Support for multiprocessor synchronization and resource sharing in System-On-Programmable Chips with SoftCores

Paolo Gai*, Giuseppe Lipari, Marco Di Natale, Matteo Duranti, Alberto Ferrari+
*Evidence Srl, Pisa  ReTiS Lab Scuola S. Anna, Pisa  +Parades, Roma (Italy)
pj@evidence.eu.com, {lipari,marco}@sssup.it, duranti@gandalf.ssup.it, aferrari@parades.rm.cnr.it

Abstract

In this paper we present an arbitrator implementation for enabling multiprocessor synchronization and resource sharing primitives with guaranteed maximum latency in real-time operating systems (RTOS). The arbitrator has been developed for use in conjunction with the NIOS II softcore in the Altera family of System On Programmable Chips.

1 Introduction

To satisfy tighter cost constraints and the increasing complexity of applications, embedded systems designers are moving towards platforms with multiple computational units. On the silicon technology side, the continuous scaling allows integration of functions in a system-on-chip with very high parallelism. This trend leads to consider multi-processor architectures, reconfigurable hardware and softcores (CPU implemented with FPGAs) as cost effective solutions.

In this paper we present the development of specialized arbitrator hardware for enabling the implementation of multiprocessor synchronization and resource sharing primitives with guaranteed maximum latency in the Erika Enterprise real-time OS.

The Altera Stratix and Cyclone SOPC are FPGA devices offering the opportunity for the implementation of one or more 32-bit embedded CPUs (or Softcores) called NIOS II with on-chip memory and a number of peripheral and communication devices in a reprogrammable form. The designer can develop his own custom peripherals and also his own Nios custom instructions by integrating custom logic inside the CPU. The interconnection bus provided in the Altera SOPC Builder toolset is the 32-bit wide Avalon Bus [2]. The bus avoids the typical bottlenecks of conventional symmetric architectures thanks to the Simultaneous Multi-Master concept. When generating the FPGA configuration code, SOPC Builder creates an arbitrator for each slave device shared among multiple masters. SOPC builder provides basic multiprocessor synchronization mechanisms, which can be made suitable to the requirements of embedded real-time systems with little extra programming. In this paper, we discuss the tradeoffs between the use of the standard Altera MUXTEX component and a suitably implemented specialized arbitrator.

2 Real-Time Multiprocessor Locking

The main disadvantage of standard test-and-set locking is contention over the shared bus. Every processor involved in a busy wait spins over the lock variable using bus bandwidth and delays all the others (including the processor that acquired the lock). Furthermore, it is impossible to bound the time spent by a processor when trying a lock.

![Figure 1. Architecture of the arbitrator.](image)

In our case, algorithms of type Spin-on-read or Spin-on-delay [1] are not suitable, since they require a cache consistency protocol, which is not available on Altera platforms. In Queuing Lock algorithms [3, 5], a dedicated queue (typically FIFO) keeps track of which CPU has attempted a lock on the variable. Every processor attempting a lock, enters the queue and starts spinning on a dedicated variable. When the processor holding the lock exits from the critical section, it is extracted from the queue and signals availability of the lock to the next processor. Among these, the G-T algorithm [4], has been selected for our implementation.

The required data structure (one for each lock variable) consists of a bit SPIN_BIT for each processor, a variable storing the identifier of the processor that performed the last attempt at locking the variable CPU_INDEX and one bit LOCK_VALUE containing the value indicating the "locked" status for the corresponding processor (Figure 1). Each processor performs the following instructions to try and eventually acquire the lock and later, to release it.

2.1 Implementation with the Altera MUXTEX

The first option is the use of the standard MUXTEX peripheral component provided by Altera. The MUXTEX peripheral makes a 32 bit register available to CPUs. Of the 32 bits, 16 are used to flag the locked
status and 16 are used to store the name of the CPU holding the lock. Whenever a CPU wants to acquire
the lock, it simply writes the register. The write operation succeeds in changing the register value only if
the status is unlocked. A successive read allows the
CPU to check for the success of the lock try. The only
drawback of the MUX peripheral is the lack of any
mechanism for enforcing a deterministic order (hence
predictability) in the locking of the mutex and subse-
quent access to the shared resource. To this purpose,
we implemented the queue part of the G-T algorithm
in a dedicated (shared) memory area. This solution is
clearly very efficient in terms of number of LEs (the
MUX peripheral is a truly small device, with a typ-
cal requirement of 96 LE).

2.2 Implementation using a custom arbitrator

To improve access times when trying the lock, we
implemented the G-T algorithm in a dedicated peri-
pheral using the Nios II custom instructions. Fig-
ure 1 represents an arbitrator managing up to 16 lock
variables (resources) accessed by up to 4 cpus and its
registers. On the upper-right corner, the figure shows
the connections between each CPU and the arbitrator.
CPUs access the shared memory or peripherals only
after receiving the arbitrator acknowledgement. Wait-
ning processors spin on a lock variable (SPIN_BITX)
located inside the arbitrator, which is designed to al-
low concurrent access in a single clock cycle. A rea-
time thread wishing to access a shared resource calls
a Lock function, with the lock index as a parameter,
to be later released with the explicit call to the Un-
lock function. The operating system-level C code
implementing the Lock function exploits three custom
instructions of the Extended Custom Instruction type.

\begin{verbatim}
lock(id)
int value, dest;
dest = GT_Swap(id);
do
  value = Spin_Read(id, dest & 0xffffffff);
  while((value == (dest & 0xffffffff)));
end
\end{verbatim}

GT_Swap(id) is used to try a lock on the corre-
sponding variable. The arbitrator adds the CPU
identifier to the FIFO queue and returns the iden-
tifier of the CPU that made the previous request and
the value indicating the locked condition for it. In
detail, the arbitrator substitutes the pair
CPU_INDEX, LOCK_VALUE, stored in its internal
SPIN_STATUS register, with the pair CPU_INDEX,
SPIN_BIT of the requesting CPU. The swap (and
enqueuing) is completed by returning the previous
value of SPIN_STATUS to the requesting CPU. The
GT_Swap instruction is the only one operating on the
shared registers and requesting atomic implementa-
tion of the swap of the register pair CPU_INDEX,
LOCK_VALUE. Hence, it is the one that required
the most care in the design and implementation. Initially,
we designed the arbitrator to handle the requests se-
quentially, serving one CPU request for each clock cy-
cle, but when more than one CPU requested a swap
in the same cycle, the arbitrator was forced to add
wait states before sending the acknowledgement. The
resulting implementation was a Multi-Cycle Custom
Instruction of variable length type. Next, extra logic
was added to allow parallel (single clock cycle) pro-
cessing of multiple swap requests from more than one
CPU. This required a static priority ordering of the
CPUs.

Spin_Read(lock.id, dest). The Spin_Read instruction
must be called inside the spin lock cycle. The instruc-
tion receives in input an integer value identifying one
of the SPIN_BIT locations. The instruction reads and
returns the value contained in the corresponding reg-
ister.

Spin_Flip(lock.id). The Spin_Flip instruction signals
the end of a critical section on a given resource. It flips
(one’s complement) the value of the SPIN_BIT bit for
the pair (CPU, resource) involved in the operation.

2.3 Implementation performance

The table shows the number of logic elements and
the area of the FPGA that is required to imple-
ment the data structures and the logic of the
GT_Swap instruction inside an Altera Stratix FPGA
(EP1S40F780C5 containing 41.250 LEs and 3.423.744
memory bits) for different pairs (number of cpus, num-
ber of lock variables).

<table>
<thead>
<tr>
<th>LEs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth.</td>
<td>48</td>
<td>137</td>
<td>269</td>
<td>552</td>
<td>1086</td>
<td>2338</td>
</tr>
<tr>
<td>fitting</td>
<td>39</td>
<td>111</td>
<td>217</td>
<td>448</td>
<td>879</td>
<td>1923</td>
</tr>
<tr>
<td>% area</td>
<td>0.09</td>
<td>0.27</td>
<td>0.53</td>
<td>1.09</td>
<td>2.13</td>
<td>4.66</td>
</tr>
</tbody>
</table>

References

[1] T. E. Anderson. The performance of spin lock alter-
 natives for shared-memory multiprocessors. In IEEE
Transactions on Parallel and Distributed Systems,
January 1990.

with the avalon bus. In Application Note 184, San
Jose, CA, April 2002.

[3] T. S. Craig. Queuing spin lock algorithms to sup-
port timing predictability. Proc. Real-Time Systems

 rithms for shared-memory multiprocessors. In IEEE
Computer, June 1990.

synchronization mechanisms for real-time systems.