SoCrates[*] - A Multiprocessor SoC in 40 days

Mikael Collin, Raimo Haukilahti, Mladen Nikitovic, Joakim Adomat
{mci, rhi, mnc, jat}@mdh.se
Department of Computer Engineering, Mälardalen Real-Time Research Center (MRTC)
Mälardalen University, P.O Box 883, S-721 23 Västerås, Sweden

Abstract

The design time of System-on-a-Chip (SoC) is today rapidly increasing due to high complexity and lack of efficient tools for development and verification. This article describes the design and implementation of a Multiprocessor SoC *(MSoC) conducted by three master students. We propose a generic platform generator as a way to reduce time-to-market and verification time. With the project, we have shown that it is possible in a short time to develop a MSoC that fits on a single FPGA.*

1. Introduction

This paper describes the design of the first prototype of Socrates, a generic scalable platform generator that creates a synthesizable HDL description of a multiprocessor system. The platform is a result of a master thesis by three students. The goal was to build a predictable multiprocessor system with mechanisms for data prefetching on a single FPGA. This means that all development has to be done in a very short time and that the software and hardware must fit on a single FPGA. All the components except the *Real Time Unit* (RTU)[3] and the I/O node was implemented during the thesis. Linker scripts was implemented to configure the local memory of the processing nodes. Also, the RTU software interface had to be adapted to the underlying architecture.

2. Motivation

Design time including verification, has become one of the largest challenges in SoC design. The productivity gap is an example of the problem with developing complex SoC. To reduce this gap new design methods are needed. In order to meet the demands of fast verification and time-to-market, the system needs to be designed at a higher abstraction level. This can be obtained by using a platform generator, where the individual components are already verified. This means that the platform can instantly be tested and verified at system-level, which reduces the overall development time. To decrease design time we propose a parameterized system generator. SoCrates can easily scale to a number (1-6) of processing nodes and adapt the remaining components to given parameters at compile time.

3. System Overview

Socrates is a *distributed shared memory* (DSM) multiprocessor, with non-uniform memory access time. The architecture is based on a shared bus[1] on to which *nodes* are connected (figure 1). A typical node contains local memory, a network interface, and a CPU or DSP, where the CPU is an ARM7[6] clone. There exists a DSP node, based on the *CMUDSP* from Carnegie-Mellon University[4]. Due to area limitations, no DSPs are included in this the first version. Software applications are divided into threads and distributed onto the processors. Scheduling of threads is managed and controlled by a *Real-Time Unit* (RTU)[3]. Interprocess communication is performed through shared variables, whereas communication between hardware devices are made via memory mapped registers. In order to compensate for the non-uniform memory latency, the system supports a prefetch functionality. Remotely located data words are possible to fetch in advance before the point in execution where data is required. Prefetching of data words is software based[2], since prefetch instructions are inserted in the program code by the compiler or the programmer. The prefetch instructions are executed by the processor which issue a non-blocking read of a remote data word specified by the instruction. I/O is managed by a centralized node,

[*]SoCrates stands for SoC for real-time systems.

[1]Not the most attractive solution but commonly used for small multiprocessor systems[1].

which is responsible for code down-load and external communication.

3.1. Hardware Description

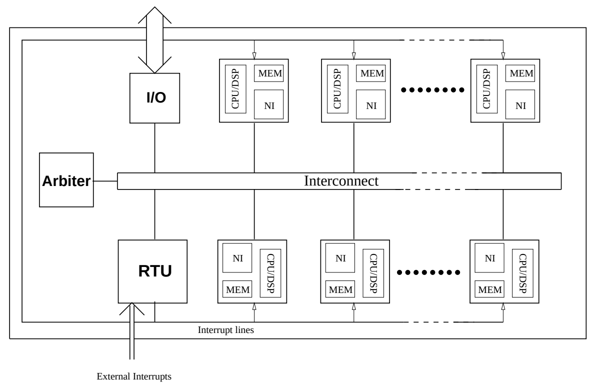

Figure 1. The architecture of the SoCrates hardware platform

The choice of an ARM architecture is motivated by its popularity and wide industrial use. Since no university license for the processor source could be obtained and the costs for ordering the processor as a hard IP was too high, the remaining option was to implement an ARM7 clone. For complexity reasons the clone only supports *ARM* state, and system/user and supervisor mode. The cache and pipeline are removed to further simplify the design, minimize area, and to increase predictability. Hence, the processor is fully predictable at the instruction level although the performance is decreased. The processor executes a subset of the ARM instruction set where multiplication, co-processor instructions, and the state changing instruction are removed. In addition, the instruction set is augmented with a prefetch instruction. The memory on each node is a dual-ported, zero wait-state Xilinx block RAM. Memory accesses are made transparent through a wrapper module, which makes it easier to change between different target technologies. Both internal and external communication is managed by a *Network Interface* (NI). The NI encapsulates the complexity of communication and address translation, acting as a simple MMU. The eight most significant bits of the 32-bit system addresses is a one-hot coded node identification field, the remaining 24 bits constitutes the local address. All nodes have their NIs connected to the shared bus mastered by a round robin arbiter. The system supports *unicast*, *multicast*, and *broadcast* transfers. For atomic transfers *locked* accesses can be utilized.

3.2. Real-Time Kernel

The hardware implemented real-time kernel performs thread scheduling and synchronization. RTU communication with the nodes is performed by a memory mapped register-based handshake scheme. Interrupts are sent out upon context switch and the nodes fetches their next thread ID. The kernel is parameterized with respect to the maximum number of threads, priorities and CPUs.

3.3. Software Development Flow

To make software development easier there is a need for a well-defined flow, from application writing to down-load of linked object code. Each nodes code, stack, and data must be mapped to their local address spaces. Therefore, each node is compiled and linked separately before final concatenation as shown in figure 2. A software interface is supported for the RTU and for I/O handling. The SW flow is based on GNU tools[5].

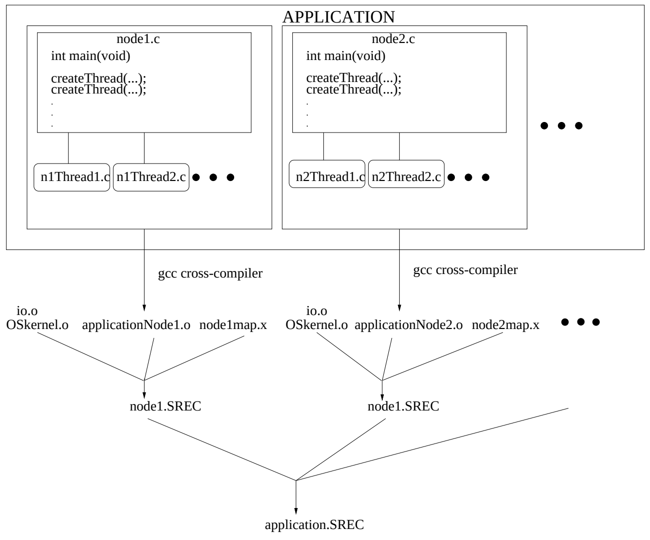

Figure 2. Flow from source to down-loadable object file.

3.4. Tools and FPGA-board

The platform for the physical implementation is a PCB containing a XILINX VIRTEX 1000 FPGA[7] (Figure 3). The FPGA has 16 Kb of block RAM that is shared among the processor nodes. A standard PC parallel port is used for code down-load and text terminals are used for application communication. In simulation, all physical I/O is modeled by C-code add-ons to Modelsim[8].

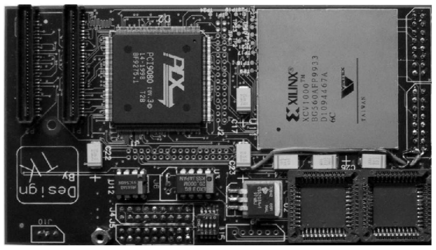

Figure 3. A photo of the FPGA board.

4. Problems

Despite fast workstations, time for simulation and verification is clearly a problem. Run-times for several days were not unusual, especially for back-annotated data. System simulation on pre place-and-route timing simulation proved to be useful as Place and Route took several days for the final implementation.

5. Current Results

A demo-application that runs on two CPU-nodes and utilizes the RTU and I/O node has been successfully implemented. The whole system including hardware[2] and software fits on a single FPGA. The developed linker scripts maps the user software to local address spaces and the system is parameterized by using generics. Figure 4 shows synthesis results of each hardware component.

Component	# gates	% of FPGA
System	653 349	58
CPU node	47 897	21
RTU	33 834	15
CPU core	41 929	19
NI	5 873	2

Figure 4. Synthesis results of SoCrates components.

6. Future Work

Our scalability relies on a ”perfect” interconnect. Today, a shared bus is used, which must be improved in the future with point-to-point/crossbar options in the generator. Software targeting is an important area and was solved by letting the user partition the application at thread level. This can be done automatically with respect to thread communication (locality) and code/data size. Interprocess communication HW support will be integrated from existing research projects at MRTC. With automatic generation of synthesis scripts, a pushbutton design flow seems within reach. Virtual memory and dynamic memory allocation HW support will be added.

[2]The prototype system contains two ARM cores, one RTU and one I/O-unit.

7. Conclusions

This project shows that it is possible to develop (including a whole CPU core) a multiprocessor SoC that fits on a single FPGA in very short time. By working closely in a small group of 3-5 people, information as new ideas and bug-reports during verification, are propagated immediately. A platform concept gives rapid design time and keeps the designers focused on debugging the application and not the platform itself. An “all synthesizable” approach is far from feasible in most projects but it is a convenient way of getting rapid results. A system view is crucial when designing SoC. All parts interact and focusing on narrow HW or SW issues does not give a satisfying solution.

References

[1] David E.Culler, et al., *Parallel Computer Architecture, A Hardware/software approach*, Morgan Kaufmann Inc, San Francisco California, 1999, ISBN 1-55860-343-3.

[2] VanderWiel, S.P. & Lilja, D.J. *When Caches Aren’t Enough: Data Prefetching Techniques*, IEEE Computer, Volume: 30 Issue: 7, July 1997, pp: 23 - 30.

[3] L Lindh, et al., *From Single to Multiprocessor Real-Time Kernels in Hardware*, IEEE Real-Time Technology & Applications Symposium, Chicago, May 15 - 17, 1995

[4] Carnegie-Mellon Low Power Group
http://www.ece.cmu.edu/ lowpower/benchmarks.html

[5] The GNU project
www.gnu.org

[6] ARM Ltd.
www.arm.com

[7] Xilinx Inc.
www.xilinx.com

[8] Mentor Graphics Corporation
www.mentor.com/modelsim