WP162 (v1.1) April 10, 2003

Multiprocessor Systems

By: Jeremy Kowalczyk

With the availability of the Virtex-II Pro™ devices containing more than one Power PC processor and MicroBlaze™ and PicoBlaze™ soft processor cores, it is important to understand the basics of multiprocessor systems. This document provides a background for building true multiprocessor systems. It is by no means a comprehensive discussion on the topic of multiprocessing. For more information, see *Parallel Computer Architecture* by Culler and Singh, with Gupta.

Introduction

General-purpose microprocessors become cheaper and faster every year, making it easy to build powerful multiprocessor systems using these off-the-shelf processors as building blocks. These systems can solve problems many times faster (proportional to the number of processors) than a single processor alone. However, these processors must communicate in order to efficiently divide and solve a problem. In the past, there has been a heavy focus on either hardware or software to provide facilities for this communication. Both approaches limited the resulting systems in areas of speed and flexibility.

Lately, a focus on a higher-level layer, the *communication abstraction*, has given rise to more flexible systems that provide good speed-up and low cost. The communication abstraction provides a link between the software (programming model) and the hardware (physical implementation of the system). The communication abstraction needs to address the following issues of multiprocessor systems:

- *Naming*: How are shared data referenced?
- *Operations*: What operations are provided to the user on these data?
- *Ordering*: How are accesses to data ordered and coordinated?
- *Replication*: How are data replicated to reduce communication while still maintaining coherency (each processor must know where the most up-to-date data is)
- *Communication Cost*:
  - *Latency*: What is the latency of communication?
  - *Bandwidth*: How much data can be transferred per unit time?
  - *Synchronization*: How do data producers and consumers synchronize?

A good communications abstraction (Figure 1) addresses all of these issues and thereby defines what the hardware must provide to the software and what the software must provide to the user.

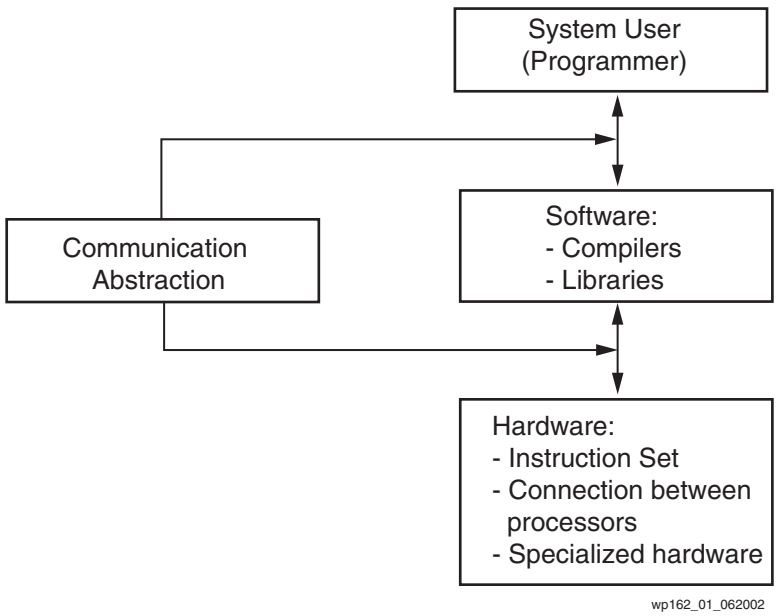

Figure 1: **Communication Abstraction**

These considerations have given rise to two popular types of general-purpose communication abstractions: shared memory (or shared address space) and message pass-

ing. Each of these types of communication abstraction deals with the issues above in a different manner, producing systems with different advantages and disadvantages. The remainder of this white paper describes each type of communication abstraction.

Shared Memory

In a shared memory system, all of the processors of the machine share a portion or all of the memory space. A system using this model has the following requirements:

- Provide for simple program synchronization. Functions must be provided to the programmer to allow him/her to initialize the environment, partition a job between the processing nodes, and complete execution.
- Provide for memory coherency. The machine must make sure that all of the processing nodes have an accurate picture of the most up-to-date memory (for example: if Processor A has a piece of dirty data in its cache and has not written it back to memory, all other processors must know this).
- Provide for atomic operations on data. The machine must allow for only one processor to change data at a time. All reads and writes must be atomic (the machine must not allow a processor to request a piece of data, and before the request is answered, allow another processor to change that data, so that the requesting processor doesn't get the data it originally requested)

A shared memory system has the following properties:

- Any processor can *directly* reference any memory location.
- Communication occurs implicitly as result of loads and stores.
- Location of data in memory is transparent to the programmer.
- A programming model is similar to multi-threading on uniprocessors (but threads run on different processors)
- Actual throughput near maximum theoretical throughput (speed up nearly proportional to the number of processors) on multi-programmed workloads in many cases
- Inherently provided on wide range of platforms (standard processors today have specific extra hardware for share memory systems)
- Wide range of scale: few to hundreds of processors
- Memory may be physically distributed among processors.

Software Programming Model

A shared memory system can be programmed using an extension of the multi-threaded uniprocessor approach. Rather than timesharing a single processor between threads, each thread can execute on a processor so that an increase in speed is obtained.

The typical structure of the virtual address space of a multi-threaded parallel program has a shared instruction segment, a shared segment for global variables, and private segments for each thread to store a stack and other private variables. Communication occurs when a *store* to a shared global variable by one processor is followed by a load by another. A shared global variable can also be used with an atomic operation for synchronization events between the processors. The programmer can access a *thread id* (proc_num in the code examples) to control execution flow through the common instructions.

Code Example

A pseudo-code example of a shared memory program is given below. In this implementation, a software library provides functions that synchronize execution. This is just one standard example, but illustrates the basic operations of a shared memory machine.

Problem: Sum all the elements of an array of size *n*.

Multiprocessor Software Functions Provided:

- INITIALIZE – assigns a number (proc_num) to each processor in the system; assigns the total number of processors (num_procs).
- LOCK(data) – allows a processor to "check out" a certain piece of shared data. While one processor has the data locked, no other processors can obtain the lock. The lock is blocking, so once a LOCK is encountered, execution of the program cannot proceed until the LOCK is obtained.
- UNLOCK(data) – releases a lock so that other processors can obtain it.
- BARRIER(n_procs) – When a BARRIER is encountered, a processor waits at that BARRIER until n_procs processors reach the BARRIER, then execution can proceed.

Example:

```
INITIALIZE; //assign proc_nums and num_procs

read_array(array_to_sum, size); //read the array and array size
//from file

if (proc_num == 0)  //initialize the sum
{
 LOCK(global_sum);
 global_sum = 0;
 UNLOCK(global_sum);
}
local_sum = 0;

size_to_sum = size/num_procs;
lower_ind = size_to_sum * proc_num;
upper_ind = size_to_sum * (proc_num + 1);

for (i = lower_ind; i < upper_ind; i++)
 local_sum += array_to_sum[i];
//if size =100, num_proc=4, processor 0 sums 0 to 24, proc 1 sums
//25 to 49, etc

LOCK(global_sum);  //locks the sum variable so only this process can
//change it
global_sum += local_sum;
UNLOCK(global_sum);  //gives the sum back so other procs can add to
//it

BARRIER(num_procs);  //waits for num_procs to get to this point in
//the program

if (proc_num == 0)
 printf("sum is %d", global_sum);

END;
```

The above program assigns a certain section of the array for each processor to sum. The processors add to a local_sum, then add their results to the global_sum. Each processor must lock the global variable so that two processors do not write to it at the same time. Finally, the result is ready when all of the processors have added to the global_sum and reached the barrier. The system designer needs to provide the necessary synchronization functions.

Hardware Implementation

A system that supports the above programming model could be made in many different ways. It needs to provide memory coherency (which is transparent to the programmer) and the ability for the LOCK and other synchronization functions to work.

Here is one possibility:

- Multiple processors on a shared bus (each has a cache)
- One memory
- Atomic memory instructions that only allow one processor to have a certain piece of shared data and write to it at a time (to support the LOCK operation). This code is being run on different processors, each with a specific unique value for proc_num.
- Hardware agent attached to the shared bus that maintains memory coherency (what happens if Processor 0 gets the global_sum and writes to it in its cache? If Processor 1 then needs to write to it, this agent must tell Processor 1 where to get the most up-to-date data. This is a coherency problem)
- Protocol for keeping coherency that the hardware described above enforces.

There are many protocols that maintain coherency (for example, MESI, SCSI, Dragon, and write-invalidate are some of the popular protocols). These all have different tradeoffs: some generate more traffic, and others allow for many more operations on cached data (and consequently faster program execution). It is important for the system designer to choose a protocol that fits the needs of the programs to be run. If there were a lot of data sharing, a protocol that generates as few transactions as possible would be a good choice to avoid congestion. If a program has infrequent shared data transfer, then a protocol which allows copies of valid data to be in multiple processor's caches would be advantageous.

Here is a diagram of the example hardware configuration:

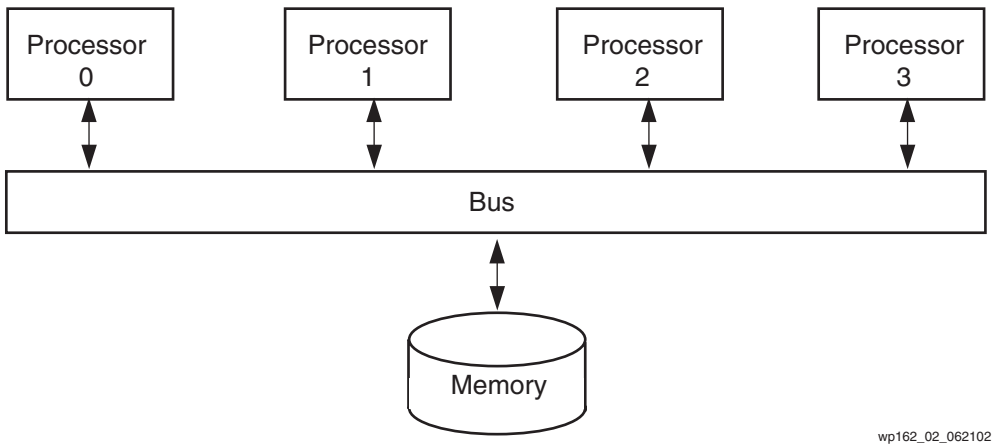

Figure 2: **Example Hardware Configuration**

The processors share some memory space, but not necessarily all. In Figure 3, all of the processors can access the Shared Memory, but only the owning processor can access its own data (Processor 2 cannot access Processor 1's private memory).

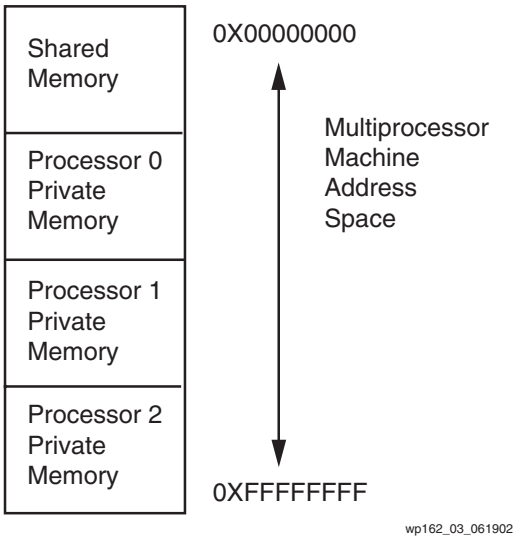

Figure 3: **All Processors Access Shared Memory**

Advantages

- Programming is very similar to multi-threaded programming on a uniprocessor.
- For programs which use shared data more extensively, the ability to have global data in a local cache can speed execution tremendously.
- Popular processors today have hardware extensions to support coherency between multiple caches and memory, so that systems with two to four processors can be built with low incremental cost.

Disadvantage

- Hardware cost – The hardware "agent" mentioned earlier adds complexity to the cache controller, which is expensive in most cases. This agent can become a bottleneck, unless designed well. Making this agent scalable is a major challenge.

Message Passing

A message passing machine handles the problems of communication between processors in a more explicit manner than shared memory. Instead of communicating through shared variables, the processors communicate by sending messages to each other. These messages and the programmer take care of all the coherency and synchronization problems inherent in a multiprocessor machine, but at a cost.

A message passing system has the following properties:

- Complete computer as building block, including I/O
- Programming model: directly access only private address space (local memory)
- Communication via explicit messages (send/receive)
- Communication integrated at I/O level, not memory system, so no special hardware
- Resembles a network of workstations (which can actually be used as multiprocessor systems). (The Xilinx Turns Engine is an example of a workstation cluster used as a multiprocessor machine.)

Software Programming Model

Each processor has SEND and RECEIVE functions available to the programmer. Each processor runs a program that passes messages back and forth to finish a job. The same types of synchronization primitives can be provided, but they are really just messages, so SEND/RECEIVE is all that is really needed.

In a message passing system, the memory space is segmented for each processor and there is no shared space. Any transfer of data requires a movement of that data from one memory space to another (Figure 4). So, if Processor 1 needs to have "Mydata," Processor 0 must send it to Processor 1, which places it in its own memory space.

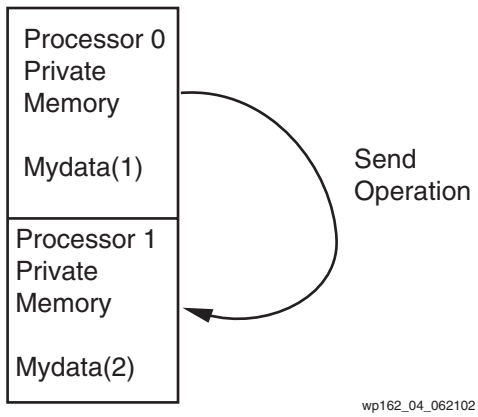

Figure 4: **Transfer of Data from One Memory Space to Another**

There appears to be a coherency problem in Figure 4, but since the programmer knows where the data was sent, he/she can make sure to update the data correctly. This makes programming a message passing machine much more difficult than traditional sequential programming.

Code Example

Here is the same array summing example, for a message passing machine:

Problem: Sum all of the elements of an array of size *n*.

Multiprocessor Software Functions Provided:

- INITIALIZE – assigns a number (proc_num) to each processor in the system, assigns the total number of processors (num_procs).
- SEND(receiving_processor_number, data) - sends data to another processor
- BARRIER(n_procs) – When a BARRIER is encountered, a processor waits at that BARRIER until *n_procs* processors reach the BARRIER, then execution can proceed.

Example:

```
INITIALIZE; //assign proc_num and num_procs

if (proc_num == 0) //processor with a proc_num of 0 is the master,
//which sends out messages and sums the result
{
 read_array(array_to_sum, size); //read the array and array size
 //from file
 size_to_sum = size/num_procs;

 for (current_proc = 1; current_proc < num_procs; current_proc++)
 {
  lower_ind = size_to_sum * current_proc;
  upper_ind = size_to_sum * (current_proc + 1);
  SEND(current_proc, size_to_sum);
  SEND(current_proc, array_to_sum[lower_ind:upper_ind]);
 }

 //master nodes sums its part of the array
 sum = 0;
 for (k = 0; k < size_to_sum; k++)
  sum += array_to_sum[k];

 global_sum = sum;
 for (current_proc = 1; current_proc < num_procs; current_proc++)
 {
  RECEIVE(current_proc, local_sum);
  global_sum += local_sum;
 }

 printf("sum is %d", global_sum);
}
else //any processor other than proc_num = 0 is a slave
{
 sum = 0;
 RECEIVE(0, size_to_sum);
 RECEIVE(0, array_to_sum[0 : size_to_sum]);
 for (k = 0; k < size_to_sum; k++)
  sum += array_to_sum[k];
 SEND(0, sum);
}

END;
```

The above program accomplishes the same task as the shared memory program. The processor on the system that is the master (processor 0) sends out parts of the array to the other processors to sum. The SEND and RECEIVE operations are blocking, so program execution cannot proceed until they are completed. Then the master waits for the processors to send the sum back, creates a global sum, and exits.

Hardware Implementation

Since the building block of a message passing machine is a complete system itself, this system could be built with a cluster of workstations on a LAN or off-the-shelf processors on a shared bus.

- No special hardware, proven, off-the-shelf processors

- Network topology (way the processors are connected) is more important due to larger, more frequent data transfer (could incur more hardware, like nodes that route traffic, or intelligent interfaces to other processors).

A message passing machine could use the same topology as the shared memory machine above. Here are some examples of different network topologies that could be used for either shared memory or message passing systems:

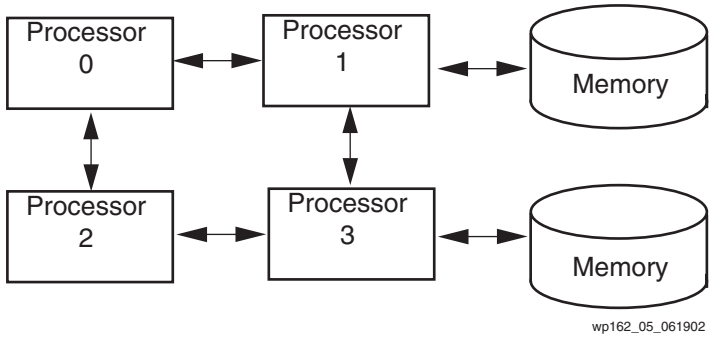

Figure 5: **Parallel Configuration**

This configuration allows transactions to be executed in parallel (as opposed to the shared bus configuration). Other topologies allow for different levels of parallelism with different costs (extra hardware, extra latency). For example:

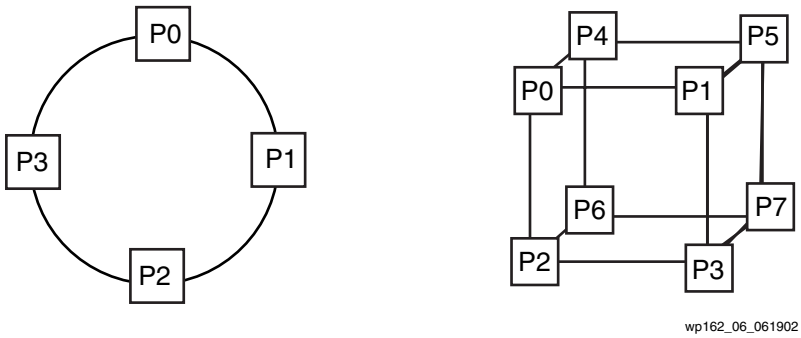

Figure 6: **Ring and Cube Network Topologies**

Advantages

- Easier to build than scalable shared memory machines
- Easy to scale (but topology is important)
- Programming model more removed from basic hardware operations
- Coherency and synchronization is the responsibility of the user, so the system designer need not worry about them.

Disadvantages

- Large overhead: copying of buffers requires large data transfers (this will *kill* the benefits of multiprocessing, if not kept to a minimum).
- Programming is more difficult.
- Blocking nature of SEND/RECEIVE can cause increased latency and deadlock issues.

Building Multiprocessor Systems with Xilinx Devices

Xilinx will be publishing additional white papers with more details regarding the design of a multiprocessor system on a Virtex-II Pro device using PowerPC processors or on any Virtex™ device using MicroBlaze or PicoBlaze processors. Application notes with reference designs will also be developed.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/27/02	1.0	Initial Xilinx release.
04/10/03	1.1	Replaced existing text in **Software Programming Model** and **Building Multiprocessor Systems with Xilinx Devices**. Made edits to **Code Example**, page 4, **Hardware Implementation**, **Software Programming Model**, and **Code Example**, page 7.