Instruction Set Simulator for MPSoCs based on NoCs and MIPS Processors

Leandro Möller[1], André Rodrigues[1], Fernando Moraes[2], Leandro Soares Indrusiak[3], Manfred Glesner[1]

[1] Darmstadt University of Technology - Institute of Microelectronic Systems - Darmstadt, Germany
[2] Faculty of Informatics - Catholic University of Rio Grande do Sul - Porto Alegre, Brazil
[3] Department of Computer Science - University of York - York, United Kingdom
Email: moller@mes.tu-darmstadt.de

Abstract

Even though Multiprocessor System-on-Chip (MPSoC) is a hot topic for a decade, Instruction Set Simulators (ISSs) for it are still scarce. Data exchange among processors and synchronization directives are some of the most required characteristics that ISSs for MPSoCs should supply to really make use of the processing power provided by the parallel execution of processors. In this work a framework for instantiating ISSs compatible with the MIPS processor is presented. Communication among different ISS instances is implemented by message passing, which is actually performed by packets being exchanged over a NoC. The NoC, the ISS and the framework that controls the co-simulation between them are all implemented in Java. Both ISS and the framework are free open-source tools implemented by third parties and available on the internet.

1. Introduction

Multiprocessor systems have become a standard in the computer industry since the release of the Intel Pentium D in 2005 [1]. Since then, processor manufacturers have focused in multi-core architectures to raise the processing power, favoring a larger number of cores instead of trying to achieve higher clock speeds, avoiding also the complexity of superscalar pipelines. While executing several small applications in parallel have a significant improve in performance with actual multiprocessor systems, a unique complex application needs a careful development to use wisely this processing power. It is not simply to write the application code with multiple threads, but each thread has to be really executing in the same time as the other threads, instead of paused in a *wait* directive.

While communication infrastructures based on bus have been sufficient for multiprocessor systems so far, the increase of number of cores and data transfer associated will demand a more complex on-chip interconnection. For this purpose Networks-on-Chip (NoCs) have arisen as a scalable solution to future increase of number of cores. The use of a NoC represents no direct changes to the developer of the complex application, but it counts when the execution time of the complex application is being analyzed.

The design space exploration of the scenario presented in the previous paragraphs and the tools to aid the development of complex applications are the goal of this work. A MIPS-like processor was connected to the HERMES NoC and presented in [2]. In [2] the debug of complex applications are implemented based on print directives. The work presented here improves the debugability by providing an Instruction Set Simulator (ISS) for the MIPS processor while considering the communication time and traffic under simulation in the NoC.

The ISS used in this work is the MARS ISS, developed by the Missouri State University [3]. This ISS was connected the RENATO NoC model [4], which is an actor-oriented model based on the HERMES NoC. The simulation environment used to control both the simulation of the NoC and the ISS is the Ptolemy II [5], developed by the EECS department at UC Berkley.

The rest of this work is divided as follows. Section 2 presents other ISSs targeting MPSoC architectures. A background about the tools and basic information required to understand this work is presented in Section 3. Section 4 presents how the communication among ISSs takes place. Section 5 presents timing delays of the system and Section 6 concludes this work.

2. Related Works

In this section different MPSoCs that have tools for debugging their embedded software are presented. Table 1 summarizes the most important information of these works and adds the work proposed in this paper. As presented in Table 1, all works use SystemC as simulation engine and memory mapped techniques to communicate with other processors, except the work proposed on this paper that uses the Ptolemy II simulation engine and the message passing technique to communicate with other processors.

MPARM [6] uses ARM processors connected through AMBA bus to compose the MPSoC. Multiprocessor applications are debugged with the SWARM ISS, which is developed in C++ and was wrapped to communicate with the MPSoC simulated in SystemC. The platform allows booting multiple parallel uClinux kernels on independent processors.

STARSoC [7] uses OpenRisc1200 processors connected through Wishbone bus. Debugging is implemented with the

OR1Ksim ISS, which is implemented in C language. The OR1Ksim also allows to be remote operated using GDB. Operating System is not yet supported.

HVP [8] supports several processors and therefore several ISSs. The work presented MPSoCs that contain ARM9 processors using ARM's ISS and in-house VLIW and RISC processors debugged by the LisaTek ISS. The ARM processors execute a lightweight operating system (name was not disclosed). The communication among processors was reported to be AMBA among ARM processors and SimpleBus among the in-house processors used.

SoClib [9] is a project developed jointly by 11 laboratories and 6 industrial companies. It contains simulation models for processor cores, interconnect and bus controllers, embedded and external memory controllers, or peripheral and I/O controllers. The MPSoC accepts the following processor cores: MIPS-32, PowerPC-405, Sparc-V8, Microblaze, Nios-II, ST-231, ARM-7tdmi and ARM-966. The GDB client/server protocol has been implemented to interface with these processors. The following operating systems are supported: DNA/OS, MutekH, NetBSD, eCos and RTEMS. Several bus and NoCs with different topologies wrapped with the VCI communication standard were ported and presented at www.soclib.fr.

The proposed work is based on a MIPS-like processor, implemented in hardware by the Plasma processor available for free at Opencores [10] and implemented by MARS [3] when simulating the processor as an ISS. All previous works used SystemC as simulation environment; this work uses Ptolemy II [5]. This work also differs from the others because it exchanges data between processors by using the native protocol of the NoC, therefore no extra translation is needed before sending and receiving packets.

Table 1 – MPSoCs that have tools for debugging embedded software.

Work ID	Simulation engine	Processor	Communication	Data exchange	ISS	OS
MPARM	SystemC	ARM	Bus (Amba)	Memory	SWARM	uClinux
STARSoC	SystemC	OpenRisc 1200	Bus (Wishbone)	Memory	OR1Ksim	No
HVP	SystemC	Several	Bus (several)	Memory	Several	Yes
SoClib	SystemC	Several	Bus / NoC (several)	Memory	GDB	several
Proposed	Ptolemy II	Plasma (MIPS)	NoC (Hermes)	Message	MARS	No

3. Background

This session reviews the required infrastructure to build our MPSoC simulation environment.

3.1. Ptolemy II

Ptolemy II [5] is a framework developed by the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley and it is implemented in Java. The key concept behind Ptolemy II is the use of well-defined models of computation to manage the interactions between various actors and components. In this work only the Discrete Event (DE) model of computation was used, but others are available on Ptolemy II.

In DE, the communication between actors is modeled as tokens being sent across connections. The sent token and its timestamp constitute an event. When an actor receives an event, it is activated and a reaction might occur, which may change the internal state of the actor and / or generate new events, which might in its turn generate other reactions. The events are processed chronologically [5].

3.2. MARS ISS

MARS [3] is a MIPS Instruction Set Simulator (ISS). This means that MARS simulates the execution of programs written in the MIPS assembly language. MARS can be executed by command line or Graphical User Interface. MARS was developed by Peter Sanderson and Kenneth Vollmar, from the Missouri State University, and is written entirely in Java and distributed in an executable Jar file. MARS can simulate 155 basic instructions from the MIPS-32 instruction set, as well as about 370 pseudo-instructions or instruction variations, 17 syscall functions for console and file I/O and 21 syscalls for other uses.

3.3. RENATO NoC

RENATO NoC [4] was developed using the Ptolemy II framework and its behavior and timing constraints are based on the HERMES NoC. The basic element of the NoC is a five bi-directional port router, which is connected to 4 other neighbor routers and to a local IP core, following a MESH topology. The router employs a XY routing algorithm, round-robin arbitration algorithm and input buffers at each input port.

The RENATO NoC model can be connected to a debugging tool called NoCScope [11]. NoCScope provides improved observability of RENATO routers and overall resources in use. Seven scopes are currently available, allowing the user to see information about hot spots, power consumption, buffer occupation, input traffic, output traffic, end-to-end and point-to-point communications.

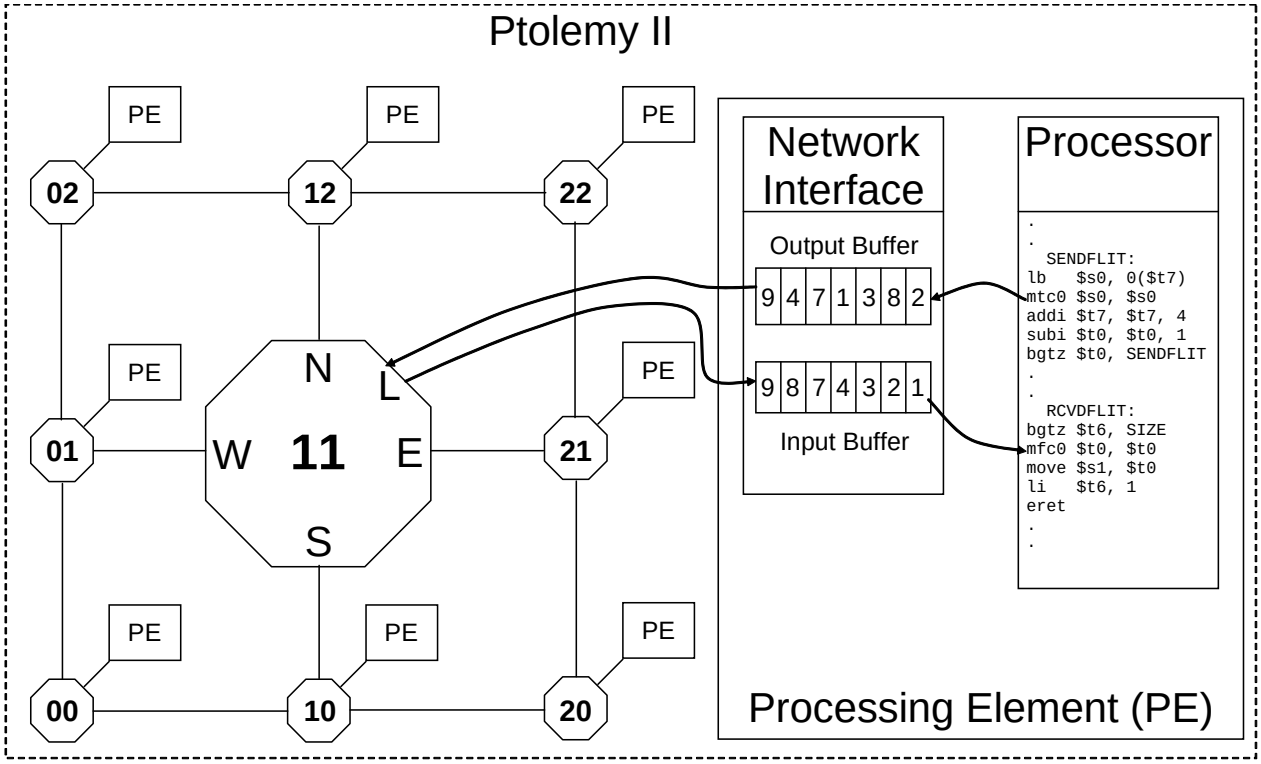

Figure 1 – Block diagram of the proposed multiprocessor ISS.

4. Communication among processors

This section presents how the MARS ISS was connected to the RENATO NoC to allow the creation of a multiprocessor ISS. Figure 1 shows a block diagram of the system that will be used in the next subsections to guide the explanation of each component.

4.1. Processor to NI

In the current version of this work, each processor executes the MIPS assembly code of one task of the application. Communication between tasks happens by exchanging packets. In order to send a packet to another task, the header of the packet and the packet data need to be first stored in the data memory of the processor. The header of the packet is composed by the address of the target router where the processor is connected and the number of data flits this packet contains. After that, the send packet subroutine is called.

The send packet subroutine first reads the size flit of the packet stored in the memory to a register and reads to another register the output buffer size available in the NI. If there is enough space available in the NI to store the packet, the subroutine proceeds sending the packet flit by flit to the NI. The process of "reading" a flit from the NI uses the instruction "move from coprocessor 0" (*mfc0*), while the process of "sending" a flit to the NI uses the instruction "move to coprocessor 0" (*mtc0*). Thus, from the point of view of the processor, coprocessor 0 is now the NI.

4.2. NI to NoC

With the packet stored in the NI output buffer, the NI sends the packet flit by flit to the input local port of the router where this NI is connected. This happens following the flow control protocol in use by the NoC and using the timing delays set on the NoC model being executed by Ptolemy.

4.3. NoC to NI

When packets are being received from the NoC into the NI, a different buffer (input buffer) is used, thus allowing parallel sending and receiving of packets. The receiving of packets also occur following the flow control in use by the NoC and using the timing delays set on the NoC model.

4.4. NI to processor

As soon as the flits of the packet arrive in the input buffer of the NI, the NI launches a specific interruption to the processor meaning that a new packet has arrived. The MARS ISS, which was executing its task, saves its context and receives the interruption in the form of a Java exception. The standard routine for handling exceptions is called. By the ID of the specific exception, the exact exception is found out to be the "new message from network exception". The specific subroutine of this exception is launched. This subroutine mainly reads the

complete packet from the NI using the "move from coprocessor 0" (*mfc0*) instruction to read each flit of the packet. After the complete packet was read from the NI and stored in the processor's memory, the processor's context is restored and it can now continues with its execution possibly using the data that was received.

5. Synchronization

The straightforward solution in Java to connect more than one MARS ISS to the NoC is to create a new MARS instance object for every new MARS instantiated in the NoC. However, this alternative failed due to the fact that MARS has been programmed using several static classes, attributes and methods. All of its main resources, such as the memory and the register bank, are declared as static. Therefore, if one tries to run more than one instance of MARS concurrently inside a single Java Virtual Machine (JVM), all the running instances will share the same resources, which will lead to unexpected behavior.

One possible workaround for this problem is to run each MARS instance in a different JVM. Java does not directly share memory between multiple VMs, so by running each MARS in a different JVM, one is safely isolating each instance of MARS. One problem with this approach is that the exchange of messages between different JVMs is only possible by using APIs such as Java Remote Method Invocation (RMI) and sockets, which would greatly increase the complexity of the system.

Another solution would be to reprogram MARS to remove the problematic static attributes and make them unique for each instance. However, this solution was also not optimal, considering the large number of static members declared in MARS and that every new future version of MARS would also require these modifications.

A better solution is to instantiate isolated ClassLoaders, one for each instance of MARS to be loaded. This works because a static element in Java is unique only in the context of a ClassLoader, therefore the static elements will not interfere with the other instances of MARS called by other ClassLoaders. By using this approach, the task of exchanging messages between the MARS instance and its corresponding NI also becomes trivial, and can be done simply by injecting a NI object when instantiating MARS.

A side effect of this solution is that each MARS instance and the NoC are considered as different threads by Java, and this would require extra algorithms based on *wait* and *notify* directives to maintain the time constraints followed by the NoC. As the main goal of this work is not provide good latency figures to the multiprocessor system application under simulation, we proceeded without the extra algorithms, aiming a faster simulation. Figure 2 presents a printout of the most important events occurred during the transfer of a packet composed by 2 header flits and 10 payload flits from MARS #1 to MARS #2. MARS #1 is connected to router 00 as illustrated in Figure 1 and MARS #2 is connected to router 21. No extra traffic is currently occupying the NoC.

```
3002 MARS #1 sending target  flit   (21) to NI #1
3002 MARS #1 sending size    flit   (10) to NI #1
3002 MARS #1 sending payload flit #0 (9) to NI #1
3003 MARS #1 sending payload flit #1 (9) to NI #1
3003 MARS #1 sending payload flit #2 (4) to NI #1
3003 MARS #1 sending payload flit #3 (7) to NI #1
3003 MARS #1 sending payload flit #4 (1) to NI #1
3003 MARS #1 sending payload flit #5 (3) to NI #1
3003 MARS #1 sending payload flit #6 (8) to NI #1
3003 MARS #1 sending payload flit #7 (2) to NI #1
3003 MARS #1 sending payload flit #8 (6) to NI #1
3086 MARS #1 sending payload flit #9 (5) to NI #1
3087 NI #1   sending target  flit   (21) to NoC
3089 NI #1   sending size    flit   (10) to NoC
3091 NI #1   sending payload flit #0 (9) to NoC
3093 NI #1   sending payload flit #1 (9) to NoC
3095 NI #1   sending payload flit #2 (4) to NoC
3097 NI #1   sending payload flit #3 (7) to NoC
3099 NI #1   sending payload flit #4 (1) to NoC
3101 NI #1   sending payload flit #5 (3) to NoC
3103 NI #1   sending payload flit #6 (8) to NoC
3105 NI #1   sending payload flit #7 (2) to NoC
3107 NI #1   sending payload flit #8 (6) to NoC
3109 NI #1   sending payload flit #9 (5) to NoC
3112 NoC     sending target  flit   (21) to NI #2
3116 NoC     sending size    flit   (10) to NI #2
3120 NoC     sending payload flit #0 (9) to NI #2
3120 NI #2   sending payload flit #0 (9) to MARS #2
3124 Noc     sending payload flit #1 (9) to NI #2
3128 Noc     sending payload flit #2 (4) to NI #2
3132 Noc     sending payload flit #3 (7) to NI #2
3136 Noc     sending payload flit #4 (1) to NI #2
3140 Noc     sending payload flit #5 (3) to NI #2
3144 Noc     sending payload flit #6 (8) to NI #2
3148 Noc     sending payload flit #7 (2) to NI #2
3152 Noc     sending payload flit #8 (6) to NI #2
3156 Noc     sending payload flit #9 (5) to NI #2
3166 NI #2   sending payload flit #1 (9) to MARS #2
3170 NI #2   sending payload flit #2 (4) to MARS #2
3172 NI #2   sending payload flit #3 (7) to MARS #2
3174 NI #2   sending payload flit #4 (1) to MARS #2
3175 NI #2   sending payload flit #5 (3) to MARS #2
3177 NI #2   sending payload flit #6 (8) to MARS #2
3178 NI #2   sending payload flit #7 (2) to MARS #2
3180 NI #2   sending payload flit #8 (6) to MARS #2
3181 NI #2   sending payload flit #9 (5) to MARS #2
```

Figure 2 – Timing delays of the most important events during the transfer of a packet between two processors.

All the following comments presented in this paragraph refer to Figure 2. Between times 3002 and 3086 MARS #1 sends the packet to the NI connected to it (NI #1), exactly as explained in Section 4.1. Eleven of the twelve flits of the packet were sent in the first 2 simulation cycles, and the last flit of the packet at time 3086. This strange behavior implies the following results: (1) MARS #1 thread was executed two times concurrently to Ptolemy thread, between times 3002-3003 and 3086; (2) MARS thread can be faster enough to execute at least 11 *mtc0* instructions in a row during 2 simulation cycles of Ptolemy; (3) MARS thread was not called again during 83 simulation cycles (3086-3003). Between times 3087 and 3109 each flit of the packet was sent constantly every 2 simulation cycles from NI #1 to the NoC, exactly as explained in Section 4.2. This behavior is equal to the real HERMES NoC that needs 2 clock cycles to transfer a flit using handshake flow control. Between time 3112 and 3156 all the flits from the packet were delivered from the NoC to NI #2 as explained in Section 4.3. However, due to some technical difficulties in the current version, it was not possible to deliver each flit every 2 simulation cycles, but 4 simulation cycles in this

case. At time 3120 it is possible to see that NI #2 delivered the first payload flit immediately to MARS #2. Between times 3166 and 3181 the rest of the payload flits were delivered to MARS #2 as described in Section 4.4. Here again it is possible to see that the data transfer did not follow a constant pattern, similar to one the occurred between times 3002 and 3086. This unpredictable behavior is a side effect of running multiple threads with no proper synchronization.

6. Conclusion and Future Work

This work presented an ISS for multiprocessor systems based on the MIPS processor. In this work the RENATO NoC model was connected to two instances of the MARS ISS and as result applications based on more then one processor can be easily debugged with the presented approach. The most important contribution of this work is the NI, which allows both systems to communicate, thus creating a more realistic multiprocessing system model composed by computation and communication.

Initial figures regarding latency between processors' communication through the NoC were measured and we report to be insufficient in the current version. In order to have a good latency figure we must: (1) back annotate the timing delays of each assembly instruction from a real MIPS processor to MARS; (2) add extra synchronization logic to mimic the timing delays between processor and NI. In the current version of this work we guarantee only the NoC timing delays as presented in [4]. Future works will be related to steps 1 and 2.

References

[1] Intel Corporation. Intel Pentium D (Smithfield) Processor. Available at: http://ark.intel.com/ProductCollection.aspx?codeName=5788.

[2] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. "HeMPS - A Framework for NoC-Based MPSoC Generation". In: ISCAS'09, 2009, pp. 1345-1348.

[3] Vollmar, D. and Sanderson, D. "A MIPS assembly language simulator designed for education". Journal of Computing Sciences in Colleges, vol. 21(1), Oct. 2005, pp. 95-101.

[4] Indrusiak, L.S.; Ost, L.; Möller, L.; Moraes, F.; Glesner, M. Applying UML Interactions and Actor-Oriented Simulation to the Design Space Exploration of Network-on-Chip Interconnects. In: ISVLSI'08, 2008, pp. 491-494.

[5] Eker, J.; Janneck, J.; Lee, E.; Liu, J.; Liu, X.; Ludvig, J.; Neuendorffer, S.; Sachs, S.; Xiong, Y. "Taming Heterogeneity - The Ptolemy Approach". Proceedings of the IEEE, vol. 91 (2), Jan. 2003, pp. 127-144.

[6] Benini, L.; Bertozzi, D.; Bogliolo, A.; Menichelli, F.; Olivieri, M. "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC". The Journal of VLSI Signal Processing, vol. 41 (2), Sep. 2005, pp. 169-182.

[7] Boukhechem, S.; Bourennane, E. "SystemC Transaction-Level Modeling of an MPSoC Platform Based on an Open Source ISS by Using Interprocess Communication". International Journal of Reconfigurable Computing, vol. 2008, Article ID 902653, 2008, 10 p.

[8] Ceng, J.; Sheng, W.; Castrillon, J.; Stulova, A.; Leupers, R.; Ascheid, G.; Meyr, H. "A high-level virtual platform for early MPSoC software development". In: CODES+ISSS'09, 2009, pp. 11-20.

[9] Pouillon, N.; Becoulet, A.; Mello, A.; Pecheux, F.; Greiner, A. "A Generic Instruction Set Simulator API for Timed and Untimed Simulation and Debug of MP2-SoCs". In: RSP'09, 2009, pp. 116-122.

[10] Opencores. Available at: http://www.opencores.org.

[11] Möller, L.; Indrusiak, L.S.; Glesner, M. "NoCScope: A Graphical Interface to Improve Networks-on-Chip Monitoring and Design Space Exploration". In: IDT'09, 2009.