COMMUNICATION SCHEME FOR AN ADVANCED JAVA CO-PROCESSOR

Tero Säntti and Juha Plosila

{teansa | juplos}@utu.fi
phone: +358 2 333 6956 fax: +358 2 333 6950
Communication Systems Laboratory
Department of Information Technology
University of Turku Lemminkäisenkatu 14-18 20520 Turku Finland

ABSTRACT

This paper describes interface strategies for a Java co-processor (from now on JPU). The interface units are interchangeable, and share a common communication scheme towards the co-processor. The first version of the interface is designed for single CPU and single co-processor environment. The other is for a network of multiple CPUs and co-processors. The co-processor does not need to know what kind of environment is is placed in, as all communication goes through the interface unit. This modularity of the design makes the co-processor more reusable and allows system level scalability. This work is a part of a project focusing on design of an advanced Java co-processor for Java intensive SoC applications.

1. INTRODUCTION

Java is very popular and portable, as it is a write-once run-any-where language. This enables coders to develop portable software for any platform. Java code is first compiled into bytecode, which is then run on a Java Virtual Machine (hereafter JVM). The JVM acts as an interpreter from bytecode to native microcode, or more recently uses just in time compilation (JIT) to affect the same result a bit faster at the cost of memory. This software only approach is quite inefficient in terms of power consumption and execution time. These problems rise from the fact that executing one Java instruction requires several native instructions. Another source for inefficiency is the cache usage. As the JVM is the only part of software running natively, it occupies the instruction cache, whereas the Java bytecode is treated as data for the JVM, hence being located in the data cache. Also the actual data processed by the Java code is assigned to the data cache. This clearly causes more memory accesses missing the cache. When the execution of the bytecode is performed on a hardware co-processor this is avoided and the overall amount of memory accesses is reduced.

This work is a part of the REALJava [1] project, which aims to design a Java co-processor that is easily implemented to various systems. We have chosen to use asynchronous techniques in this project because then we can achieve good performance with reasonable power consumption and vary easy integration with existing systems, as no clock limitations need to be considered. Asynchronous self-timed circuit technology [4], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform for obtaining a highly modular low-power and low-noise Java accelerator implementation.

Overview of the paper We proceed as follows. In Section 2 we shortly describe the structure of any JVM, and show how the proposed co-processor fits into the specifications. Section 3 describes the requirements for the interface. Sections 4 and 5 go in to more detail for the interfaces in small and large systems respectively. Finally in Section 6 we draw some conclusions and describe the future efforts related to the REALJava co-processor.

2. GENERIC JAVA VIRTUAL MACHINE STRUCTURE

In the Java Virtual Machine Specification [3], Second Edition the structure and behavior of all JVM's is specified at a quite abstract level. This specification can be met using several techniques. Usual solutions are software only, including some performance enhancing features, such as JIT (Just In Time Compilation). We have chosen to use a HW/SW combination [1] in order to maximize the hardware support and minimize the power consumption. The HW portion (highlighted in Figure 1) handles most of the actual Java bytecode execution, whereas the SW portion takes care of memory management, class loading and native method calling. This partitioning gives the possibility to use the co-processor with any type of host CPU(s) and operating

systems, as all of the platform dependent properties are implemented in software and (most of) the common bytecode execution is done in hardware.

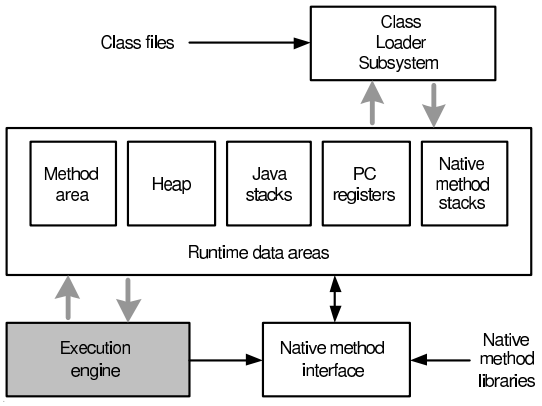

Figure 1: Internal architecture of the JVM

Because Java supports multithreading at language level, it makes sense to integrate several co-processors as a SoC. This gives an ideal solution for complex systems running several Java threads and possibly some native code at the same time. The system architecture can be chosen to be a network of any kind or bus based, as suitable for other components in the system.

3. REQUIREMENTS FOR THE INTERFACE AND PROTOCOL

To control the execution flow of the JPU there must be a control channel from the CPU to the JPU providing methods for halting and continuing the execution of the Java thread. Also the JPU must be able to send an interrupt request (IRQ) to the CPU, to notify the CPU of an unhandled instruction, need for more memory to be allocated or the end of the current thread. Naturally the CPU needs also data connection to the JPU, to set several parameters such as memory offsets, reserved range and so forth. The JPU is also given a complete memory channel, to access both program code and the data required.

The co-processor has own built-in address generation block and accesses the memory directly without CPU intervention. The interface must also be reasonably fast, as all communication goes through it. In the multiple JPU case the interface must also handle some protocol issues not present in the single JPU setting. The protocol developed for this purpose is presented in Section 5.

In both cases the JPU has no direct access to the CPU, but has an interrupt request line. Using this interrupt the JPU notifies the CPU of unhandled instructions or other occasions requiring CPU intervention. Upon receiving an IRQ the CPU first reads the contents of a status register in the JPU to determine the cause of the interrupt. Then the CPU performs whatever tasks necessary, and orders the JPU to continue execution. In a multiple JPU environment the IRQ is managed by sending a datagram to the CPU. A system with wired IRQ lines is not practical since any given JPU may be allocated to different CPUs at times.

4. SINGLE CPU SINGLE JPU

In a single CPU single JPU environment the JPU is placed on the memory bus. Figure 2 shows a conceptual view of the placement. This approach makes it easy to integrate the JPU to (virtually) any existing system. The JPU may be included as a separate ASIC or integrated into the same chip with the CPU. Clearly the single chip solution provides several benefits, such as lower power consumption for all CPU-JPU communication and also for CPU-MEM communication and reduced area on the circuit board. The separate ASIC approach gives more flexibility to incorporate the JPU to systems with 3rd party processors.

Figure 2: The JPU connected to the CPU and Memory

The control of the bus-ownership is assigned to the CPU. The CPU sets the CTRL-signals to direct the data flow as required. The chosen strategy gives good performance, especially for consecutive data transfers between given end points (CPU-MEM,CPU-JAVA or JAVA-MEM). Also the number of extra control pins required between the CPU and the co-processor is kept at minimum. This solution is shown in detail in Figure 3.

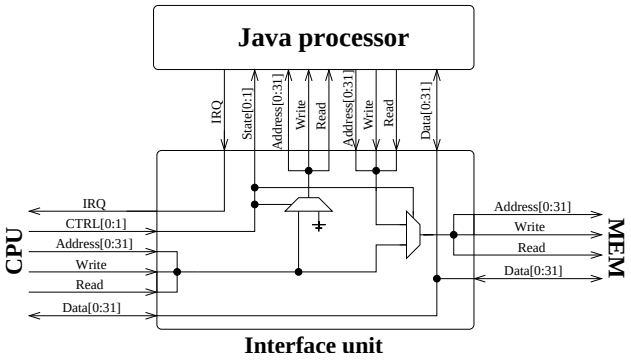

Figure 3: Interface as a separate component, connected to the Memory Bus. The data bus just connects all three endpoints together, and all units have tri-state drivers.

The CTRL signals control both the interface and the JPU. This information is used by the co-processor to decide when the registers contain valid data and the bus is reserved for the co-processor. Then the co-processor can execute the thread currently indicated by the registers. This approach gives the simplest structure for all components, and requires a minimal set of external communication pins from the CPU to the co-processor. The CPU must remain silent on the data bus, when the co-processor is executing, and vice versa. If the CPU is limited in I/O, the CTRL-signals can be obtained from the highest bits of the address, but this masks a portion of the memory from the CPU. Also the address bus of the CPU must maintain its state during JPU execution. The CTRL codes are shown in Table 1.

CTRL	Bus control	Destination	Notes
00	CPU	MEM	State at reset and native programs
01	CPU	JPU	Setup for JPU
10	JPU	MEM	JPU executing
11	JPU	MEM	JPU halts after finishing current operation

Table 1: Function of CTRL-signals.

For the single JPU case no special protocol is needed. The only requirement is that the CPU must be silent on the memory bus during JPU execution.

5. JPU IN A SOC / NOC ENVIRONMENT

As stated before in Section 2, It seems very beneficial to include more than one JPU in a large system. This approach brings forth true multithreading and thus improves performance. Also large systems possibly contain several software subsystems, such as internet protocols, user interface controllers and so on, these can easily be coded in Java, and since they all are executed in parallel the user experience is enhanced. The multithreading also improves the predictability of the real time performance, as the threads do not get any wait states and the caches and stacks for each thread are kept intact inside their respective JPUs. Figure 4 gives an example of connecting multiple CPUs, JPUs and memories together using a traditional bus-structure.

As can be seen in Figure 5, the channel between the JPU and the interface unit is exactly the same as for the single JPU case. This means that the JPU does not know its environment, and does not require any modification depending on the higher level architecture. The interface unit needs only a slight modification depending on the actual bus structure chosen for the system. The interface is responsible for keeping track of who is using this particular JPU.

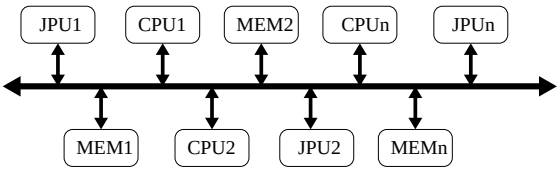

Figure 4: A large SoC with several CPUs, JPUs and memories using a bus structure.

This information is required to be able to allocate the JPU to some CPU, or tell an other CPU that this JPU is already in use. Also the IRQ has to be directed to the correct CPU. The system may have several memories as well, so the interface keeps a record of the memory containing this thread's memory areas. These additions give the JPU similar access to system resources as in the single JPU system.

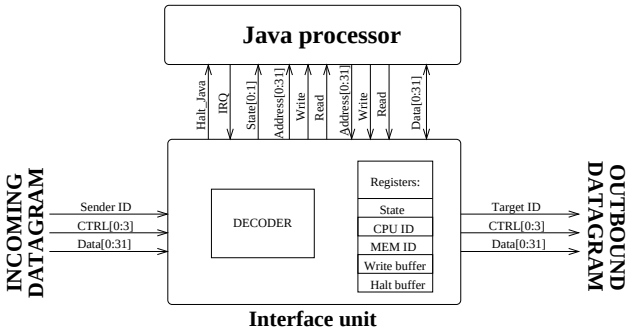

Figure 5: An example implementation for the pipelined bus.

The structure of the underlying network or bus is rather irrelevant, as long as the lower level provides two properties: 1) the datagrams must arrive in their destination in the same order that they were sent, and 2) the datagrams arriving from two different sources to a same destination must be identifiable. The first property can be be achieved with a lower level network protocol, like ATM adaptation layer (AAL) for internet, or by the physical structure of bus. The example we use here is a pipelined bus structure which guarantees the order of the datagrams by structure. The second property seems quite natural, and should be present in all solutions.

We chose to use the pipelined bus [2], since it provides a good platform for multiple processing units accessing the bus simultaneously. The bus provides high throughput at the expense of increased latency in comparison to a conventional bus. These properties rise from the structure of bus. Figure 6 shows the internal 3-level pipelines in each transfer stage. Our example system has a bus with 32-bit data word and 4 control bits per datagram as a payload. The bus itself contains more information about destination and the sender.

The sender's id is also passed on to the interface unit. A simple yet efficient protocol for this case is given in Table 2.

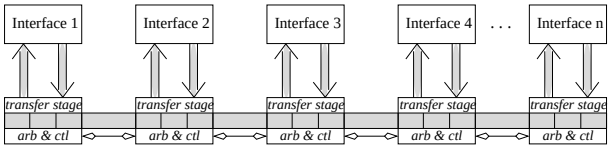

Figure 6: Detailed view of the pipelined bus with the interface units.

The coding and interface allow for the CPU to send several commands down the pipeline improving the overall performance. To prevent possible deadlocks, the CPU can send only one command together with a halt instruction, until the JPU sends a message stating that the JPU has completed all its actions, and has halted. The exception is a write command, which may contain both parts of the command before the JPU has halted. If this is not enforced at the CPU, a deadlock may result, if the JPU has pending memory reads and the CPU sends a burst of commands. The data from the memory may be blocked behind the instructions from the CPU, and they cannot move forward until the JPU has halted, which it cannot do before it receives the pending data from the memory. The high throughput also makes it reasonable to design a burst mode to the protocol. In this mode the interface unit counts the register addresses instead of receiving every address from the CPU. To enter the burst mode the CPU sends first a burst command which contains the number of consecutive addresses to be written or read. The the next datagram contains the starting address and either a read command or a write command.

6. CONCLUSIONS AND FUTURE WORK

Two strategies for connecting a Java co-processor were described in this paper. Both use identical co-processors without any modification. A simple yet efficient communication protocol for NoC environments was presented.

We plan to continue with designing the REALJava co-processor, and manufacturing it as a separate ASIC in the first stage. Later a larger NoC system with several CPUs and JPUs will be designed to implement a real-life application.

Code	Description	Notes
0000	status check	returns state information of the co-processor
0001	halt	halts JPU after current operations have finished and sends an acknowledgment
1000	continue	JPU continues execution
0010	read	JPU returns the contents of the register specified in the DATA part of the datagram
0011	halt&read	JPU halts and then reads
1010	read&continue	JPU performs read and then continues
1011	halt&read &continue	JPU halts, then performs read, then continues
0100	write	2 cycle operation, 1st cycle contains the register address and the 2nd contains the data to be written
0101	halt&write	JPU halts and the first cycle of write is completed concurrently
1100	write&continue	2nd cycle of write is completed and then JPU continues execution
X11X	burst length	DATA contains the length of the burst, the next datagram contains read or write and start address may be combined with halt and/or continue
1101	illegal	since write is 2 cycle command, it can not contain both halt and continue
1001	reserve	reserves the JPU for a CPU, if the JPU is available and DATA$\neq$0, if DATA=0 the JPU is released

Table 2: A protocol for the CPU to access the JPU.

7. REFERENCES

[1] Z. Liang, J. Plosila, and K. Sere. "Asynchronous Java Accelerator for Embedded Java Virtual Machine", *In Proc. of IEEE CAS Symposium on Emerging Technologies, Frontiers of Mobile and Wireless Communication*, Shanghai, China, June 2004.

[2] P. Liljeberg, J. Plosila, and J. Isoaho. "Self-Timed Communication Platform for Implementing High-Performance Systems-on-Chip",*VLSI Integration*, Elsevier Journal, to appear in 2004.

[3] T. Lindholm and F. Yellin. "The Java Virtual Machine Specification", Second Edition, Addison-Wesley, 1997.

[4] J. Sparso and S. Furber. "Principles of Asynchronous Circuit Design - A System Perspective", Kluwer Academic Publishers, 2001.