HW/SW Methodologies for Synchronization in FPGA Multiprocessors

Antonino Tumeo, Christian Pilato, Gianluca Palermo,
Fabrizio Ferrandi, Donatella Sciuto
Dipartimento di Elettronica e Infromazione, Politecnico di Milano
Milano, Italy
{tumeo, pilato, gpalermo, ferrandi, sciuto}@elet.polimi.it

ABSTRACT

Modern Field Programmable Gate Arrays (FPGA) can be programmed with multiple soft-core processors. These solutions can be used for MultiProcessor Systems-on-Chip (MPSoCs) prototyping or even for final implementation. Nevertheless, efficient synchronization is required to guarantee performance in multiprocessing environments with the simple cores that do not support atomic instructions and are normally used in the standard FPGA toolchains. In this paper, we introduce two hardware synchronization modules for Xilinx MicroBlaze systems, with local polling or queuing mechanisms for locks and barriers, and present a comparison of these solutions to alternative designs.[1]

Categories and Subject Descriptors: C.1.4 [Parallel Architectures]: Distributed architectures

General Terms: Design, Performance, Experimentation.

1. INTRODUCTION

Nowadays, Multi-Processor Systems-on-Chip (MPSoCs) are becoming the de-facto standard for designing embedded systems with high-performance requirements. On these systems, concurrent accesses to shared data or external devices are performed to allow the processors to cooperate on the same application. However, accesses to shared resources must be correctly sequenced and mutual exclusion during the execution of the critical sections of an application is required to allow a consistent operation of the system. Synchronization, for which different algorithms have been proposed since the inception of multiprocessor systems [1], is becoming a very important topic also for this class of systems.

Even the basic synchronization primitives, like *locks* (variables that indicate if a critical section of a program is currently executed or not) and *barriers* (points that all the processors must reach before continuing with the execution of the program), require some hardware support, typically atomic instructions, to be executed. A challenge for today embedded systems, where real-time responsiveness is required and the simple processing cores rarely supports advanced synchronization mechanisms, is to find solutions that limit the use of spin-locks with busy waiting without complicating too much the system design. Furthermore, *ad hoc* solutions are sometimes required to include, in the same system, multiple processing elements without atomic instructions support (e.g. ARM9-based architectures augmented with DSP units).

Recently, most FPGA vendors (e.g. Xilinx and Altera) have shipped system synthesis infrastructures, including different soft-cores (e.g., the NIOS from Altera [2], the CORTEX-M1 [3] from ARM, the MicroBlaze [4] from Xilinx), dedicated hardware components and interconnect elements. Thanks to the ever increasing size of reconfigurable logic devices, FPGA-based Systems-on-Chip (MPSoCs) are an becoming an appreciable solution for multiprocessor system prototyping [5] and, sometimes, even for final implementation of low volume, mission critical embedded systems (e.g., radar and military applications). They allow early evaluation of the challenges posed by new programming paradigms, new architectural approaches and can accelerate the development cycle of complex production systems. All these soft-cores, however, do not support atomic instructions, and thus other synchronization mechanisms are required to allow cooperation of the processors.

In this paper, we investigate the possibility to create a FPGA-based MPSoC with hardware supported synchronization where atomic primitives (e.g., *test-and-set*) are not available in the cores that compose the system. Our objective is to provide efficient lock and barrier primitives for such systems, where even standard *spin-locks* are difficult to be implemented. To achieve this goal, standard solutions like mutex hardware module and bus locking mechanisms are analyzed and evaluated. Then, two novel synchronization co-processors are proposed.

2. BASIC ARCHITECTURES

Thanks to the embedded design flows offered by the major FPGA producers, it is easy to design systems with multiple processors, connecting pre-designed elements together. Soft-core and hard-core processors can be connected to buses, internal and external memories, hardware accelerators and peripheral controllers to fast design embedded systems. However, when a multiprocessor architecture is required, the process is still not fully automated and some exploration have to be done in order to find a fully working solution and evaluate the possible trade-offs among different designs.

For this work, we used the Xilinx Embedded Developer Kit version 10.1 [6], which includes the MicroBlaze v7.10b [4]. MicroBlaze is a widely used and highly configurable Harvard architecture soft-core processor, with low/moderate occupation on FPGA. For example, it is possible to select pipeline implementation with three or five stages, depending on the required performance. Thus, with this core, multiprocessor systems can be implemented also on those small programmable devices on which a single, more complex, processing element (e.g., LEON 3) cannot even fit. The MicroBlaze sports several different types of connection. It can be connected to a local, private, memory, through the Local Memory Bus (LMB), which has very low latencies (a single clock cycle for reading, two

[1]Research partially funded by the European Community's Sixth Framework Programme, hArtes project.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'09, February 22–24, 2009, Monterey, California, USA.

Copyright 2009 ACM 978-1-60558-410-2/09/02 ...$5.00.

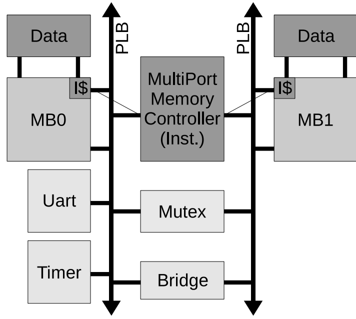


Figure 1: The multiple buses approach

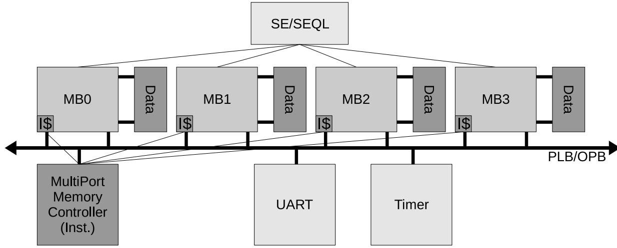


Figure 2: The shared bus approach

clock cycles for writing) but limited in size. It accesses external memory and peripherals by means of a IBM Core-Connect compliant bus connection, which in the latest version can be interfaced either with the On-Chip Peripheral Bus (OPB) or the faster Processor Local Bus (PLB). PLB is a very fast bus (typical data rate of over 500 MB/s at 100 MHz) used to connect high performance embedded processors, while OPB is a slower and simpler solution normally adopted for peripherals (typical data rate around 167 MB/s at 125 MHz). The MicroBlaze can be connected to coprocessors and accelerators through the Fast Simplex Link (FSL) ports, which allows to move data from the register file of the processors to the attached devices with simple means of single cycle blocking and non-blocking get and put instructions. Finally, it can manage its level-one caches, both for data and instructions, through the Xilinx Cache Links (XCL) ports, which use a simple point-to-point protocol to access purposely architected multiported memory controllers.

However, the MicroBlaze does not support any cache coherency mechanism and atomic instructions for synchronization. Thus, when implementing a multi-MicroBlaze system, there are several aspects that should be accounted for. When accessing shared data, data caches should not be used, or ways to enforce coherency are required. Furthermore, when multiple processors access the same location, solution alternative to atomic instruction to guarantee consistency should be adopted. For coherency, we adopt a model in which shared data are not cached but, eventually, moved from the shared, external, slower memory to the local memory for calculations, and then committed back to the external memory. For synchronization, instead, we analyze some standard mutex solutions and then propose two new designs.

Taking care of these aspects, several topologies for interconnecting MicroBlazes can be realized thanks to the EDK. We choose two solutions, one designed following the latest Xilinx guidelines [7] with a multiple buses topology to use the Xilinx Mutex Intellectual Property (IP) core [8] (see Figure 1) and one developed in house, with a shared bus topology [9] where we developed several synchronization alternatives (see Figure 2).

3. SYNCHRONIZATION MECHANISMS

This section details the synchronization mechanisms compared in this paper. The first one is bus locking, while the second one is the mutex mechanism introduced by Xilinx in its toolchain. The last two are our proposals, the Synchronization Engine (SE) and the Synchronization Engine with Queue Locking (SEQL).

3.1 Bus Locking

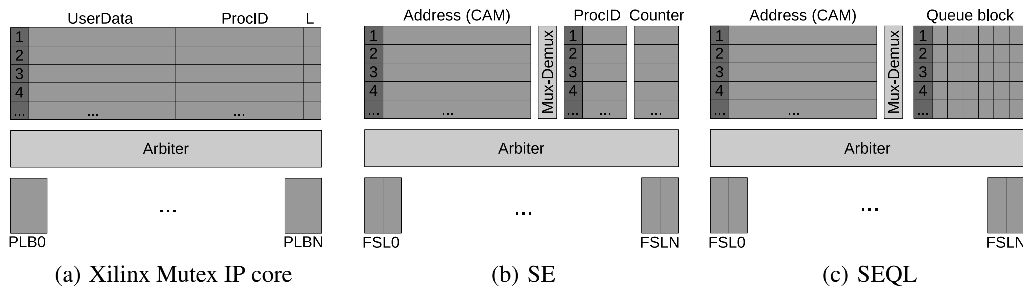
Bus Locking is a feature supported by some bus specifications that blocks re-arbitration and allows a master to retain the bus until the required operations are performed. Thus, with bus locking the processor can perform atomic operations. The Xilinx MicroBlaze v7.10b supports this mechanism through a register setting, but only when connected to the OPB. With a simple series of assembly instructions we defined two primitives that, when executed, respectively locks the bus (*bus_lock()*), changing the specific value in the status register (MSR) of the processor, and releases it resetting the register (*bus_unlock()*). These primitives can be used to execute inside them a critical section of code, in a blocking fashion (i.e. the lock becomes the bus locking, and the unlock is the unlocking of the bus), or to implement a spin lock mechanism.

Using only the bus lock/unlock actually eliminates the spinning, reducing the bus and memory contention (since the processor are blocked). However when the bus is locked, no other accesses to the shared memories are possible. If the program is running different threads at the same time, where some of them perform heavy memory operations and others are doing many synchronization operations, the access patterns to the bus can significantly limit the overall performance of the system. Using the pseudo-code of , the memory cycle of the processors are less influenced, but the spinning and the contention of the bus grow proportionally. We refer to the second solution for our benchmarks.

Implementing barriers with these mechanisms requires to use the lock and the unlock functions to write a counter in shared memory and then allow the processors to spin on it until it reaches the desired value.

3.2 Xilinx Mutex

The Xilinx Mutex IP core [8] is the solution for mutual exclusion included by Xilinx in the latest version of the EDK. The design is schematized in Figure 3(a). Basically, it is a simple memory, with a configurable number of slots (each of them represents a mutex), that can be accessed, alternatively, from multiple PLB ports. Each memory slot is composed by a bit that identifies if the lock has been taken or not, and up to 8 bits that save the identifier of the processor currently retaining the mutex. Optionally, each slot can use other 32 bits to store user defined data (e.g., memory addresses). When a lock is executed, the processor checks if the lock bit for the required mutex is set to 1 and, if not, it sets it to 1 and writes its id in the specific locations. If the lock is already taken, the primitive spins until the lock is released. When the unlock primitive is invoked, it sets the lock bit value to *0*. A trylock primitive, that checks one time the value of the lock bit and sets it to *1* is also available. Barriers are implemented using one of the mutexes to protect a counter in shared memory. This counter is used to register the arrival of the processors at the barrier, and when the requested number of processors has reached it, they are released. A fixed priority is used to determine the access ordering when contemporary requests are received. Note that only accesses from the different ports are mutually exclusive (i.e., read and writes to the memory from the same processor do not interleave), while accesses from the same port are not. Thus, it is not possible to use a shared bus with this IP core, and it is necessary to instantiate a PLB bus for each processor. This is not a problem while accessing the shared memory, since the multiport memory controller allows connection of multiple buses and even to bypass it, but it can be limiting when



(a) Xilinx Mutex IP core (b) SE (c) SEQL

Figure 3: Structure of the three synchronization modules

access to shared IP cores is required. In fact, these cores, which can be instanced only on a single bus, are accessed from the others through low performance bridges. Furthermore, more resources are required to instantiate multiple buses and, in the other hand, few processors can be implemented on small FPGAs.

3.3 Synchronization Engine

The Synchronization Engine (*SE*) is our *ad hoc* design of a coprocessor that manages locks and barriers and connects to each MicroBlaze through the FSL ports. FSL is a point-to-point protocol that allows for single clock cycle access to and from the register file of the processor, and is thus suited for the implementation of application accelerators and coprocessors, with lower latency with respect to bus, even high performance like the PLB. A schematic of the design is shown in Figure 3(b). The multiple FSL ports are managed by an arbiter which stores, in First-In First-Out (FIFO) ordering, the requests coming from the processors. When multiple requests arrive at the same time, fixed priority is used. The arbiter accesses a Context Addressable Memory (CAM) with a configurable number of 32-bit slots, which is used to store the identifiers of the lock (which normally are the addresses of the shared memory or of the shared memory mapped peripherals to lock). A second memory (Block RAM - BRAM, embedded in the FPGA) stores the identifiers of the processors currently retaining a lock. Finally, a third memory is used to store the counters to support barriers.

When a lock primitive is executed, it is initially checked if the identifier to lock is already in the CAM. If the check is negative, then this identifier is saved in the first free slot of the CAM, and the id of the processor that made the request is saved at the corresponding address of the first BRAM. If the check is positive, then the lock is taken by another processor, and thus the requesting processor starts spinning until the lock is released. When the unlock primitive is executed, the CAM is checked against the requested identifier to unlock and when there is a match, the CAM slot at the returned address is resetted along with the corresponding BRAM slot. Barriers are managed thanks to the counter BRAM. When a barrier request is made, the CAM is checked again the identifier for the barrier. If the identifier is not found, then a new barrier is initialized, saving its identifier in the first free slot of the CAM and the id of the requesting processor in the first BRAM. The counter BRAM is then initialized with the number of the processors requested to reach the barrier before release, minus one. When another processor executes a barrier request on the same identifier, the CAM returns the matching address in the counter BRAM, and the counter is consequently decremented. When the last processor reaches the barrier, the counter reach 0 and the slots in the CAM and in the processor id BRAM are resetted. When a processor is waiting while other processors reach the barrier, it locally spins checking if the barrier id is still present in the CAM.

We adopt a CAM, instead of a standard linearly addressable memory, because we designed the module to act similarly to a lock/barrier cache. When all the slots in our module are full, then the system rolls back to the use of the bus locking mechanism or, in case it is not available (for example with the PLB), it uses one of the slots to protect lock and barrier arrays in shared memory. At this point, we do not focus on replacement mechanisms to decide which locks should be stored in the module and which may be stored in memory, since we only want to verify what are the performance when hardware assisted solutions are used. We defer the study on the sizing of such modules at later works.

3.4 SEQL

The Synchronization Engine with Queue Locking (*SEQL*), shown in Figure 3(c) is a modified design of the SE, again developed in order to act as a barrier and lock centralized coprocessor. In this module we substituted the BRAMs for the processor *id*s and the barrier counters with a block of FIFO queues. For each slot in the CAM, a queue is instantiated. The number of slots and queues is, again, configurable. When a lock primitive is invoked by a processor, the CAM is checked against the lock identifier (shared address to lock). If the match is negative, then a free slot is initialized with the lock identifier and the processor gets the lock. If the match is positive, then another processor is currently retaining the lock for the requested resource. Thus, the requesting processor id is added to queue at the address returned by the CAM, and it is put in a blocking wait state. When the processor that currently has the lock executes the unlock primitive, the first element in queue (i.e. the first processor that was put in queue for waiting the lock) is popped and the lock is directly given to it. If no processors are queued in the corresponding queue, the lock id is simply removed from the matching CAM slot. When the barrier primitive is executed, the behavior is slightly different. If the id of the barrier is not present in the CAM, then a free slot is initialized and the requesting processor puts itself in queue and waits for release. When other processors will reach the barrier, the CAM will return the corresponding queue in which previous processors are waiting. If the number of queued processor is equal to the number of requested processor, minus one (the current processor), then the all the required processors reached the barrier and the release can start, popping the elements and restarting the processors from the blocking wait state, until the queue is empty. Again, we use a CAM, instead of a linearly addressable memory, since our objective is to use this module similarly to a lock cache. If all the instantiated slots are full, the system falls back to bus locking or, if not available, uses a slot of the memory to manage the array of locks and barriers and memory.

With this blocking approach, the queues should have has many slots as the number of processors. A non-blocking approach can also be devised, with a queue depth determined by the maximum number of cooperating threads desired by the programmer rather than by the number of processors. In this case, the lock primitive does not block the processor, but rather return the status of the lock, and the queues save a thread id instead of the processor id. When a thread asks for a lock, if the lock is not taken, the execution of the thread continues unaltered. If it is taken, then the thread id

	Bus Locking	Xilinx	SE	SEQL
Features				
Connection	OPB	PLB	FSL	FSL
Shared bus	Yes	No	Yes	Yes
Multiple buses	No	Yes	Yes	Yes
Spin-locks type	No Spin	On module	On module	No Spin
HW barriers	No	No	Yes (local spin)	Yes (no spin)
Resource utilization				
Slices	0	560	1492	1886
Slice FFs	0	730	1674	2067
LUTs	0	460	2781	3274
BRAMs	0	0	4	4
Performance [cycle]				
Lock	33	83	46	44
Unlock	32	84	43	43
Barrier	242	466	78	50

Table 1: Features, area and performance comaprison of the different synchronization mechanisms

is saved in the pertaining queue, and the processor can switch to another ready thread and do other useful work. When the thread that currently has the lock releases it, a processor is interrupted and the waiting thread can be reset to the ready state, and eventually resume its execution.

4. EVALUATION

In this section we evaluate the different synchronization mechanisms analyzed in this paper. The mechanisms present several different features. In the upper part of Table 1, we report the most significant aspects before moving to the performance analysis. We tested our hardware synchronization modules (i.e., SE and SEQL) with both the bus types, the PLB and the slower OPB, since they use the FSL connection and thus only depend from the presence of the coprocessor ports on the MicroBlazes. The *Xilinx Mutex IP core* requires the PLB, with a multiple buses topology. *Bus Locking*, instead, can be applied only using a single shared OPB bus. The Xilinx Mutex IP core does not support hardware barriers and spinning is performed on memory locations of the module. The Bus Locking mechanism does not use spinning, since the processor retaining the lock is the only one that accesses the bus, while the others are stopped. However, hardware barriers are not supported. *SE* uses spinning locks, but the spinning is on memory locations of the module itself, and supports hardware-assisted barriers, albeit with a local spinning mechanism. Finally, *SEQL* supports locks and barriers without the necessity for the requesting processors to perform busy waiting (they are stopped in blocking wait).

Table 1 in the central part reports the resource utilization of the different hardware synchronization modules with 8 ports and 16 slots for mutexes implemented. Bus Locking obviously does not require any dedicated logic on the device. Xilinx IP Cores has the lower utilization, however, for its implementation, a dedicated PLB bus instance is required, and thus occupation is not limited to the module itself. Actually, for this reason, we cannot design systems with more than three MicroBlazes with this module, while, with our solution, we were able to scale up to six processors. Furthermore, both SE and SEQL use a CAM, which requires BRAMs for implementation, while memory on the Xilinx module is limited to the register of the mutexes. The differences in logic utilization between SE and SEQL is determined by the use of the queues, implemented through shift registers, instead of simple memories. On our target platform, the maximum synthesis frequency for both the SE and the SEQL is around 157 MHz with Xilinx ISE 10.1 on a Virtex-II Pro XC2VP30.

Table 1 in the lower part presents the average latency in clock cycles for the execution of the *lock*, *unlock* and *barrier* primitives on the different synchronization mechanisms, measured through the system timer. It is easy to see that *Bus Locking* is faster for the execution of the lock and unlock primitives. This is due to the fact that the processor is simply executing three assembler instructions in order to set the status register of the processor to enable and disable bus locking. The performances of the SE and the SEQL, when doing lock and unlock, is very similar to bus locking. Access to these modules is fast and the execution latency just comprises the execution of some get and put instructions and the delay due to the hardware operations. Notice that lock on SE is two clock cycles slower than SEQL. This is due to the following reason: the delays only refer to the time of lock acquisition and no spinning is required. However, the lock primitives with SE needs to perform the test before setting the lock and it takes approximately two clock cycles. With lock and unlock, the Xilinx Mutex IP core is the slowest. This is due to the fact that, to access the core, the bus must be traversed, and even if it is has less load since the system is used a multiple buses topology, arbitration times must be taken in consideration. With barriers the situation gets worse, since not only the Mutex core must be accessed to get the lock on the counter for barrier arrival in shared memory, but the processors also have to spin on it. With the Bus Locking mechanism, the performance is again limited by the necessity to spin on the counter in shared memory, and thus the effects of contention appear. With SE the situation gets better: the processors are spinning only on the counters in the memory of the fast synchronization module. Finally, the SEQL presents the best result since the processors are not spinning at all: they are queued and put in blocking wait until all the processors arrives.

5. CONCLUSIONS

In this paper we analyzed the possibility to implement efficient synchronization on MPSoCs prototypes designed on FPGA with processors that do not support atomic instructions. We used the Xilinx MicroBlaze soft-cores and the the Xilinx toolchain as the prototyping platform for this study. We presented two coprocessors which allow hardware support for locks and barriers. The former, the Synchronization Engine, limits the spinning for locks and barriers to local memories in the module. The latter, the Synchronization Engine with Queue Locking support, completely eliminates the spinning for both the locks and the barriers through the use of queues with blocking waits for the request. We compared, on working prototypes, these solutions to two other standard ones applicable to the same problem.

6. REFERENCES

[1] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[2] Altera Corporation, 101 Innovation Drive, San Jose, California 95134, USA. *Nios II Core Implementation Details*, v8.0.0 edition, 2008. http://www.altera.com.

[3] ARM Cortex-M1 processor. Available at http://www.arm.com/products/CPUs/ARM_Cortex-M1.html.

[4] MicroBlaze Processor Reference Guide. Xilinx Corporation.

[5] Research Accelerator for Multiple Processors (RAMP), http://ramp.eecs.berkeley.edu/.

[6] Xilinx embedded developer kit (EDK). Xilinx Corporation.

[7] Designing Multiprocessor Systems in Platform Studio, WP262 (v2.0) November 21, 2007, Xilinx Corporation.

[8] XPS Mutex (v1.00a). DS631 february, 20, 2008, Xilinx Corporation.

[9] A. Tumeo et al. A Design Kit for a Fully Working Shared Memory Multiprocessor on FPGA. In *Proc. of (GLVLSI)*, pages 219–222, 2007.