An Interrupt Controller for FPGA-based Multiprocessors

Antonino Tumeo, Marco Branca, Lorenzo Camerini,
Matteo Monchiero, Gianluca Palermo, Fabrizio Ferrandi, Donatella Sciuto
Politecnico di Milano

Dipartimento di Elettronica e Informazione
Via Ponzio 34/5
20133 Milano
Italy
E-mail: {tumeo, monchier, gpalermo, ferrandi, sciuto}@elet.polimi.it

***Abstract*— Interrupt-based programming is widely used for interfacing a processor with peripherals and allowing software threads to interact. Many hardware/software architectures have been proposed in the past to support this kind of programming practice. In the context of FPGA-based multiprocessors this topic has not been thoroughly faced yet. This paper presents the architecture of an Interrupt Controller for a FPGA-based multiprocessor composed of standard off-of-the-shelf softcores. The main feature of this device is to distribute multiple interrupts across the cores of a multiprocessor. In addition, our architecture supports several advanced features like booking, broadcasting and inter-processor interrupt. On the top of this hardware layer, we provide a software library to effectively exploit this mechanism. We realized a prototype of this system. Our experiments show that our Interrupt Controller efficiently distributes multiple interrupts on the system.**

I. Introduction

Multiprocessors System-on-Chip (MPSoC) represent a new paradigm for processor design. This trend has been pursued by most semiconductor companies, both for high performance computers (e.g. the IBM Cell [1]), and complex embedded systems (e.g. ST Nomadik [2] and Philips Nexperia [3]). These systems typically consist of processors and several heterogeneous components, like peripherals, system buses and controllers.

The Interrupt Controller is a device commonly found in computer systems (both single-processor and multiprocessors), which deals with interrupts generated by the peripherals and the processors, handles the interrupt priorities, and delegates the execution to a processor.

Reconfigurable platforms have recently emerged as an important alternative to ASIC design, featuring a significant flexibility and time-to-market improvement with respect to the conventional digital design flow [4].

MPSoCs based on FPGA technology, composed of softcores have recently appeared [5]–[9]. This paper proposes an Interrupt Controller for this kind of systems. Our solution provides the same functional behavior of an Interrupt Controller for single-processor systems, but adds some features to exploit the CPU-level parallelism of a multiprocessor. In fact, a single-processor Interrupt Controller, if used in a multiprocessor environment, must be statically connected to a single processor, binding the management of any peripheral to a given processor.

Our design allows a peripheral to interrupt any of the processors of the system. In detail, our Interrupt Controller distributes the workload of the interrupt handling among the available processors. This feature decreases the interrupt management latency, because all requests are handled concurrently and not sequentially as in a single-processor system. Furthermore, the Interrupt Controller allows to delegate specific processors to handle interrupts generated by certain peripherals (booking), supports interrupt broadcasting and allows inter-processor interrupts.

These feature make several applications possible – for example, parallel processing of many interrupts coming from external devices (e.g. reactive systems), or peripheral sharing – since a peripheral can be forced to interrupt different processors.

We integrated the Interrupt Controller in CerberO [9]: a multiprocessor on FPGA composed of Xilinx MicroBlaze softcores. Our design overtakes several limitations of the standard Xilinx design flow, allowing for efficient interrupt management, otherwise not feasible.

The paper is organized as follows. Section II discusses some related works. The Interrupt Controller architecture is described in Section III. Section IV introduces the software layer that interfaces with the Interrupt Controller. Most relevant features are summarized in Section V, discussing some examples. Experimental results are discussed in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Interrupt management is a common design point in multiprocessor systems. In 1966 Goutanis et al. [10] clearly highlighted the related problems and the advantages offered by a correct solution in a multiprocessor setup.

A large part of the Intel Multiprocessor specifications [11] is devoted to describe how interrupts can be managed by the integrated Advanced Programmable Interrupt Controller (APIC).

In the context of embedded systems, interrupt management is crucial. These systems, usually composed of a general purpose processor and many specific Intellectual Property (IP) cores (peripherals and accelerators), use interrupts as a synchronization mechanism between the different processing elements and to react to external events. Reactive applications represent an important application domain, whose performance are tightly coupled with the interrupt management. These must often comply with real-time constraints which make efficient interrupt handling critical [12]. For this kind of systems, multiprocessor architectures are emerging as interesting solutions. For example, ARM [13] and IBM [14] offer specific designs that allow seamless management of interrupts on their embedded multiprocessor architectures.

Interrupt management is inherently important also on System-on-Chips developed with the Xilinx Embedded Design Kit (EDK) or Altera System-on-programmable-chip Builder (SOPC). Both companies provide comprehensive support [15], [16] to manage interrupts in a single processor system, but miss in providing a multiprocessor-aware controller.

LEON3 [8] is one the first examples of a design kit for softcore multiprocessor systems on FPGA, and the associated library [17] includes a multiprocessor interrupt controller. Nevertheless, this design differs from ours since it adopts several different mechanisms, like interrupt masking to distribute the signals to the different processors. So, unlike our solution – more hardware-oriented – LEON3 moves some complexity to the software.

In addition, LEON3 implements a full Sparc V8 architecture, which translates into a large area on the FPGA (approximately twice larger than many other softcores, like the Xilinx MicroBlaze and the Altera NIOS-II). On the other hand, our work explicitly targets Xilinx MicroBlazes providing a general module for this kind of systems, and filling a gap in the standard Xilinx toolchain. This makes possible realistic prototyping of a multiprocessor on FPGA (i.e. it would be difficult fit an equally sized LEON3 multiprocessor on our FPGA).

III. Interrupt Controller Architecture

The Interrupt Controller proposed in this paper is intended for integration in the CerberO MultiMicroBlaze architecture [9], as shown in Figure 1, while maintaining full compatibility with the Xilinx libraries. Each processor, and each peripheral implemented in the target system is connected to the controller which effectively acts as an interface between the general purpose and the application specific cores.

Figure 2 shows a high level block diagram of the Interrupt Controller architecture. It is composed by two main components: the first one (*OPB Interface*) is the block which interfaces the core with the main bus of the system, the second one (*IntC Core*), implements the true logic of the controller.

A. Interface

The interface is based on the On-chip Peripheral Bus (OPB) specification [18] to allow memory mapped access from the processors. The interface send the following signals to the controller logic:

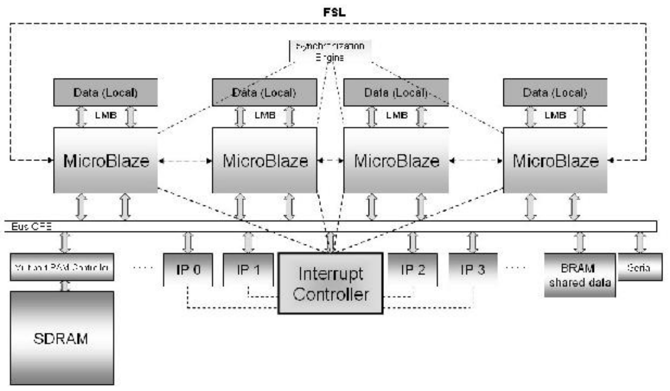

Fig. 1. The CerberO MultiMicroBlaze architecture with the Interrupt Controller

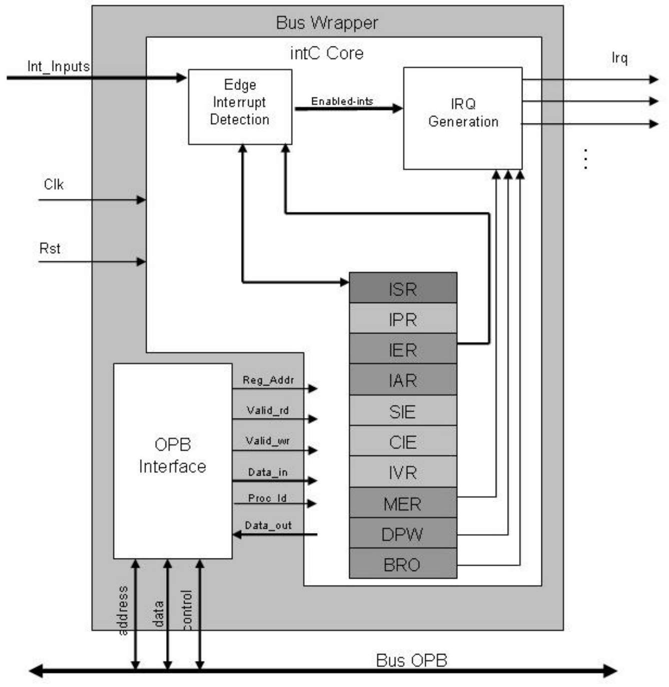

Fig. 2. Interrupt Controller Architecture

- *Reg_Addr*: the register address where the processor wants to write to or to read from
- *Valid_rd*: read operation from the register specified by *Reg_Addr*
- *Valid_wr*: write operation towards the register specified by *Reg_Addr*
- *Data_in*: the data to write
- *Data_out*: the data to read
- *Proc_id*: the identification of the processor accessing the controller

The last signal allows to associate each event to and from the Multiprocessor Interrupt Controller with a corresponding source processor, so it is essential for handling interrupts in a MultiMicroBlaze architecture.

B. Controller Logic

The second component, *IntC Core*, implements the core logic of the controller. As the picture shows, it is composed by three sub-components, each one with a specific purpose:

- The *Register Block* provides a useful support for all the main facilities of our Interrupt Controller. It allows handling of interrupt priorities, deciding which interrupts can be handled and managing of interrupt acknowledge signals. To allow each processor to respond only to interrupts generated by specific IPs (*booking*) and to allow *broadcasting* of interrupts to all the processors in the system (i.e. all the processors receive the same interrupt), the register offered by the standard Xilinx [15] implementation have been extended.
  Our design implements the following registers:
  - *Interrupt Status Register* (ISR) stores the active interrupts
  - *Interrupt Pending Register*(IPR) stores the interrupts that are both active and enabled
  - *Interrupt Enable Register* (IER) keeps track of which interrupts are allowed to be handled
  - *Interrupt Acknowledge Register* (IAR) is a support for disabling interrupts that receive the corresponding acknowledgment signal
  - *Set Interrupt Enable* (SIE) is a support for writing into IER
  - *Clear Interrupt Enable* (CIE) is a support for deleting from IER
  - *Interrupt Vector Register* (IVR) contains the identification of the next interrupt that must be served.
  - *Master Enable Register* (MER) is used for enabling our Interrupt Controller to manage interrupts
  - *Device Processor Waiting* (DPW) is a support to store which interrupts have been booked
  - *Broadcast* (BRO) is a support to store which interrupts have to be handled in broadcast way
- *Edge Interrupt Detection* detects interrupts coming from the IPs of the system and activates the logic which communicates pending requests to the connected processors. In particular, it monitors the signals vector *Int_Inputs* composed of all interrupt lines coming from the cores, and rises the *Enabled-Ints* signal according to the arriving signals.
- *Irq Generator* contains the generation logic of the interrupt signals toward processors.

Figure 3 shows a block diagram representing the internal architecture of Irq_Generator. It is composed of two main sub-blocks. *Broadcast Interrupt Generator* generates signals in case of interrupts that need to be propagated to all processors. *Standard Interrupt Generator* handles all the other kinds of interrupt.

Each of the two modules is active in mutual exclusion. The signal *Broadcast_logic_Enable* is active until all processors complete the handling of the broadcast interrupt requests. This means that broadcast events have the highest priority in our

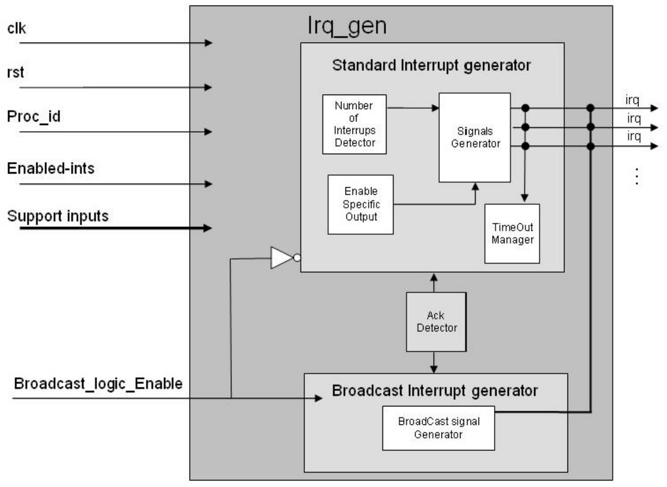

Fig. 3. Detailed block diagram of Irq Generator

design. In our opinion, this could be a reasonable design choice for real-time system. In fact, it is fair to suppose that a broadcast event implies that all processors have to answer to this event as fast as possible. If the system handled other interrupts during a broadcast event, some processors would never receive the broadcast interrupt.

The block *Standard Interrupt Generator* in Figure 3 is composed of:

- *Number of Interrupts Detector* useful to count the interrupts that still have to be handled
- *Timeout Manager* manages delays of incoming acknowledge signals. In order to dynamically allocate the handling of the interrupts to all available processors and allow real time responsiveness, if a processor is not able to acknowledge an interrupt before a pre-determined deadline, the Multiprocessor Interrupt Controller will forward the signal to another processor.
- *Enable Specific Output* allows a processor to wait for interrupt from a specific device
- *Signal Generator* forwards the interrupt signals to the processors

Finally, Figure 3 shows the *Ack Detector* block, which receives the interrupt acknowledges from the processors. When the *Broadcast Interrupt Generator* is enabled, this component waits for the acknowledge of the same interrupt from all the processors, blocking all the operations until this condition has been satisfied.

C. A Simple Example

With the purpose to better understand the behavior of the proposed architecture, it is useful to introduce a simple example which shows how the component reacts to the arrival of multiple interrupts in a short time interval.

Figure 4 shows the following:

1) Two application specific cores connected to the Interrupt Controller generate interrupts in the same time interval.

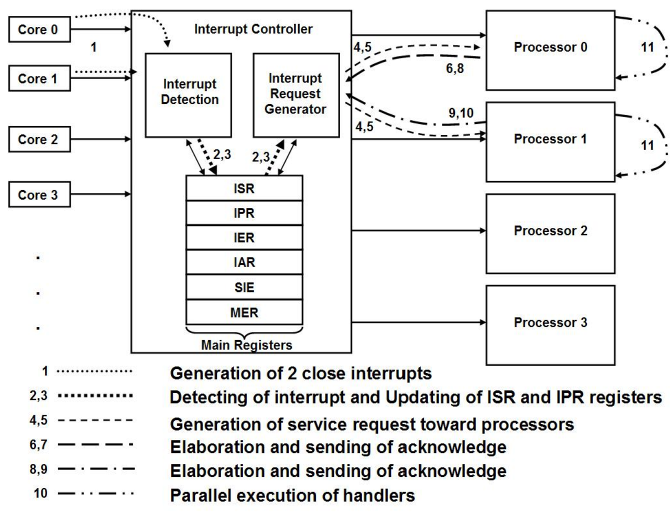
Core 0
Core 1
Core 2
Core 3
1
Interrupt Controller
Interrupt Detection
Interrupt Request Generator
2,3
2,3
ISR
IPR
IER
IAR
SIE
MER
Main Registers
4,5
6,8
9,10
4,5
Processor 0
Processor 1
Processor 2
Processor 3
11
11
1 Generation of 2 close interrupts
2,3 Detecting of interrupt and Updating of ISR and IPR registers
4,5 Generation of service request toward processors
6,7 Elaboration and sending of acknowledge
8,9 Elaboration and sending of acknowledge
10 Parallel execution of handlers

Fig. 4. Flow of parallel interrupt handling

Interrupt management for these two cores has been previously enabled by writing on the IER register

2) The interrupts are detected by the *Edge Interrupt Detection* module
3) The *Edge Interrupt Detection Module* stores the arrival of the interrupts in ISR register. Writing on ISR also influences the content of the IPR register
4) *IRQ Generation* raises two outputs toward two free processors (not handling other interrupts)
5) The Interrupt Controller waits for acknowledgment
6) The first processor which detects the interrupt request chooses the active interrupt with the highest priority and sends the related acknowledgment.
7) The first processor starts executing the handler associated to the selected interrupt.
8) The second processor which detects the interrupt request chooses the second active interrupt and sends the related acknowledgment.
9) The second processor starts executing the handler associated to the selected interrupt.
10) From now on, the two processors will concurrently handle the two generated interrupts.

IV. Software Layer

The software layer is composed by a driver and a generic interrupt handler. The driver exposes the Interrupt Controller features to the processors. It contains three main functions to implement booking, broadcasting, and inter-processor communication. The next section discusses the detailed behavior of our architecture when dealing with these features. The generic interrupt handler is called *DeviceInterruptHandler* and executes whenever a processor receives an interrupt request. This routine chooses which is the interrupt to handle by reading the Interrupt Controller registers, acknowledges the Interrupt Controller and launches the specific interrupt handler to manage the event. If it is an interrupt from a peripheral, the handler included in the peripheral driver is executed. If it is an inter-processor interrupt, an appropriate handler is launched.

Notice that the accesses to the Interrupt Controller from multiple processors must be exclusive. This is ensured by the generic handler which properly uses lock-protected critical sections to correctly perform the access to the Interrupt Controller registers. In addition all software (handler and driver) is shared, thus residing in the same memory segment. It is fetched to the local caches when needed. On the other hand, the local data stack grows in the local data memories.

The CerberO architecture [9], to which the Multiprocessor Interrupt Controller is targeted, provides full, seamless support for these aspects.

V. Booking, Broadcasting, and Inter-processor Communication

In this section, we describe the behavior of our Interrupt Controller in our multiprocessor system in order to show how it provides functions described in the previous sections. First of all we need to introduce the architecture of the system we will use for our examples. The system we refer to is composed of four processors, each with its own *Identification* (Id). Besides, there are several cores able to generate interrupts and a single common bus which is used for all communications among processors and cores. Finally there is a set of local memories, one for each processor, and a shared memory, where benchmark code resides.

a) Booking: Consider a processor which offloads some work to an accelerator. Once the accelerator completes, it must signal to the processor who requested the job. In a multiprocessor environment, the Interrupt Controller could send the interrupt generated by the accelerator to any processor, unless the processor books the interrupt/accelerator.

In our system, this is accomplished by calling the *Book_Peripheral* primitive, and specifying which is the peripheral to book. Once received this request, the Interrupt Controller keeps track of two basic pieces of information: the identifier of the peripheral which has been booked (writing the DPW register), and the Id of the booking processor.

Once this information have been registered, the processor can go on doing some work, without waiting for the results computed by the accelerator. In fact, when results will be available, the accelerator will generate an interrupt.

While the interrupt signal coming from the booked peripheral is being handled, the Interrupt Controller disables all interrupt requests towards processors which do not correspond to the booking one. Thanks to this trick, we are sure that only the booking processor will handle the interrupt. So, this is the only processor which will search the interrupt to handle, when executing the *DeviceInterruptHandler*.

b) Broadcast: Any peripheral can be configured to generate *broadcast* interrupts. This kind of interrupt is distributed by the interrupt controller to all processors, unlike a conventional interrupt which is sent to a single processor. Our software driver allows to configure any peripheral to generate broadcast interrupts. For example, this can be done during the startup of our system, or whenever needed. The addresses of the selected peripherals are registered by writing the BRO register. In this way, the Interrupt Controller will know exactly how an interrupt must be served.

Suppose that a peripheral generates an interrupt to broadcast. Our Interrupt Controller realizes that the interrupt must be served as a broadcast signal. It enables the broadcast logic, disables the logic that control the normal interrupt flow and then it raises all the outputs towards the processors. Any processor, once detects the interrupt, starts communicating with the Interrupt Controller, searching for the interrupt with the highest priority to serve. The processor realizes that a broadcast interrupt has been raised and, since the broadcast interrupts have the highest priority, selects the appropriate handler to serve it.

c) Inter-processor interrupts: Inter-processor interrupts (or software interrupts) are used to make processors communicate. For example, this feature can be extremely useful to implement mutual exclusion on a peripheral. Suppose that a processor exclusively owns a peripheral. Once the peripheral is not needed anymore, a broadcast interrupt can be generated to notify all processors, which can thus compete to acquire the peripheral. The advantage of this scheme with respect to a lock-based one, is that, while the peripheral is busy, the processors are spinning on a local variable, rather than on the global lock. In detail, to implement such a scheme, a global lock is needed to handle the contention for the peripheral, and each processor executes an algorithm as follows:

1) Acquire the lock; do just a Test-and-Set, if not succeed go on to 2(b);
2) (a) Book the peripheral, (b) while other processors wait on a local variable;
3) The peripheral completes and notifies the processor;
4) The processor broadcast an interrupt, which sets the local variable to makes all contender to go on;
5) go back to 1, until all contending processors have used the peripheral.

Notice that inter-processor interrupts are supported in multicast, as well – i.e., a processor can generate a software interrupt signal to multiple processors at the same time.

VI. Evaluation

This section discusses some experimental results obtained integrating the Interrupt Controller in the CerberO architecture. We first present some data about the area occupation, and then some experiments on a simple micro-benchmark.

A. Area Occupation

Figure 5 shows the occupation of the Multiprocessor Interrupt Controller compared to the the single processor standard Xilinx solution. The modules have been synthesized with Xilinx ISE 8.2 targeting the Xilinx XC2VP30 FPGA. Consider that a MicroBlaze (version 5.00c w/o floating point unit) occupies 1332 slices. We evaluated the area of our device varying the number of output lines to processors (P) and the number of input lines from peripherals. Notice that the area linearly

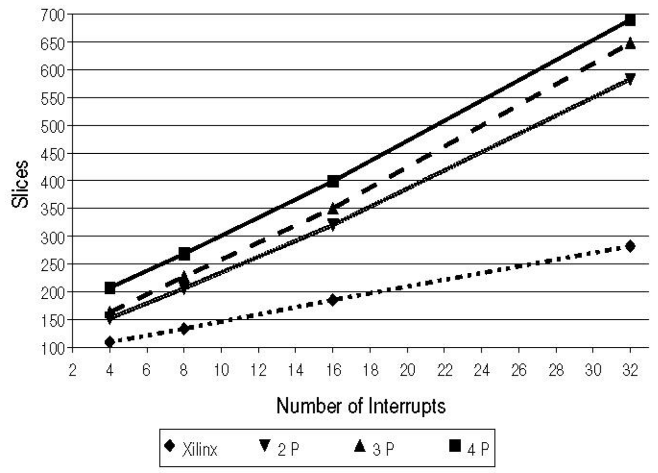

Fig. 5. Occupation in terms of slices of the proposed design on Virtex II-Pro XC2VP30 scaling the number of processor outputs and the number of interrupt inputs

depends on the latter figure, while is much less sensitive to the number of outputs. The overhead with respect to the single-processor implementation grows linearly with the number of IP cores. When 16 peripherals able to generate interrupts and 4 receiving processors are connected the Interrupt Controller occupies the 30% of a single MicroBlaze.

B. Experiments

The micro-benchmarks used to evaluate the Interrupt Controller consist of multiple simultaneous interrupts generated by multiple (one to four) Xilinx OPB Timer/Counter. The timers are connected to our Interrupt Controller and implemented within the CerberO architecture (2 to 4 processors). In order to provide realistic results, the interrupt handlers of several application specific accelerators have been profiled and handlers with the same delays have been produced.

We adopted the following handler delays:

- *0 cycles*: the latency between the raising of an interrupt signal and the launch of the interrupt service routine
- *100 cycles*: a small size handler that reads or writes 16 32-bit registers from a peripheral connected to the OPB bus
- *200 cycles*: a small size handler that reads or writes 32 32-bit registers from a peripheral connected to the OPB bus
- *1000 cycles*: a medium size handler and it is comparable to copy some registers to the external DRAM through a memory controller connected to the OPB bus
- *10000 cycles*: a huge handler that accesses many registers on a peripheral and transfers their contents in external memory

Figure 6 shows the results obtained executing concurrent handlers of the same duration. This emulates a heterogeneous multiprocessor architecture with many identical application specific accelerators.

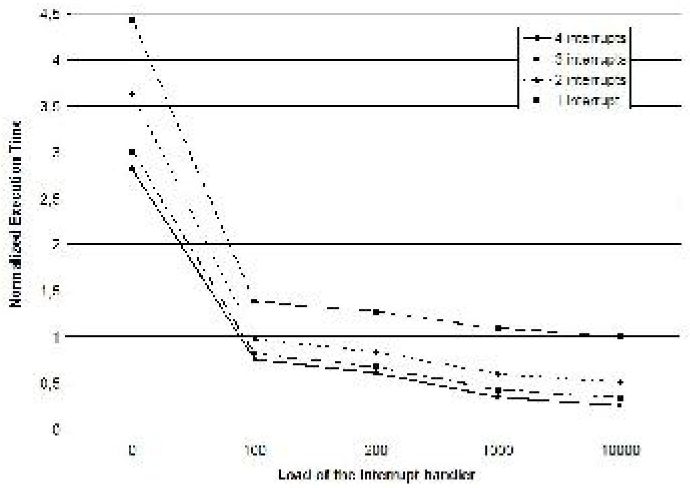

Fig. 6. Normalized execution time of concurrent handlers of the same duration

On the X-axis is reported the duration of the concurrent interrupt handlers, while the Y-axis corresponds to the measured delay from interrupts arrival to handler exit normalized with reference to the values obtained with the standard Xilinx controller. The line marked with 1 corresponds to the results of the Xilinx core: values falling under this line are better with respect to the single processor interrupt handling mechanism.

When handling a single interrupt, the proposed design suffers from the bus contention and the synchronization overhead required to access the module in the multiprocessor system. The smaller the handler, the bigger the overhead, and, since there is a single interrupt to handle, no advantages can be obtained from the parallel nature of the system. As the number of simultaneous interrupts grows, the Multiprocessor Interrupt Controller takes advantage of the parallelism. Similarly, as the handlers require more time to execute, better the results are, since the handler overhead (locking) is easier hidden by the parallelism.

Figure 7 shows the results obtained with the parallel execution of handlers of different duration. The considered handlers have delays between 100 and 400 cycles, emulating

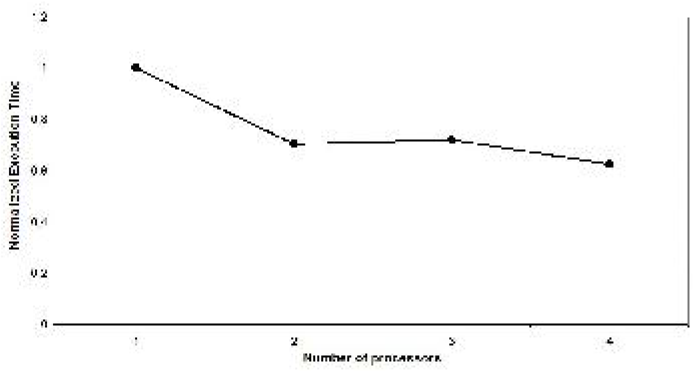

Fig. 7. Results obtained on architectures with a varying number of processor receiving interrupts connected to 4 IPs and non-uniform latency handlers

an application specific architecture with several homogeneous general purpose microprocessors with many different application specific cores. The data are relative to architectures with a different number of processors receiving interrupts from 4 peripherals, and normalized with respect to a single processor architecture with the standard Xilinx core. The results show that the proposed design allows to fully exploit the parallelism available with the multiprocessor platforms. Since the duration of the handlers is non homogeneous, the results are limited by the longest one. Nevertheless, the results show that with multiple interrupts continuously coming, the Multiprocessor Interrupt Controller permits to fairly distribute the handling of the events reducing the latency and thus augmenting the responsiveness of the system.

VII. Conclusions

This paper presented the architecture of an Interrupt Controller for a FPGA-based multiprocessor. Our design efficiently distributes multiple interrupts on a multiprocessor, exploiting CPU-level parallelism. In addition, it supports several features useful in a multiprocessor system, like booking, broadcasting and inter-processor interrupt. We integrated the Interrupt Controller in CerberO, a FPGA-based multiprocessor, and showed that it is fully working and allows effective performance scaling when multiple interrupts are handled.

References

[1] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[2] Nomadik - Open multimedia platform for next generation mobile devices. Technical Report TA305, 2004.

[3] Philips Semiconductor. highly integrated, programmable system-on-chip (SoC). available at http://www.semiconductors.philips.com/products/nexperia.

[4] Frank Vahid. The softening of hardware. *Computer*, 36(4):27–34, 2003.

[5] P. James-Roxby, P. Schumacher, and C. Ross. A single program multiple data parallel processing platform for fpgas. *2004. FCCM 2004. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 302–303, 2004.

[6] C.R. Clark, R. Nathuji, Lee, and S. Hsien-Hsin. Using an fpga as a protyping platform for multi-core processor applications. In *Workshop on Architecture Research using FPGA Platforms, 2005*, 2005.

[7] A. Hung, W. Bishop, and A. Kennings. Symmetric multiprocessing on programmable chips made easy. *Design, Automation and Test in Europe, 2005. Proceedings*, pages 240–245, 2005.

[8] Leon3 Processor. available at http://www.gaisler.com.

[9] Antonino Tumeo, Matteo Monchiero, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. A design kit for a fully working shared memory multiprocessor on fpga. In *GLSVLSI '07: Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI*, pages 219–222, New York, NY, USA, 2007. ACM Press.

[10] R. J. Gountanis and N. L. Viss. A method of processor selection for interrupt handling in a multiprocessor system. *Proceedings of the IEEE*, 54(12):1812–1819, December 1966.

[11] Intel. *MultiProcessor Specification, version 1.4*, May 1994.

[12] Huaidong Shi, Ming Cai, and Jinxiang Dong. Interrupt synchronization lock for real-time operating systems. In *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, pages 171–171, September 2006.

[13] ARM11 MPCore. Available at http://www.arm.com.

[14] IBM. *Multiprocessor Interrupt Controller Data Book*, March 2006.

[15] Xilinx. *OPB Interrupt Controller (v1.00c)*, January 2005.

[16] Altera. *Nios II Processor Reference Handbook*, November 2006.

[17] Gaisler Research AB. *GRLIB IP Core Users Manual*, December 2006.

[18] Xilinx. *OPB Arbiter (v1.02e)*, September 2005.