors in a JTAG chain and perform common debugging tasks such as stopping the processor, inserting instructions, and reading registers.

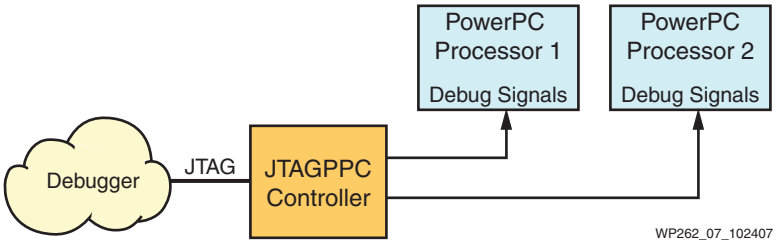

Figure 7: **Debug Interface to Multiple PowerPC Processors**

The MicroBlaze processors use a similar on-chip debug module called the Microprocessor Debug Module (MDM). The MDM module contains a configurable number of debug interfaces and has the ability to control a maximum of eight MicroBlaze processors at any one-time. It can select any one of the connected MicroBlaze processors at any particular instant and perform the common debugging tasks. Therefore, the debug interface can be used in a time-multiplexed manner across the different targets.

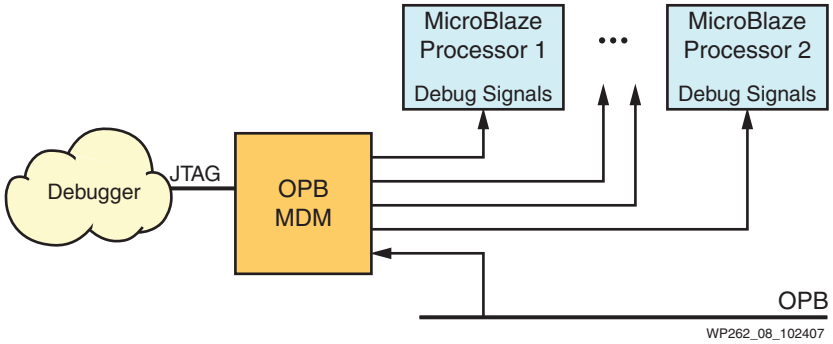

Figure 8: **Debug Interface to Multiple MicroBlaze Processors**

MDM also offers a JTAG based UART interface to the processors in the system. This UART is accessible via the PLBv46 bus interface.

In a system with both the PowerPC and MicroBlaze processors, a combination of JTAGPPC controller and MDM can be used.

The debug tools do not provide automatic support for debug target groups, i.e processor sets, on which debug tasks can be performed in a lock step manner. The user is required to perform individual operations on each processor manually. There is also no cross-triggering style feature - i.e., the ability to stop or start one processor when a

breakpoint is reached on the other. These features will be supported by Platform Studio in the upcoming releases

Clocking and Reset

Each processor defines a sub-system within the overall multiprocessing system. Each sub-system can potentially define its own clock and reset domains.

The MicroBlaze processor must run at the same frequency as the PLBv46 bus that it connects to. However, the PPC processor can be clocked at even multiples of the PLBv46 frequency. Other peripherals need to be synchronous with the bus frequency. Usually, this bus frequency becomes the system frequency or Fmax.

By using multi-ported IPs to communicate with each other, each processing sub-system can run the communication interfaces at their own frequencies, thus creating multiple clock domains. Similarly, FSLs can create asynchronous FIFOs between the two connected nodes. This method helps in making the most of a natural advantage of configurable processing systems; namely, that processor subsystems are typically heterogeneous in nature and due to various constraints it may not be possible to ensure a uniform Fmax without lowering it. Therefore, the ability to partition the clocking domains allows each node to be clocked at the best rate possible.

In multiprocessor systems, multiple reset domains can also be envisioned. Also from a system perspective, there are two different types of resets that can be desirable:

- Processor Reset

  Resets only a particular processor or a defined set of processors. The processor's peripherals are not reset. This may be desirable to selectively clear the state of a subset of processors.

- System Reset

  Resets the system including processors and peripherals.

The proc_sys_reset IP module is provided by Platform Studio that provides consistent sequencing of resets for the processor, bus and peripherals as well as providing independent processor reset signals. By connecting the various reset signals of the proc_sys_reset in a consistent manner, the two resets that are defined above can be achieved. While debugging, the debugger tools provide an option to perform the type of reset that is desired.

Runtime Processor Identification

A common requirement in multiprocessor systems is the ability of software to identify uniquely the processor on which it is executing at run-time. This feature is extremely useful in systems where code is portable across the various processors, and hence the ability to identify the executing processor can help customize the behavior of software.

The PowerPC processors on the Virtex-4 FX devices provide a PVR register that can be accessed by software. The PVR register is shown in Figure 7.

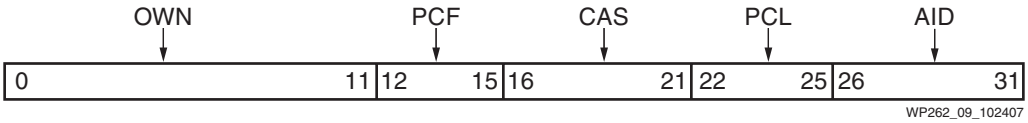

Figure 9: **Processor-Version Register (PVR)**

The least significant nibbles of the OWN and AID fields are available as configurable bits to the user.

The MicroBlaze v5.00a (and later) processors also offer configurable software-accessible PVR registers. The PVR features can be configured to be BASIC or FULL. In the BASIC version, an 8-bit USER1 field is available for configuration by the user. In the FULL version, there is an additional 32-bit USER2 field that can be used for custom purposes. Hence, the USER1 and USER2 fields can be used to identify processors in a multiprocessor system.

Software Design

This section describes software considerations in the multiprocessor design. The two most important considerations are:

1. How to assign the memory maps of the software programs that run on each processor
2. How software APIs are used to communicate between multiple processors.

Memory Map

Any memory that is shared in a multiprocessor design must be mapped and used in a non-conflicting manner by the software that executes on each processor.

Figure 8 for the MicroBlaze processors and Figure 9 for the PowerPC processors, show example memory maps for how two separate ELF files map into the local memories and the shared external memory that is available in the system.

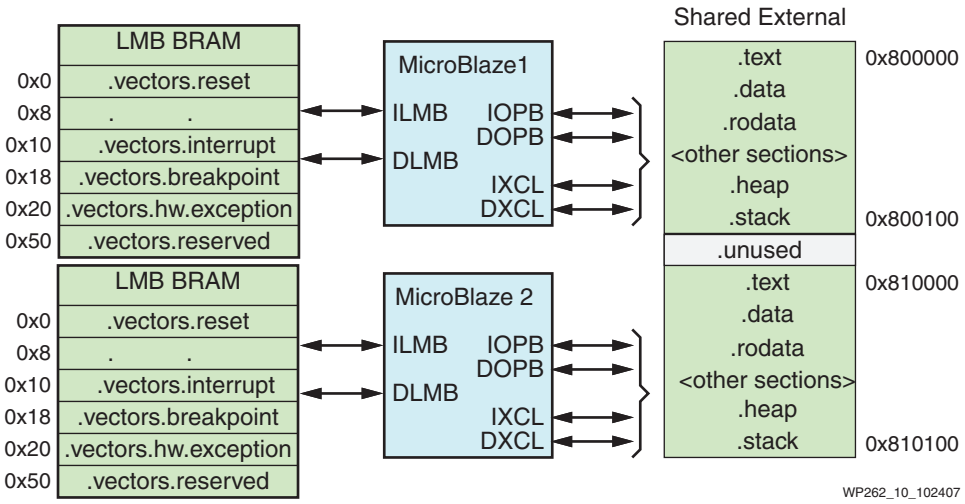

Figure 10: **Dual MicroBlaze Processor Memory Map**

In both the cases, the user has the flexibility to partition the external memory between the two executables as best suits the case. However, since the boot memory can typically not be shared, boot sections of the ELF files (.vectors.* on the MicroBlaze processor and .boot* on the PowerPC processor) are mapped to private local memories to ensure proper reset behavior. The memory map of each executable built by the compiler is controlled by special files that can be passed to the linker, known as Linker Scripts. Platform Studio abstracts away the details of linker scripts from the user. Assigning memories to the executable is greatly simplified by the Linker Script Generation utility. In a click-to-assign fashion, it allows the user to specify where in

memory the various sections of an executable file reside. The user can assign memories to each executable running in the multi-processor design and can also make sure that there are no conflicts.

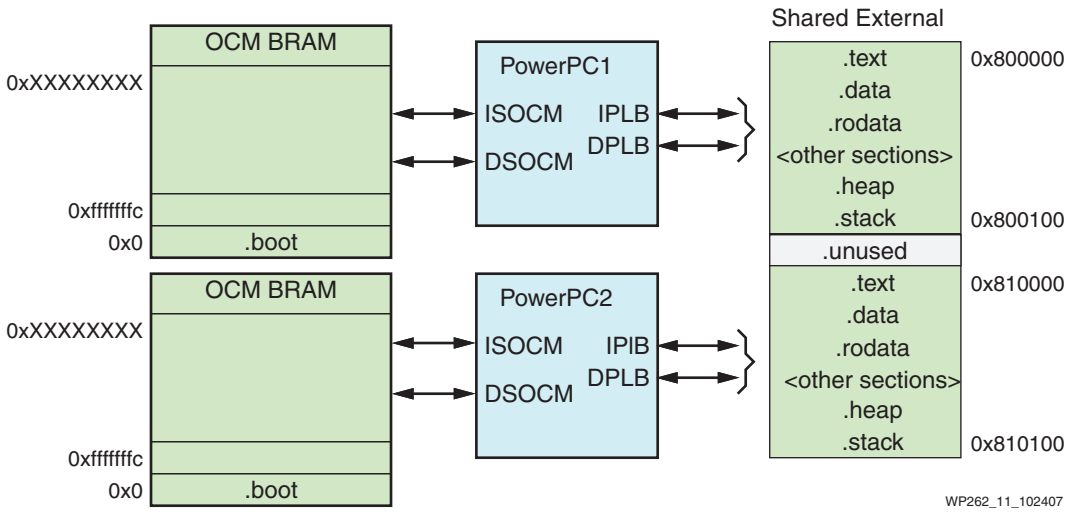

Figure 11: **Dual PowerPC Processor Memory Map**

It is possible to share code across multiple processors by using shared memory for holding the code. This scenario is useful when both processor are executing essentially the same software. Data sections (including heap and stack) still need to be kept private. Also, it is unlikely that the software will perform the same dynamic steps on both processors – some processor aware behavior is required. The PVR registers available on the MicroBlaze and PowerPC processors help software identify at run-time the processor instance that is executing the software at any given time. This technique can be used to share boot code as well, which may be useful when there is no private boot memory available.

Software applications interface with the devices in the system through Platform Studio device drivers. The device driver APIs abstract the details of the physical memory map from the user software. This is because these driver APIs access all peripherals with specific instance names rather than hard-coded addresses. The generated system software library maps these instance names to assigned addresses on the shared bus. In this way, user errors are minimized and software becomes very explicit on the regions of memory that it accesses.

Communication and Synchronization

Shared memory communication

Shared memory communication is the most common and obvious way of passing information between processors. Having a shared global variable or data structure in memory, software on a processor can easily update the value of the variable and have it be visible to other processors. All that is required is the address of the variable or a pointer into the shared region.

The region of code in which the shared data is modified is known as a Critical Region in OS terminology. Unless there is some sort of well-defined non-conflicting way in which each processor accesses the shared data, a synchronization protocol or construct is usually required to serialize accesses to the shared resource.

The XPS Mutex sychronization primitive described in the Hardware Design section, can be used for this purpose. As shown in the pseudo code in Figure 10, the semantics are similar to the well-known lock before use and unlock after use methodology used in uni-processor mutexes.

For more information on the software API for the mutex, refer to the mutex driver documentation in Platform Studio. The reference designs provided with *XAPP996 Dual Processor Reference Design Suite* illustrate software usage of the Mutex core

.

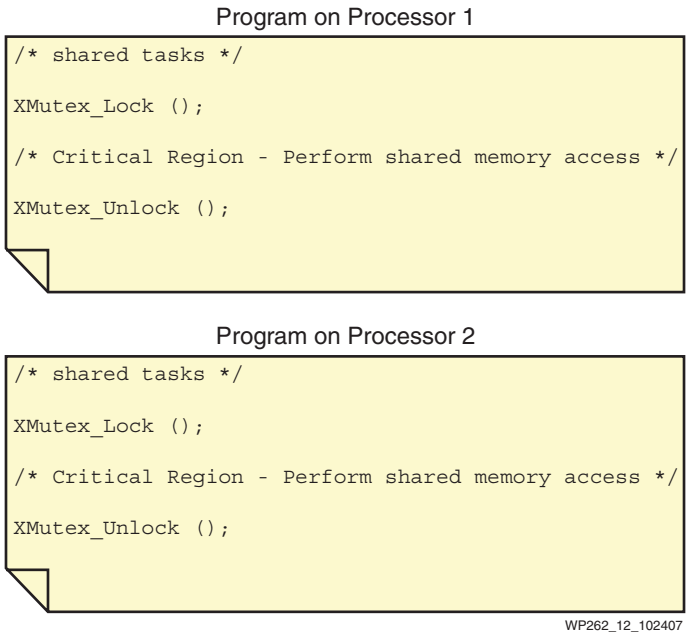

Figure 12: **Shared Memory Communication**

Another important consideration when using shared memory is cache coherency. If the shared memory region is cacheable by the processors in the system, the user has to consider those situations which could leave the cache in an incoherent state. Neither the MicroBlaze nor the PowerPC processor provide cache coherency in hardware. When both processors access the same physical memory, updates by one processor to the memory are not directly seen by the cache subsystem of the other. If required, it is up to the software to ensure coherency. A simple way of ensuring coherency is to invalidate those cachelines which correspond to the shared memory prior to access by the program. This causes the processor to refill the cache with any (potentially) modified data from main memory when accessing those cachelines. Another option is to dedicate non-cached regions of the main memory for shared memory purposes. Using shared on-chip memory via processor local interfaces is another approach to solving the coherency problem. This is because the local memory interfaces either do not enter the cache subsystems or offer a memory latency close or equal to a cache hit and hence such memories are typically not marked cacheable.

Each of these options has some advantages and disadvantages in flexibility, performance, and functionality. The scheme that works best for the application must be carefully selected.

Some types of shared data access do not require any synchronization nor memory coherency. For example, if the shared data model is similar to the simple form of *multiple readers*, *single writer* problem, the readers do not need to access the shared data in a mutually exclusive manner from each other. It is sufficient that the writer get to

access the data in an exclusive fashion. Such common synchronization, coherency, and consistency paradigms are described in popular publications on Operating Systems concepts. The shared memory should be modeled based on the paradigm the application finds as best fits.

Message Passing

The XPS Mailbox hardware primitive described in the Hardware Design section and its software drivers can help provide message passing features between processors. The software API is oriented in the fashion of the read() and write() calls, therefore the software can treat the mailbox as a serially accessed file for sending and receiving data. The software library provides blocking and non-blocking versions of the API. The asynchronous message passing feature allows the software on a processor to make progress without having to waste cycles in a spin-loop for data to arrive on a mailbox, thereby isolating slow senders from fast receivers that have other time critical tasks to perform. The asynchronous message passing feature can also be used as a form of inter-processor interrupts.

For more information on the software API for the mailbox, refer to the mbox driver documentation in Platform Studio. The reference designs provided with *XAPP996 Dual Processor Reference Design Suite* illustrate software usage of the Mailbox core.

Rendezvous and Barriers

One common requirement of multiprocessing systems is that they come out of reset and perform some sort of synchronization step with each other before proceeding onto individual dedicated functions. For example, in multiprocessing systems with master slave relationships, the master processor is in charge of initializing the operating environment for all slaves after which it *kicks* off the slaves. This sort of synchronization is typically formulated as *rendezvous* or *barrier* type problems. Software is free to define its own rendezvous protocol by using any of the inter-processor communication schemes described in the previous sections. For example, a rendezvous maybe implemented by using a single variable or a set of variables in shared memory. Alternatively, slave processors may be sent a *startup* message via an XPS mailbox by the master. A third possibility is to use the interrupt generation capability of the mailbox for interrupting processor(s), thereby signalling a barrier or rendezvous type of event.

Debugging

Platform Studio debugger and profiling tools can allow the user to debug processors in a completely independent fashion. Debuggers may perform reset actions on a processor before downloading code to debug. Unless resets are clearly isolated between processor sub-systems, debugging on a particular processor may cause unwanted side-effects on other processors. The debugger tools provide an option to debug each processor in an isolated and non-intrusive fashion.

Platform Studio debugger tools do not provide co-operative debug and cross-triggering capabilities between software on the different processors. Future releases of Platform Studio will support these features.

Operating Systems

Software running on each processor can include not only bare-metal libraries, but also operating systems. SMP is a common requirement for multiprocessor support by some of the popular operating systems such as Linux. However, there are several OS vendors who support Asymmetric multiprocessing with a heterogeneous mix of processor cores. As explained before, the heterogeneous model is more suited for embedded systems on FPGAs. Third party vendor operating systems layer on top of the Xilinx hardware and software IP infrastructure described above to provide many useful multiprocessing solutions in software. For a list of third party vendors and their support for multiprocessor systems, see http://www.xilinx.com/ise/embedded/epartners/listing.htm#RTOS

Conclusion

This white paper gives a high level overview of various typical scenarios in which multiprocessors are used. Generic concepts are described in detail as they relate to the designing of multiprocessor systems in Platform Studio. Various useful methodologies, capabilities, and common issues are discussed. The reference designs provided in *XAPP996 Dual Processor Reference Design Suite* are a useful starting point for building a custom multiprocessor design with Platform Studio.

Reference Documents

XAPP996 Dual Processor Reference Design Suite

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
4/23/07	1.0	Initial Xilinx release.
4/27/07	1.1	Added Ref Documents section; made other minor edits.
11/21/07	2.0	Rewritten for EDK 9.2.