



WP262 (v1.1) April 27, 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 multiple instances of 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.

---

© 2007 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

# 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 etc.

## Interface Processing

On a system which acts as a bridge or switch with 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

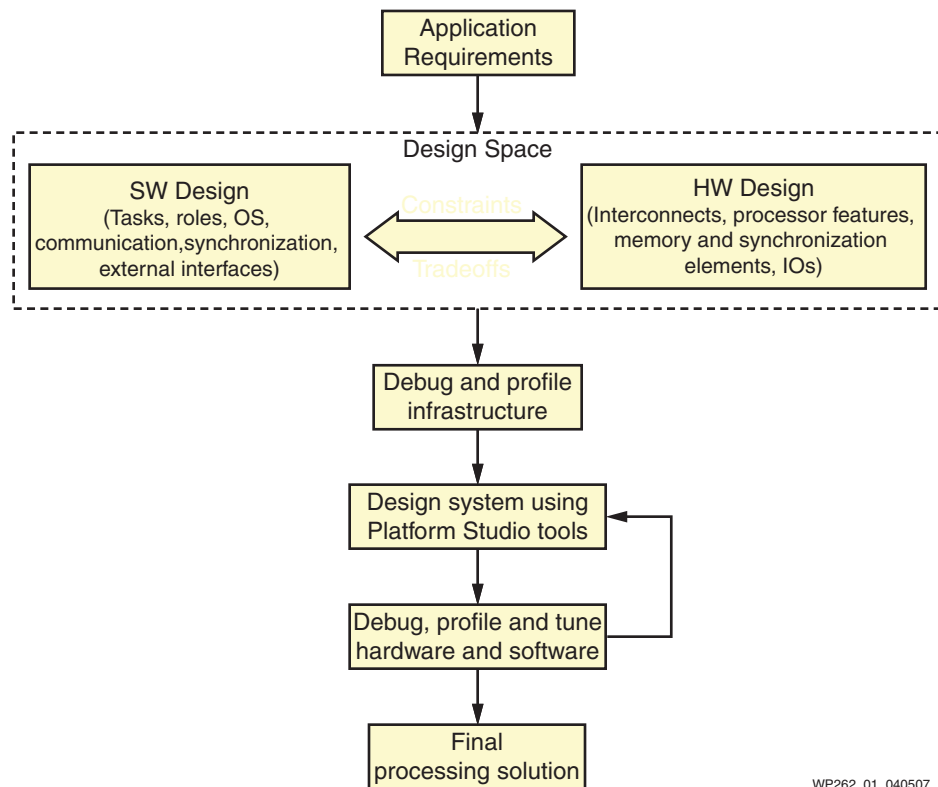
[http://www.xilinx.com/ise/optional\\_prod/tmrtool.htm](http://www.xilinx.com/ise/optional_prod/tmrtool.htm)

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 various requirements and features 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 or running the system at a frequency higher than 66 MHz might be hardware requirements. The flexibility of the system designer lies in exploring the huge design space offered by FPGA logic, coupled with configurable features on soft and hard microprocessors. 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.



WP262\_01\_040507

Figure 1: System Design Flow

## Background

The 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 data path 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 PowerPC 405 supports the IBM CoreConnect bus architecture and has high performance debug and trace features.

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 soft processor is supported on both the Xilinx Spartan and Virtex series of FPGAs.

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.

# Hardware Design

This section describes the hardware design considerations in multiprocessor systems.

## Overall Topology

The topology of a multiprocessor system is defined by the interconnect infrastructure used to connect the various processing elements, memories and peripherals. Figure 2 shows the use of a shared bus topology in a simple multiprocessor system in Platform Studio.

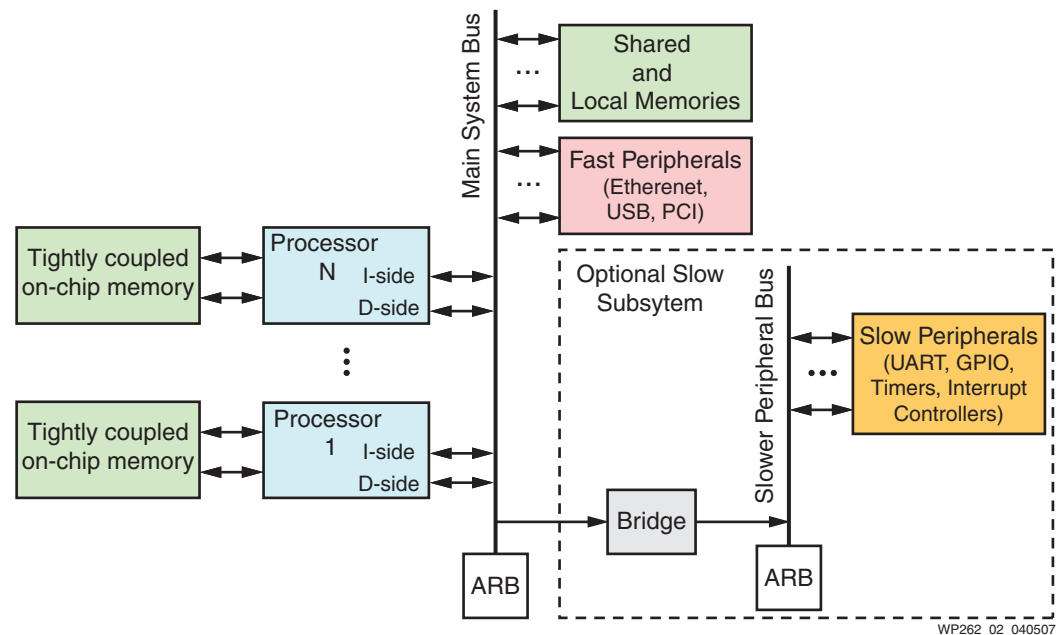


Figure 2: Overall Multiprocessing Topology

All the processors sit on a shared main system bus. Fast peripherals used by the individual processors and shared external or on-chip memories also sit on the main system bus. Other peripherals which do not require a high speed interface to the processor/memory, may sit on a second level low speed bus. The additional levels of buses are connected by bridges and help isolate sub-systems from traffic to and from other logical subsystems.

The primary busing paradigm used by Xilinx processors is the CoreConnect specification from IBM. The Processor Local Bus (PLB) and the On-chip Peripheral Bus (OPB) are the two main system buses defined by the CoreConnect specification. There are a few other interfaces defined by Xilinx to serve other commonly required needs. Collectively, the spectrum of bus interfaces falls under three broad categories:

- local memory interfaces
- high performance peripheral and memory access interfaces
- low speed peripheral access interfaces.

## Local Memory Interfaces

Both PowerPC and MicroBlaze processors can connect to embedded dual-ported BlockRAM memories (BRAM) with special local memory interfaces. These BRAMs are available on all Xilinx FPGAs and are extremely useful as high performance memories. They also serve an important function as private boot-code memories for

both PowerPC and MicroBlaze processors. This is explained in further detail in the [Memory Map](#) section.

Local Memory Bus (LMB) and On Chip Memory bus (OCM) are the primary interfaces to on-chip BRAM memory.

## High Speed Memory and Peripheral Interfaces

High Speed interfaces are used to connect high data rate masters and slaves, such as memories, processor caches and DMA masters. They contain some combination of features such as burst support, wide-width data buses, cache line transfers, and point to point channels.

Processor Local Bus (PLB), Fast Simplex Links (FSL), and Xilinx CacheLink (XCL) interfaces are the primary interfaces for fast peripherals and high speed memory. PLB is a powerful shared bus interconnect with many advanced features. PLB is described in detail in the Xilinx CoreConnect technology pages. FSL is a special point to point FIFO interface for connecting processors to hardware accelerators and or peer communication with other processors. XCL is the Xilinx cache interface. It provides a point to point link from cache subsystems to main memory with critical word ordering on returned data.

## Low-speed Memory and Peripheral interfaces

Low speed interfaces contain the most basic features used to connect low performance masters and slaves. On-chip Peripheral Bus (OPB) is typically used as a low speed shared peripheral bus. OPB is described in detail in the Xilinx CoreConnect technology pages.

These busing paradigms are used differently across the MicroBlaze and PowerPC processors. The following sections discuss PowerPC and MicroBlaze multiprocessor systems in more detail.

## PowerPC Topology

The overall topology of the dual PowerPC system is shown in [Figure 3](#). PowerPC processors use PLB as the main system interface and OCM interface to on-chip local memory. Slow peripherals such as timer, UART and interrupt controller are available with the OPB interface. PowerPC processor designs use a bridge from the main system PLB to the OPB to access these common peripherals.

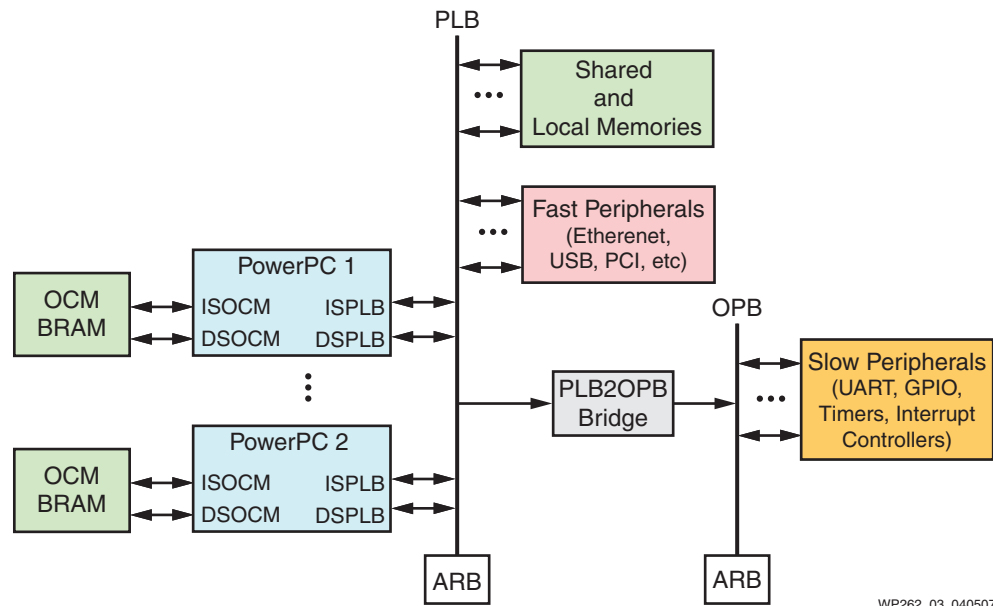
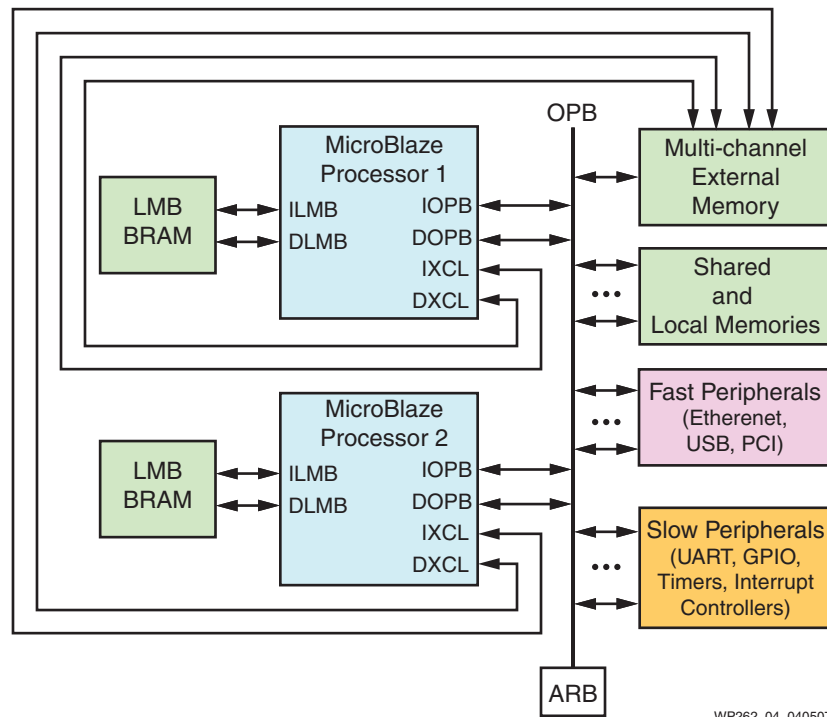


Figure 3: Dual PowerPC Processor System

This topology can be extended to more than two processors as well. Additional processors, if available on the target FPGA, can be added to the main system PLB.

## MicroBlaze Topology

The overall topology of a dual MicroBlaze system is shown in Figure 4. In MicroBlaze systems, the main system bus is OPB, while the local memory bus is LMB. The MicroBlaze processor also uses the XCL cache interfaces to multi-channel memory controllers for enhanced performance. Current memory controllers for the MicroBlaze processor provide up to 4 XCL ports, allowing cache interface to a maximum of 2 MicroBlaze processors in the system. Each MicroBlaze processor requires two ports, one for data and one for instructions. Additional processors can be added to the main system OPB.



WP262\_04\_040507

Figure 4: Dual MicroBlaze Processor Design

## Derivative Topologies

The shared bus topology does not scale for large number of processors. Similarly, due to the nature of the different bus interfaces supported by the MicroBlaze and PowerPC processors, there may be more than one system bus in designs that have both the processors and the presence of bridges in such a design bring down system performance.

It is possible to enhance the basic topologies discussed before by assigning a unique system bus per processor instead of attempting to share a single system bus. Bridge peripherals are available to connect the various system buses together to permit cross-traffic. By isolating the traffic to and from a processing sub-system, the dynamic load on the system bus is greatly reduced.

In such derived topologies, the external memory controller, which is a crucial part of an embedded system, is a key piece of the puzzle. It needs to provide access to each processor system and system bus in an optimal fashion. The standard EDK memory controllers support only one of the two system buses at a given time – PLB or OPB. They also have only one instance of a given interface. To get the most optimal system topology and performance, an architecture based on the Mult Port Memory Controller (MPMC2) is an excellent alternative. MPMC2 is a unique memory controller supported by Platform Studio that offers many different interfaces (via ports) to the same external memory. This topology allows both the MicroBlaze and PowerPC processors to access external memory via their high speed interfaces at the same time.



Figure 5 shows a MPMC2 based topology that contains both the MicroBlaze and PowerPC processors.

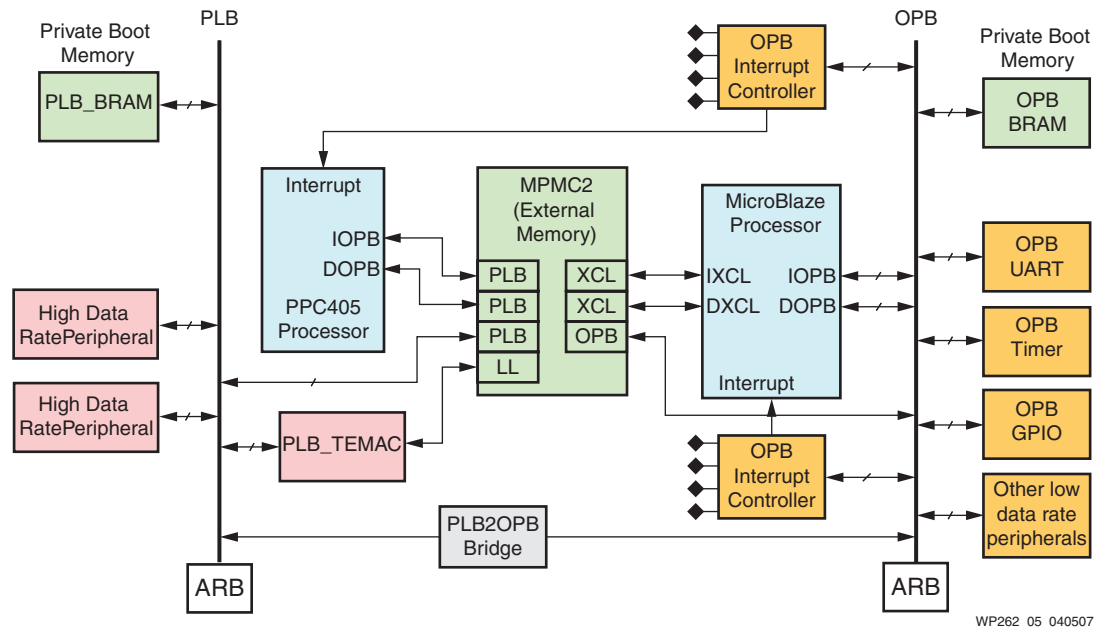


Figure 5: Example Dual PPC System with MPMC2

MPMC2 also allows multiple system bus topologies because of its ability to have multiple instances of a given interface. Thus it forms a central component in the derivative multiprocessor topologies.

## 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. Because the primary topology in multiprocessor systems involves a shared system bus, all memory and peripheral elements are equally accessible by all processors sharing the bus. It is the responsibility of the software to partition and access the available resources in a non-conflicting manner. Although a peripheral might be accessed by one processor only, the peripheral is physically visible to the other processor because it is connected to a shared bus. Therefore, the software on each processor must not access peripherals or memory segments that do not belong to the processor or are not shared with it. To distinguish between each peripheral on the bus, addresses must be assigned in a non-conflicting manner. Platform Studio address generation tools automatically assigns non-conflicting address spaces to all the slaves on the shared bus system. It also automatically creates address windows (ranges) in bridges which map from one shared bus to the other. Thus a great deal of the complexity in assigning addresses is automatically taken care of by Platform Studio.

The shared bus topology has certain implications on the memory map of the software executables that run on each processor. The MicroBlaze and PowerPC processors have fixed reset or bootup addresses: PowerPC -  $0xFFFFF0C$ , MicroBlaze -  $0x0$ . Similarly, the various exception vectors of MicroBlaze have a fixed location ( $0x8$  to  $0x5F$ ). When designing with multiple processors, it is not possible to place this boot-up memory on the shared system bus because there can be only one memory peripheral mapped to

the fixed address. Hence there would be a conflict for the vector code required by each processor. To ensure that each processor executes its own copy of the vector code, each processor must have its own private instruction and data memory mapped at these fixed address. The local memory interfaces of each processor (LMB for the MicroBlaze processor and OCM for the PowerPC processor) allow connecting private on-chip memory to each processor. Hence these memories attached to these local interfaces are ideal for holding the vector code.

Apart from the bootup 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.

## Communication and Synchronization Setup

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

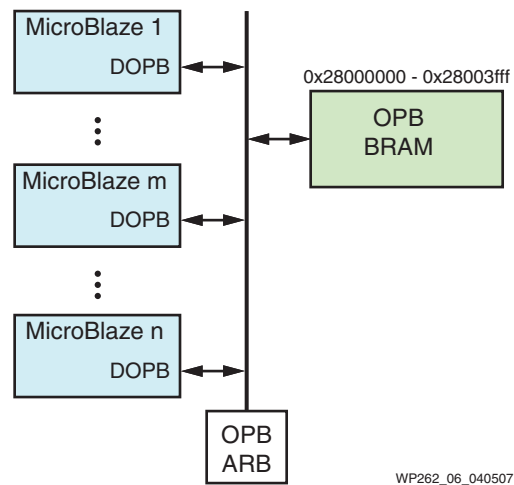
Shared memory is the most common and intuitive way of passing information between processing subsystems. A shared memory system has the following properties:

- 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 serialized 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.

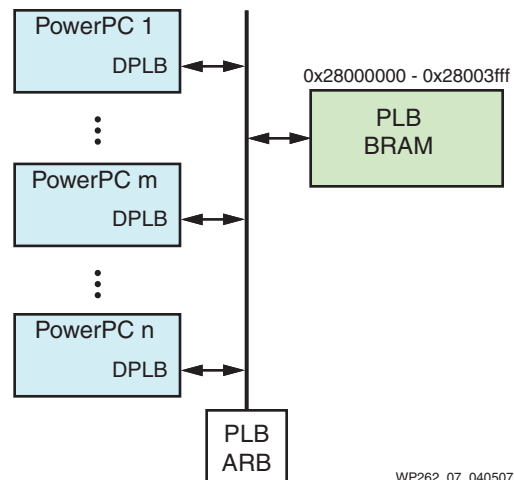
Shared memory can be built out of on-chip local memory or on external memory. Typical external memory configurations use a single external memory controller connected via the shared system bus. Because all of the processors are on the same system bus, they automatically *share* the external memory. It is the responsibility of the software designer to define the shared memory regions, based on some partitioning, and write the software protocol that uses the memory regions to pass information between the processing subsystems.

Local memory can be shared in a similar fashion using PLB and OPB BRAM controllers as shown in [Figure 6](#) and in [Figure 7](#).



WP262\_06\_040507

Figure 6: Shared OPB BRAM



WP262\_07\_040507

Figure 7: Shared PLB BRAM

The above scheme works well for any number of processors.

Apart from OPB and PLB, 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 master 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 8](#).

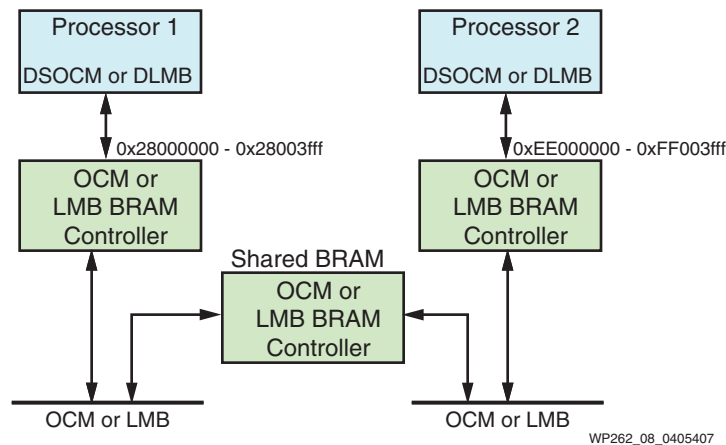


Figure 8: Two Processors Sharing On-Chip Dual-Ported BRAMs

Because the local memory interfaces on MicroBlaze and PowerPC processors are point-to-point interfaces to BRAM, the user can define a different memory map for the shared region on each of the two processors or use the same memory map.

When using shared memory in conjunction with caches on the processors, the user must be aware of coherency considerations. Neither the MicroBlaze nor PowerPC processors provide cache coherency support and the software must enforce coherency. This is described further in 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 dequeued at the other by the receiver. The FIFO is implemented using hardware logic or memory resources. 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.

Xilinx provides the OPB\_MAILBOX inter-processor communication core which provides the features described above. Each mailbox core has a single mailbox FIFO. The depth of the FIFOs is configurable by the user. The FIFOs are implemented either using distributed RAM or BRAM resources. Each mailbox provides a pair of mutex locks for allowing senders and receivers to transmit and receive messages in an exclusive fashion. [Figure 9](#) shows the Xilinx OPB mailbox communication scheme between two processors.

The Xilinx OPB 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 is not 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.

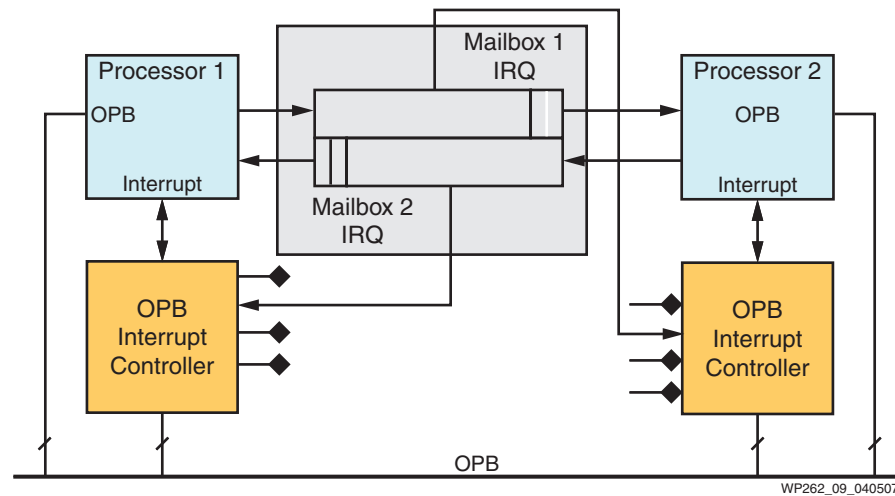


Figure 9: OPB Mailboxes

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.

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 OPB 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 OPB mailbox scheme, the alternative FSL based scheme may be used for the same purpose.

## 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 a 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 OPB\_MUTEX to provide the ability to create mutual exclusion regions among multiple processors.

The OPB 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 0x0', 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. 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 a reserved processor ID value. Figure 10 shows how the OPB mutex must be hooked up to all the processors that use it.

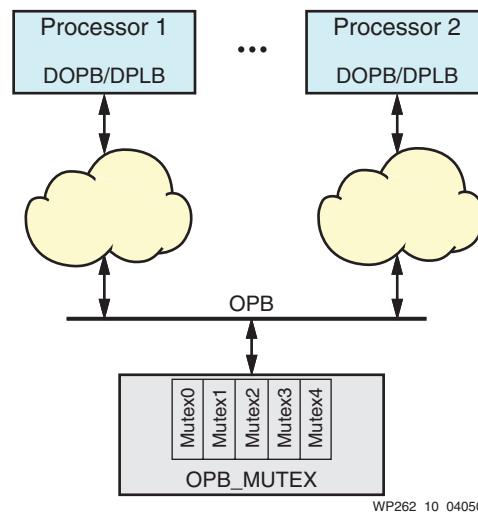


Figure 10: OPB Mutex for Synchronization

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.

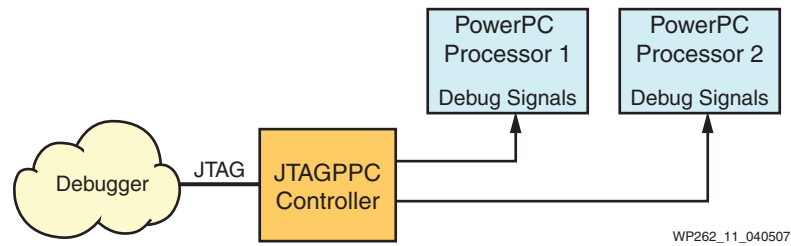


Figure 11: Debug Interface to Multiple PowerPC Processors

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.

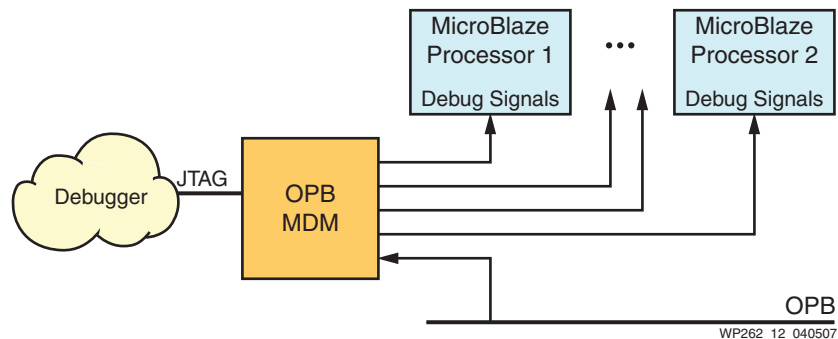


Figure 12: Debug Interface to Multiple MicroBlaze Processors

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

In a system with both the PowerPC and MicroBlaze processors, a combination of JTAGPPC controller and OPB 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 shared bus multiprocessor topology enforces the constraint that processors and peripherals on the shared bus be clocked at the same rate. The PowerPC processor is an exception. It can be clocked at even multiples of the PLB frequency. Other peripherals need to be synchronous with the bus frequency. Usually, this bus frequency becomes the *system frequency* or  $F_{\max}$ .



Processors that do not use a shared bus or shared memory topology, but instead use other communication methods to interface with each other, can have multiple clock domains. Also, 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 heterogenous in nature and due to various constraints it may not be possible to ensure a uniform  $F_{\max}$  without lowering it. Therefore, the ability to partition the clocking domains allows each node to be clocked at the best rate possible.

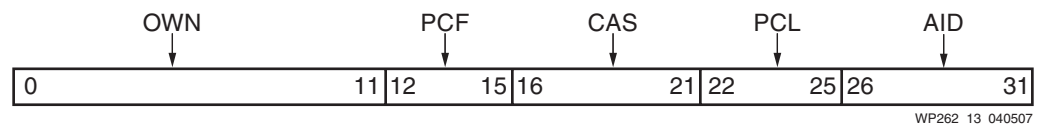
In multiprocessor systems, multiple reset domains can also be envisioned. Also from a system perspective, there are two different types of resets that can be desirable:

- **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.

By connecting the various reset signals of the system 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 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 13](#).



**Figure 13: 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 sits on the main shared system bus in a multiprocessor design must be mapped and used in a non-conflicting manner by the software that executes on each processor.

Figure 14 for the MicroBlaze processors and Figure 15 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.

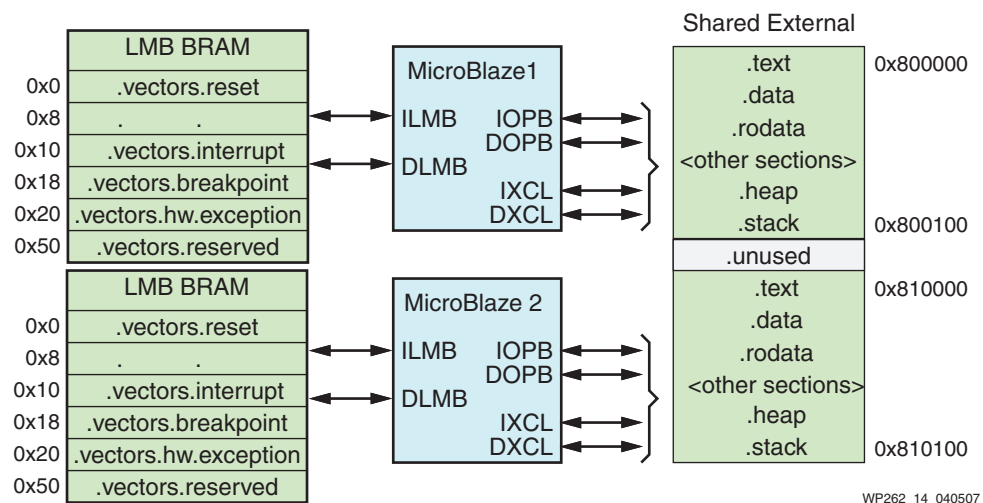


Figure 14: Dual MicroBlaze Processor Memory Map

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.

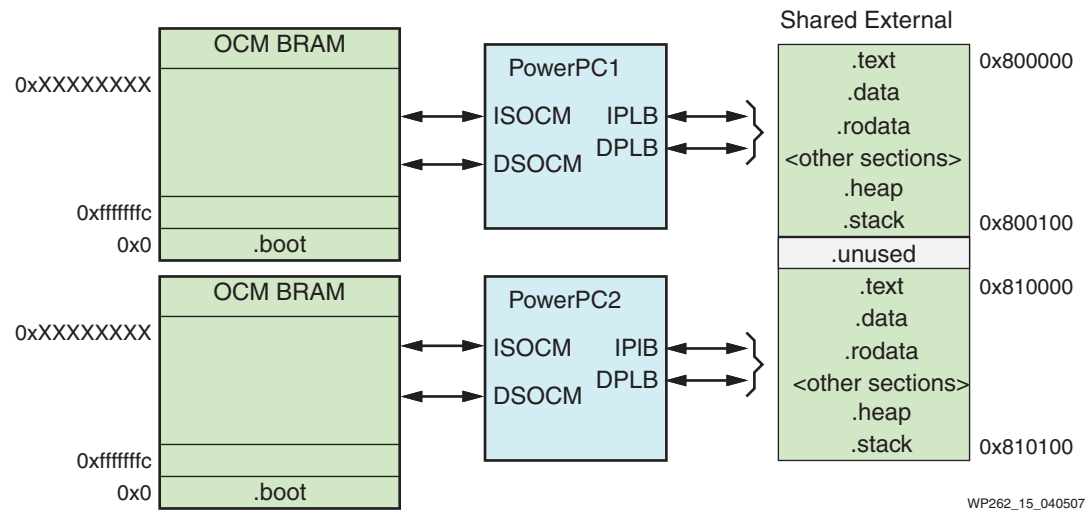


Figure 15: Dual PowerPC Processor Memory Map

It is possible to share code across multiple processors by using shared memory for holding the code. This scenario is useful when both processors 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 OPB Mutex synchronization primitive described in the Hardware Design section, can be used for this purpose. As shown in the pseudo code in Figure 16, 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 the reference designs supplied along with this whitepaper.

There also needs to be a link to this. Information will be available when the reference designs are available on the external web. Until then, this note is a reminder to make the link at the appropriate time.

Program on Processor 1

```
/* shared tasks */  
  
XMutex_Lock ();  
  
/* Critical Region - Perform shared memory access */  
  
XMutex_Unlock ();
```

Program on Processor 2

```
/* shared tasks */  
  
XMutex_Lock ();  
  
/* Critical Region - Perform shared memory access */  
  
XMutex_Unlock ();
```

WP262\_16\_040507

**Figure 16: 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 OPB 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 takes care of mutual exclusion, blocking or non-blocking accesses, and other behavior automatically. 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 *mbx* driver documentation in the reference designs supplied along with this whitepaper.

There also needs to be a link to this. Information will be available when the reference designs are available on the external web. Until then, this note is a reminder to make the link at the appropriate time.

## 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 the 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. 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>

## Common Pitfalls in Multiprocessor Design

### Starvation

The MicroBlaze processor uses OPB as the shared system bus. The platform generator tool in Platform Studio assigns static bus master priorities to the various MicroBlaze processors automatically. Because, by default, OPB provides only fixed priority arbitration, this could lead to the starvation of one or more processors on the shared bus, if the higher priority master is continuously requesting the bus. OPB provides an optional feature called *dynamic priority arbitration*, which should be enabled to prevent potential starvation situations and to allow fair access to all MicroBlaze processors.

### Deadlocks

In a system that uses both PLB and OPB, bridges are used to access peripherals across buses. When one of the processors is on PLB (PowerPC processor) and the other is on OPB (MicroBlaze processor), a deadlock situation can arise when the PowerPC processor on PLB is requesting a read on the OPB and the MicroBlaze processor on OPB is requesting a read on the PLB. There exists a possibility that the PLB to OPB bridge cannot access the OPB and the OPB to PLB bridge cannot access the PLB. This deadlock situation is caused because the OPB to PLB bridge suppresses the OPB time-out while it waits for data from the PLB. However, the PLB is busy waiting for data from the OPB. Because the PLB does not have a time-out mechanism during the data phase of a transaction, both buses are stalled and a deadlock can occur. To prevent this situation, the PLB to OPB bridge has an optional feature which allows it to wait a configurable number of clock cycles after asserting its OPB request for its grant to be asserted. If the grant is not asserted during this time period, the PLB to OPB bridge will assert a signal *BGI\_Trans\_Abort* to the OPB to PLB bridge. This signal indicates that the OPB to PLB bridge should issue a retry to the OPB master on the OPB which will cause an OPB arbitration cycle, thereby allowing the PLB to OPB bridge a opportunity to gain control of the OPB. The assertion of *BGI\_Trans\_Abort* also causes the OPB to PLB bridge to abort its current PLB transaction. By connecting this signal between the two bridges, a recovery from deadlock situations can be achieved.

## 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 accompanying reference designs provide 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.