Operating System for Symmetric Multiprocessors on FPGA

Pablo Huerta, Javier Castillo, Carlos Sánchez, Jose Ignacio Martínez
Rey Juan Carlos University, Madrid, Spain
{pablo.huerta, javier.castillo, carlos.delalama, joseignacio.martinez} @ urjc.es

Abstract

Soft-core based multiprocessor systems are getting very popular in the FPGA design world. There are many computer architectures that has been used for building multiprocessor systems on FPGAs, including SMP (Symmetric MultiProcessor). One of the main drawback of this SMP systems is the unavailability of operating systems that allow programming multi-threaded applications that make good use of the multiple processors of the system. This paper details an operating system designed to be used with SMP systems based on the MicroBlaze soft-core processor. The OS is tested with three different applications on an SMP system which implements all the software and hardware required for the OS to work on different SMP systems.

1. Introduction

Soft-core processors (SCP) are becoming so popular in FPGA design that many FPGA vendors offer SCPs optimized for their FPGAs: Xilinx provides the PicoBlaze and MicroBlaze, Altera the Nios II, Lattice the LatticeMico32, etc. There are also other SCPs provided by third parties such as the LEON 2 by Gaisler Research, or the OpenRISC 1200 by OpenCores.

The number of SCPs that can be used in an FPGA system is only limited by the number of device resources (logic elements and memory). As this number increases rapidly, designers can implement more complex multiprocessor architectures every time.

This paper is focused on Symmetric Multiprocessor systems (SMP) based on the SCP MicroBlaze. An SMP is a system where two or more identical processors are connected to a single centralized memory and I/O interface. The term "symmetric" is used because all the processors use the same mechanism to get access to the memory and peripherals.

One of the main drawbacks of these SCP systems is the absence of operating systems with symmetric multiprocessing functionalities that allow multithreaded applications to run on several processors.

In this paper an operating system with SMP functionality, and the hardware needs to build a SMP system with MicroBlaze soft-core processors will be presented.

This paper shows an operating system with SMP functionality and all the hardware needed to build an SMP system with MicroBlaze soft-core processors.

The paper presents in Section 2 the context work on the field of SMP systems on FPGA. Section 3 describes the OS architecture and the hardware abstraction layer interfacing with the underlying hardware components for the OS to provide SMP functionality. A sample application for testing the OS functionality is presented in section 4. Finally, the conclusions and future work stand in section 5.

2. Related and previous work

There are many implementations of multiprocessor systems on FPGA, but most of them designed for very specific applications for instance, [1][2] for MPEG encoding, [3] for matrix intensive applications or [4] for Internet Protocol packet processing. In the SMP systems field

There are also works addressing specific challenges for SMP systems on FPGA. For instance, [5][6] shows an SMP system based on the SCP Nios II solving cache incoherences thanks to an specific hardware module. In [7] an SMP system based on MicroBlaze processors deals with interprocessor communication and synchronization, adding a crossbar and a synchronizing element, both connected through the FSL interface of the processors. In the software area the authors present a nanokernel that allows scheduling threads in the different processors, but with some limitations: once a thread is scheduled in a processor it will run until finishes on this processor, and no more than one thread can be assigned simultaneously to each processor.

In previous work [8] we developed a library that allows some kind of SMP functionality. This library

978-0-7695-3474-9/08 $25.00 © 2008 IEEE
DOI 10.1109/ReConFig.2008.43

allows scheduling tasks in different processors in a multiprocessor system, but it has limitations such as the impossibility of migrating tasks between processors and the need of a local copy of the OS in each processor (memory waste). This system allowed us to explore different multiprocessor architectures and understand the hardware and software requirements for an OS with true SMP functionality.

Based on that experience, an OS with full SMP functionality has been developed for SMP systems based on the Microblaze soft-core processor. The OS allows running multi-threaded applications in a transparent way to the application programmer. The threads are scheduled in the different processors depending on the system load, with no limitations to moving a thread from one processor to another. Classical mechanisms for communication and synchronization between processes are supported, for example semaphores, mutex, or message queues.

3. OS architecture

The OS is based on xilkernel, the microkernel provided by Xilinx to work with its soft-core processors [9]. This section describes the OS architecture in detail.

3.1 Hardware requirements

The OS runs properly when the SMP system fulfills several hardware requirements:
- A shared memory region mapped on the same addresses for all the processors, and big enough to store both the OS and the user applications.
- A small private memory region for each processor to be used as an stack when running kernel functions.
- A CPU identifier mechanism that allows the OS to know which processor is executing any specific code.
- A hardware synchronization mechanism that allows the processors to get exclusive access to critical code sections.

The way these and other features are implemented must be described in a Hardware Abstraction Layer (HAL) in order to separate the functional side of the OS from the hardware-dependant side. The HAL will also be very helpful when porting the OS for an specific SMP system.

The features described in the HAL and their meanings are:
- NUM_CPUS: number of processors in the system.
- SHARED_MEMORY_BEGIN: address of the shared memory section.
- SHARED_MEMORY_SIZE: size of the shared memory section.
- PRIVATE_MEMORY_BEGIN: address of the private memory section of each processor.
- PRIVATE_MEMORY_SIZE: size of the private memory section of each processor.
- GET_PROCESSOR_ID(VAR): macro returning a processor identifier.
- HARDWARE_MUTEX_LOCK/UNLOCK(NAME): macros for locking/unlocking the hardware synchronization mechanism.

Figure 1 and Table 1 show both the system that will be used for testing the OS and the HAL of this system. The system is made up of:
- 4 MicroBlaze soft-core processors v4.0.
- 128 KB of blockram shared through an OPB bus.
- 1 KB of blockram for each processor connected through an LMB bus.
- A hardware synchronization mechanism, as in [8].
- A register with the CPU identifier accessible through an FSL interface for each processor.
- A timer for the processors.
- An UARTLITE peripheral for standard input and output through a terminal.

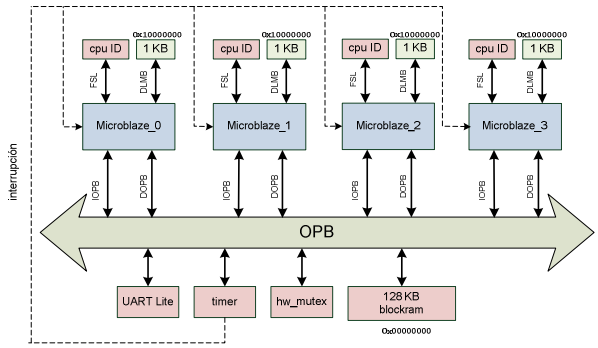

Figure 1: SMP system for the OS test

With the descriptions in the HAL the OS knows how to perform the different operations such as scheduling processes in the processors, intercommunicating and synchronizing processes, and all the other operations that involve more than one processor.

```
#define NUM_CPUS 4
#define SHARED_MEMORY_BEGIN   0x00000000
#define SHARED_MEMORY_SIZE    0x00200000
#define PRIVATE_MEMORY_BEGIN  0x10000000
#define PRIVATE_MEMORY_SIZE   0x00000400
#define GET_PROCESSOR_ID (VAR)        \
       microblaze_bread_datafsl(VAR, 0);
#define HARDWARE_MUTEX_LOCK           \
       hw_mutex_lock(0x10010000);
#define HARDWARE_MUTEX_UNLOCK \
       hw_mutex_unlock(0x10010000);
```

Table 1: HAL for the OS test

3.2 Internal data structures

The operating system maintains different data structures with the information of all the processes and processors active in the system. The main structure is the *process_struct*, which stores the following information about processes:

- *process_context*: state of the processor that was running the process the last time it was scheduled out or interrupted. It includes the values of the 32 general purpose registers plus the state machine register.
- *state*: state of the process (new, run, ready, wait, dead).
- *pid*: process identifier.
- *priority*: priority of the process when a priority based scheduling scheme is used.
- *is_allocated*: shows if this *process_struct* is in used or free for a new process.
- *block_q*: if the process is blocked, this variable shows in which queue it is blocked.

The OS references the *process_structs* with two different tables, as shown in Figure 2. The first table is the process table (*ptable[]*), an array that stores as many *process_structs* as indicated by a compile time parameter of the OS (MAX_PROCS). The second table is *current_process[]*, an array with as many entries as processors in the system, storing a pointer to the *process_struct* of the process of each processor.

3.3 System boot

During the system boot the OS has to configure some hardware elements such as the interrupt controller and the timer, and also some data structures and queues have to be initialized such as the process table, the scheduler's queue and some optional modules of the OS: semaphores, mutex, message queues, etc. These taks are carried out by only one of the processors in the system, named bootstrap processor, described as cpu_id equal to 0. During this process the other processors will be waiting for a signal to start running.

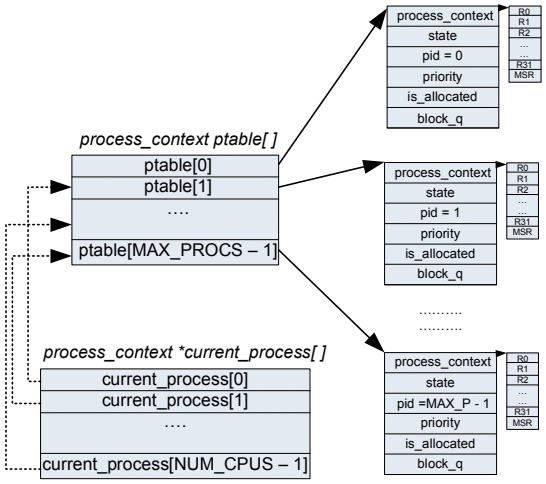

Figure 2: Data structures used by the OS

The bootstrap processor's startup sequence is:

- Timer initialization.
- Initialization of data structures for processes.
- Scheduler's queue initialization.
- Initialization of the *pthread* module, and other optional modules (*semaphores, mutex, message queues, etc*).
- Initialization of a dummy task created for keeping the processors working whilst no other processes are ready to run.
- All static threads defined by the user are created and inserted in the scheduler's queue.
- Enabling interrupts and indicating the rest of the processors to enable their interrupt managers.

From then on, when the first timer interrupt arrives the processors start running the processes that are ready in the scheduler's queue.

3.4 Interrupt handling and process scheduling

The operation of the interrupt handler is similar to the classical interrupt handling: disabling interrupts, saving the state of the processor, switching to the kernel stack, calling the specific interrupt handler, restoring the processor state and re-enabling interrupts.

Because there is more than one processor in the system, the interrupt handling is a bit more complicated than in a single processor system. If all processors are connected to the same interrupt signal, all of them will be interrupted at the same time and this will lead to having all processors trying to handle the interrupt simultaneously. For interrupt sources other than the timer, once the interrupt has been handled

there is no need to be handled again by other processor, this will lead to a wrong behavior of the whole system.

For this purpose we have introduced a simple peripheral that demultiplexes the interrupt line in as many lines as processors in the system (Figure 3). Each time an interrupt arrives it is forwarded only to the appropriate processor, and if the interrupt source is the timer a counter is incremented when the interrupt is cleared (falling edge). The counter selects the processor for the next interrupt.

This peripheral avoids simultaneously interrupting all the processors and simplifies the interrupt handler code because is not necessary to check if the interrupt was already handled by another processor.

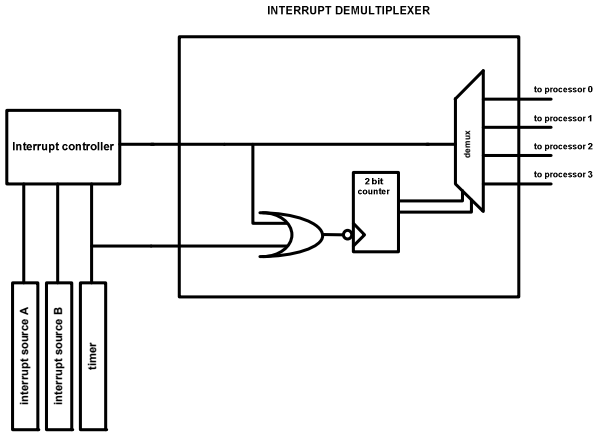

Figure 3: Interrupt demultiplexer peripheral

The timer interrupt handler is responsible for making the context switches by calling the scheduler function. Figure 4 shows an example of the way it works:

- First, the OS needs to know which processor is handling the interrupt by calling the appropriate HAL macro.
- Once the OS knows which processor is executing the handler, stores the processor state in the *process_struct* of the process that was being executed by the processor when the interrupt arrived.
- Next, switches to the kernel stack and calls the scheduler function.
- Once the scheduler returns, the *process_struct* of the scheduled process will be available through the *current_process[i]* pointer.
- The state of the processor is restored with the information of the scheduled process.
- The system returns from the interrupt.

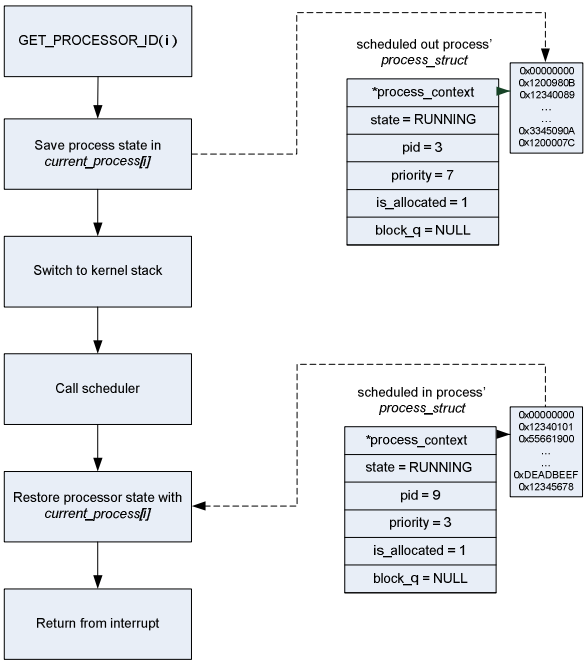

Figure 4: Timer interrupt handler example

As explained in section 3.1, a hardware synchronization mechanism is needed to allow processors to get exclusive access to critical sections of code.

One of these critical sections is the scheduler's queue. While the scheduler is getting access to its queue, other processes running in other processors may also be trying to access the same queue through a system call like *pthread_create, pthread_join,* etc. The access to that shared kernel data must be exclusive for one process at a time. In a single processor system this can be done by deactivating interrupts during these operations, but in a multiprocessor system some sort of support is needed in the form of a hardware peripheral.

All the functions in the OS that have been modified to use the hardware_mutex for synchronizing the access to shared kernel resources (for instance: queues, software semaphores, software mutex, message boxes, etc.) are the scheduler function and all the system calls. Figure 5 shows the flow graph of the scheduler function including the use of the hardware mutex for accessing shared kernel resources.

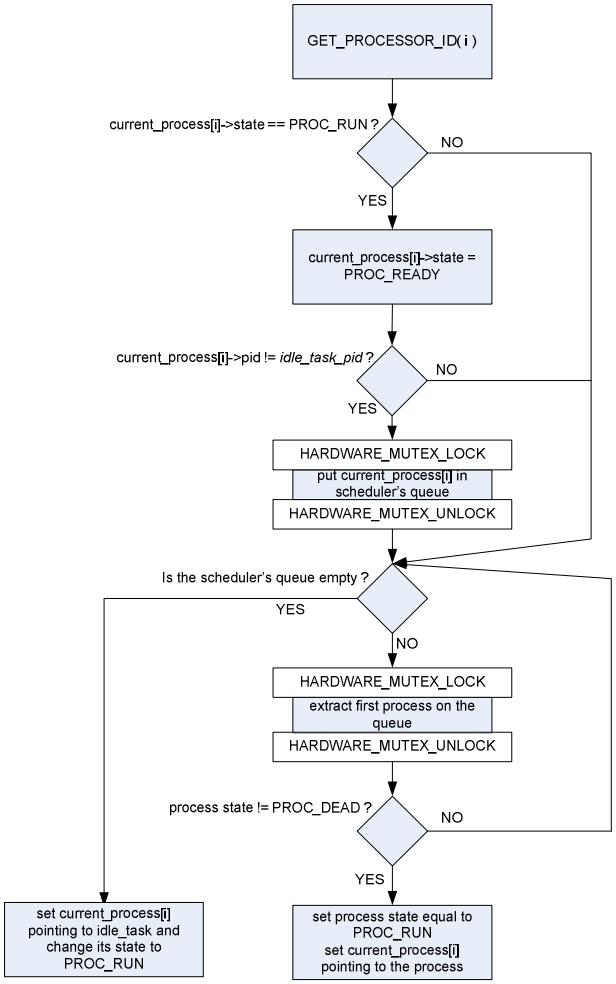

Figure 5: Scheduler function flow diagram

4. Experimental results

The system described in Figure 1 was implemented in a Celoxica's RC300 board on an Xilinx XC2V6000-FF1152-4 FPGA.

The developed OS was tested in the system running the application described in Table 2. The application creates several threads that display messages on a terminal showing the processor that is executing the thread and the thread identifier, both at the beginning and the end of the thread. During the execution many timer interrupts happen with their associated scheduler calls, therefore each thread is partially executed by a different processor. A sample output of the execution is shown in Figure 6.

```
#define NUM_THREADS 8

pthread_t p_th[NUM_THREADS];
sem_t   sem_1;

void main_thread(void)
{
   sem_init(&sem_1, NULL, 1);
   sem_wait(&sem_1);
   print("Creating threads\r\n");
   sem_post(&sem_1);
   for(i = 0; i < NUM_THREADS; i++)
     pthread_create(&p_th[i], NULL, \
      (void*)thread_1, NULL);

   for(i = 0; i < NUM_THREADS; i++)
     pthread_join(p_th[i], NULL);

   sem_wait(&sem_1);
   print("All threads finished!\r\n");
   sem_post(&sem_1);
}

void thread_1(void)
{
   int i, processor_id, thread_id;
   thread_id = pthread_self();

   sem_wait(&sem_1);
   GET_PROCESSOR_ID(processor_id);
   xil_printf("Starting: thread_id  \
      = %d\t processor_id = %d\r\n", \
     thread_id, processor_id);
   sem_post(&sem_1);

   for(i = 0; i < 5000000; i++);

   sem_wait(&sem_1);
   GET_PROCESSOR_ID(processor_id);
   xil_printf("Exiting:  thread_id = \
     %d\t processor_id = %d\r\n", \
     thread_id, processor_id);
   sem_post(&sem_1);

   pthread_exit(NULL);
}
```

Table 2: Sample application to test the OS

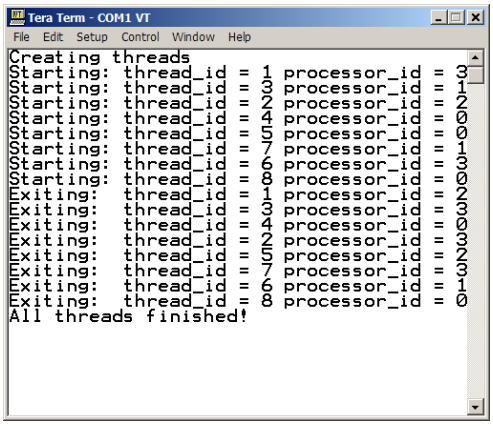

Figure 6: Sample execution of a multithreaded application

Two other applications were used for testing the performance that can be reached when executing parallelizable applications. One of the applications performs matrix multiplications using a parallel algorithm that divides the work between the available processors in the system, each one computing a fraction of the resulting matrix. The other application encrypts and decrypts data blocks of 8 Kbytes using the AES algorithm, dividing the data between the available processors.

For running this applications 4 KBytes of instruction cache were added to the processors. Data caches are not used since no cache coherency is available yet on MicroBlaze processors.

The results of the applications running on 1, 2, 3 and 4 processors are shown on Figure 7. We define efficiency as the speedup relative to the FPGA resources used by the system. Table 3 shows the LUT usage of the system for 1, 2, 3 and 4 processors.

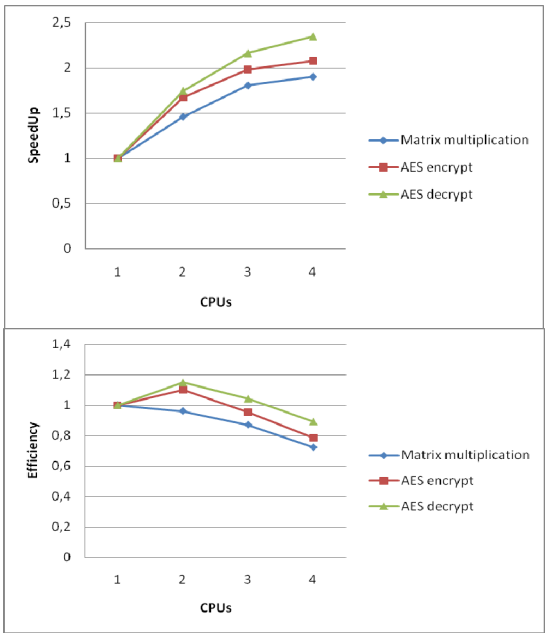

Figure 7: Speedup and efficiency for matrix multiplication and AES cryptography

The results show a maximum speedup of 2,4 for the AES decryption and 4 processors. Speedup doesn't grow linearly due to bus congestion when increasing the number of processors. Better results would be achieved if data caches could be used.

5. Conclusions and future work

An OS system with SMP functionalities for being used in SMP systems based on the MicroBlaze soft-core processor has been presented. Hardware and software requirements and solutions for this kind of systems has been described, as well as an example of implementation for different applications.

CPUs	LUTs	Ratio
1	2329	1
2	3539	1,5195
3	4834	2,0756
4	6137	2,6350

Table 3: system LUT usage

Ongoing work on this field is split into two directions. One is porting the OS to work with the 2 PowerPC processors of some FPGAs from the Virtex 2 Pro and Virtex 4 FX families. The second is to evaluate the performance of different SMP systems with different memory hierarchy architectures when executing different parallel applications.

6. References

[1] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen and T.D. Hamalainen, "A Parallel MPEG-4 Encoder For FPGA Based Multiprocessor SOC". Proceedings of the International Conference on Field Programmable Logic and Applications. 2005. pp. 380-385.

[2] A. Kulmala, E. Salminen, O. Lehtoranta and T.D. Hämäläinen, "Impact of Shared Instruction Memory on Performance of FPGA-based MP-SoC Video Encoder". Design and Diagnostics of Electronic Circuits and Systems, IEEE, 2006, pp. 57-62.

[3] Zhang, Wen-Ting et al.,"Design of Heterogeneous MPSoC on FPGA" Proceedings of the 7th International Conference on ASIC. ASICON'07. 2007, pp. 102-105

[4] K. Ravindran et al., "An FPGA-based soft multiprocessor system for IPv4 packet forwarding". Proceedings of the International Conference on Field Programmable Logic and Applications, 2005, pp. 487-492

[5] A. Hung, "Cache Coherency for Symmetric Multiprocessor Systems on Programmable Chips". Master Thesis in Electrical and Computer Engineering .University of Waterloo. 2004.

[6] A. Hung, W. Bishop and A. Kennings, "Symmetric Multiprocessing on Programmable Chips Made Easy", Proceedings of Design, Automation and Test in Europe Conference DATE'05. 2005, pp. 240-245

[7] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi and D. Sciuto, "A design kit for a fully working shared memory multiprocessor on FPGA", Proceedings of the 17th ACM Great Lakes symposium on VLSI, 2005, pp. 219-222

[8] P. Huerta, J. Castillo, J.I Martinez and C. Pedraza, "Exploring FPGA Capabilities for Building Symmetric Multiprocessor Systems", Proceedings of the 3rd Southern Conference on Programmable Logic, 2007, pp. 113-118

[9] Xilinx, "Xilkernel user guide", Document version December 2007