International Conference on Architecture of Computing Systems -Trends in Network and Pervasive Computing - Java in Embedded Systems - ARCS 2002, ISSN: 3-8007-2686-6, S. 135-140, Karlsruhe, April 2002

JSM: A small Java processor core for smart cards and embedded systems

Frank Golatowski, Hagen Ploog, Nico Bannow and Dirk Timmermann
University of Rostock, Department of Electrical Engineering and Information Technology
Institute of Applied Microelectronic and Computer Science, Germany
{Frank.Golatowski, Hagen.Ploog}@uni-rostock.de

Abstract

We describe the current work to extend our Java processor Java Silicon Machine (JSM) for embedded system applications. The JSM is a Javacard processor supporting all Javacard bytecodes. The JSM is a fully synthesizable 32bit processor soft core with a very small footprint. The processor requires 20 percent of a Xilinx Virtex 1000E FPGA area.

1 Introduction

Java processors execute Java bytecode instructions directly in hardware and overcome the disadvantage of long execution times in software based Java Virtual Machines. Corresponding to appropriate application fields and Java programming language standards various processor solutions have been developed in the past. One of the first Java processors fulfilling Sun's Javacard specification [15] was presented at [8]. In this paper we present some details of our JSM-architecture with an in depth description of its functionality. We also show the capability of its integration into small embedded or automation systems.
The paper is organized as follows: In section two, we briefly summarize the state-of-the-art in JAVA processors. In section three, we describe the architecture of the JSM. In section four, the structure of the JSM and of all modules is presented. Section five concludes with a short summary.

2 Recent Work

During the last years some work has been done in developing different kinds of Java processor cores (see **Table 1**). TINI^{2}J, SmartJ, JSMART and JSM are Javacard processors. Typically, Javacard solutions are based on a microcontroller with a small Java virtual machine. The disadvantage of those solutions is insufficient execution speed of java programs. A hardware implementation of Javacard virtual machine can boosts execution speed. Three of the Javacard processors are bilingual, which means that the processors support special microcontroller instructions and Java bytecode as well. The JSM presented supports all specified Java bytecodes, but also some newly defined microcodes. For development of Javacard applications the standard tools (Javacard Development Kit) may be used.

The TINI^{2}J is a bilingual 32bit RISC microprocessor that executes Java bytecode and native RISC instructions. The processor is an optimized core for Smart Cards and Javacard applications [1].
LavaCore is a configurable Java Processor FPGA core [4]. This core is targeted for Virtex-II-FPGAs and is part of the XILINX platform FPGA initiative and Empower™ processing solution.
The ARM926EJ-S is another synthesizable 32bit core. It is an integration of three machine instruction sets: namely the 32bit ARM instruction set, 16bit Thumb® instruction set and Java bytecode. One part of the Java bytecode is realized in hardware (140 instructions) and the remaining ones are executed by special software emulation [2].
Java Silicon Machine (JSM) [8] is a real 32bit Java processor core that executes Java bytecode corresponding to Javacard standard 2.1.2 [16]. JSM is a synthesizable 32bit soft core optimized for Javacard applications but can also be used in embedded systems [3]. The processor was evaluated on the Aptix MP3 rapid prototyping board [6].
Komodo [10], MOON [11], SmartJ [17] are further hardware realizations of Java standards. Nazomi has designed JSTAR and JSMART processors [12]. The first one is based on general Java standard and the latter one is for Javacard applications. Both are coprocessor solutions.

3 Design of the Javacard-Processor JSM

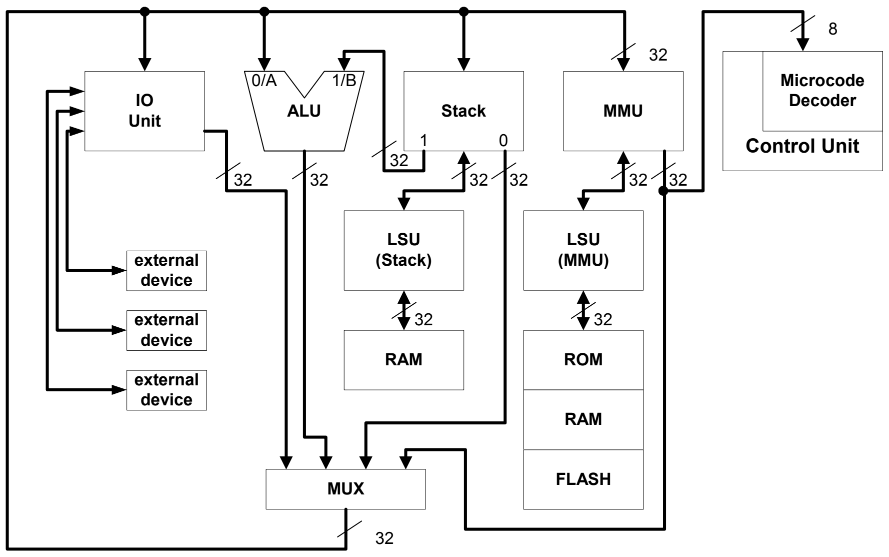
Java bytecodes are implemented on the base of microcode instructions using two cascaded microcode tables. Entries in the first table point, for each bytecode, to the specific entry in the second table containing the appropriate microcode sequences. During execution of microcode sequences direct jumps and subroutine calls are possible. These microcode instructions allow the implementation of any user-defined bytecodes.

Table 1: Different hardware solutions for Java and Java card virtual machines

	JSM [8]	SmartJ [17]	TINI^{2}J [1]	JSMART /JSTAR [12]	LavaCore [4]	Jazelle [2]	Komodo [10]	MOON [11]
Target	Softcore, Xilinx FPGA, ASIC	Integration with ST22 core	Integration with RISC processor, CPLD, ASIC	Softcore	Softcore, Xilinx FPGA	Integration with ARM	Softcore, Xilinx FPGA	Altera APEX
Special feature	Hardware security features, Integration of external devices, e.g. I^{2}C, TDES, Support of JVM stack model	Hardware security features, Combination with micro-controller	Hardware security features, Interrupt control interface	Co-processor	Optional extendable with SPU, DES, GC	Compatible with ARM, Combi-nation with ARM-controller.	Multithreaded Kernel. Interrupt Service Threads	Co-processor and Single processor solution, Support of JVM stack model
Support of additional codes	New bytecodes	Bilingual	Bilingual	Bilingual		Trilingual		
Standard	Javacard	Javacard	Javacard	Javacard/ Java	Java	Java	Java	Java

The processor is partitioned in some loosely coupled modules (finite state machines) to get a structured model, which may be easily modified and expanded. The JSM has a modular design and its processor architecture has been designed to fulfil the demands of different kinds of security applications. One of the most valuable features is that the core is extendable. Different types of external devices may be combined with the core using a simple interface. As a reference, the processor has been extended with a triple DES crypto processor and I^{2}C interface.

Fig. 1. Design of JSM- processor



4 Structure of the JSM processor

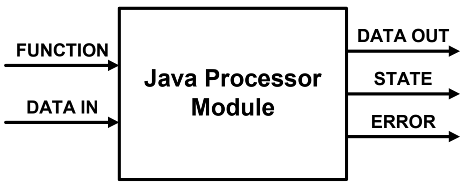
The whole JSM consists of one master FSM controlling several slave FSMs. For the ease of automatic implementation all FSMs are decoded as Moore-type machines.
The processor consists of the following main modules (**Fig. 1**), which will be explained in more detail in the following sections:

- Control-Unit (CU),
- Arithmetic Logical Unit (ALU),
- Memory Management Unit (MMU),
- Stack
- Input/Output-Unit (IO-Unit)
- Load and Store Unit (LSU)

4.1 FSM submodule

Fig. 2 depicts the general structure of a processor submodule. Each module receives input data and executes a selected function. The result of the executed function is available as data output. A state output signal monitors the internal state of the machine and an error signal shows the kind of errors (none, uncritical, critical and fatal). With the pattern applied to the function input the execution of the appropriate function is started. For each module an internal set of microcode instructions is defined. If the state is BUSY, the module has not finished its previous cycle and no new function is accepted. If the state-line signals READY, the FSM has confirmed the end of the cycle and the result vector is mapped to the output. The submodule stores the result vector on its output until a new function is selected on the modules input for execution.

Fig. 2. General structure of JSM module



4.2 Execution of Java bytecode

The execution of the Java bytecode instruction "iadd" should now be explained in more detail (**Fig. 3**). IADD adds TOS and TOS-1 and stores the result back on TOS. The JSM processor executes the IADD bytecode as follows:

`Microcode cycle 1:`

> The stack module receives POP_TOS_N_TO_OUTPUT. Thereby, the two topmost entries of the stack (TOS, TOS-1) are fed to the output and the stackpointer is decremented by two.

`Microcode cycle 2:`

> The output of the stack is connected to the input of the ALU. The ALU receives LOAD_AB, which stores the two input values in corresponding registers A and B inside the ALU.

`Microcode cycle 3:`

> The ALU receives ADD. This microcode adds the registers A and B and moves the result to the output.

`Microcode cycle 4:`

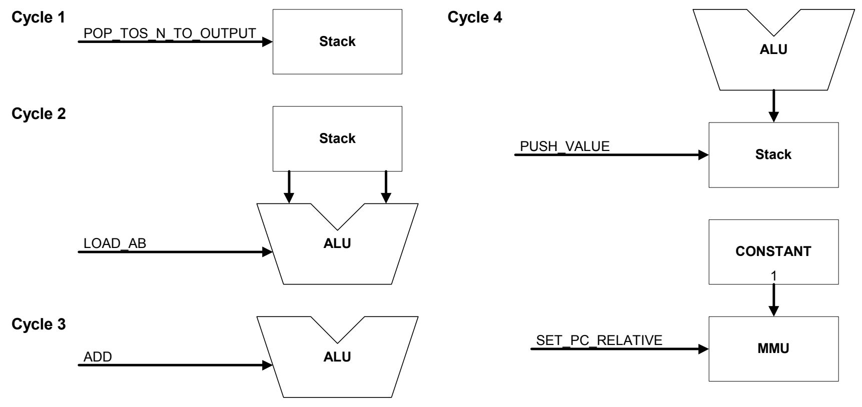
> The output of the ALU is connected with the input of the stack module. The stack receives PUSH_VALUE, which pushes the value available on the input pins to the top of stack and increments the stackpointer by one.
>
> The IADD opcode itself requires one byte. The following byte contains the next opcode. The MMU receives SET_PC_RELATIVE (constant "1"). Thereby, the PC is incremented by one; the byte the PC is pointing to is loaded and propagated to the output pins of the MMU.

For higher speed requirements and because of the independency of the submodules the MMU could receive the command for loading the next bytecode in any of the preceding microcode cycles.

4.3 Control-Unit

The control unit (CU) is responsible for bytecode execution and the controlling of all other units. The execution of bytecode is realized using a state-of-the-art microcode sequencer. By choosing a microcode sequencer the realization becomes more like a software program than a hardwired implementation. The CU executes the microcode stepwise. Therefore, the microcode is loaded from a microcode look-up-table and is directed to the special sub-module for one clock cycle whereby the execution automatically starts. For synchronization purposes the “no operation“ microcode instruction is put back on the bus afterwards. This means that the different modules have to wait for each other. Note that for each module (ALU, CU, MMU, STACK, IO-Unit) a set of different microcodes exists. For every Java bytecode at least one microcode will be executed. This is also valid for the NOP Java bytecode. This bytecode influences only the CU module where the NOP microcode is executed. The number of necessary microcodes depends on the complexity of the Java bytecode.

Fig. 3. Signals and Components of CU



4.4 Newly defined bytecodes

To manage applets by an operating system additional bytecodes are required, e.g. to prepare an applet for execution, to manipulate memory or to access external devices (communication). Memory manipulation includes software based garbage collection or deletion of objects. By using additional bytecodes the Java security concept could be compromised. Therefore, only API applets stored in ROM are allowed to use these bytecodes. For downloadable applets a multistage security system is implemented. The corresponding software secures that no downloadable applet contains any of the new bytecodes. During the download process the applet is verified by the on card download-manager to guarantee that an applet containing any of the new bytecodes will be removed from the memory before its execution starts. The third level consists of a runtime verification of the source of the bytecode to be executed. Only methods or applets stored inside the cards ROM are allowed to use the additional bytecodes. Unknown (or additional) bytecode stored in the card's applet memory forces the processor to run into a halt state. The following additional bytecodes are available (**Table 2**).

Table 2. Definition of new bytecodes

New bytecodes	Operation
start_applet	Preparation of an applet.
read_applet	Reading a word from memory and save it on stack.
Write_applet	Writes a word to memory. Value and reference are on stack.
Select_io_device	Selects a device from I/O modules.
read_io_device	Reads value(s) from I/O device and pushes it on top of stack.
Write_io_device	Outputs a value to I/O device. Value is on stack.

4.5 ALU

The arithmetic logical unit (ALU) is responsible for execution of the mathematical operations including negation, bit manipulation, remainder generation, rotation and logical operations, and test and compare operations. Two registers are buffering the input operands, which are signed 32bit values.

4.6 MMU

The memory management unit (MMU) is responsible for memory management of the JSM excluding stack. The stack manages the stack frames by itself. The MMU is able to address 2^{32} addresses. According to the different tasks various types of memory are directly supported. The ROM contains the operating system, constants, and non-moveable applets (card's API). RAM stores object data of those objects that have the transient tag set and applets that can be deleted after power down. The FLASH memory is used by

- moveable applets which can be deleted as well
- objects (array, class objects)
- object data of objects which have persistent tag enabled
- class variables

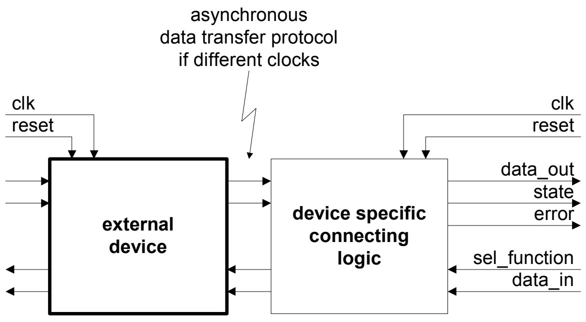
4.7 Stack

The stack is used for transferring parameters between functions or methods and it is also the place to save temporary data like variables of methods. Similar to other systems also internal data is stored on the STACK at method invocations (such as program counter), the calling method and the class of calling method. In the current implementation the stack may have $2^{10} = 1024$ entries, but could easily be expanded to meet any other requirement.

4.8 IO-Unit

The JSM was intended as a Javacard processor with the capability to connect external devices and for the use in embedded systems.
The IO-Unit serves as an interface between the processor and the corresponding IO-devices which are not defined in the Java standard or can be accessed only by using an appropriate application programmer's interface (API). Especially in a Javacard environment the IO-Unit is necessary for the communication between the card itself and the card reader. Furthermore, the IO-Unit may be used to control additional hardware to accelerate internal computations (e.g. encryption engine).
The interface for external device is depicted in **Fig. 4**.

Fig. 4. Interfacing external devices



4.9 Load and Store Unit

The LSU decouples Stack and MMU from their corresponding physical memory. The LSU translates abstract load and store instructions of MMU and stack into physical memory read and write accesses. This is an effective way to hide the special timing constraints of different memory types, e.g. RAM, EEPROM and FLASH. For every type of memory exists an adopted LSU. Furthermore the LSU hides the organisation of memory (8-, 16-, 32 bit). This provides high flexibility during the implementation process.

5 Conclusion

In this article we present the design of the Java Silicon Machine, a Javacard processor. The processor has hardware security features and is extendable with peripheral devices. An evaluation of the design for 0.18µ technology has shown a small die size of approximately 0.2 mm^2. The average execution time of a microcode instruction takes approximately 5 clock cycles. The duration of a Java instruction depends on the complexity of the instruction. During the development of the chip trade-offs must be made covering chip area and speed because of the severe demands on security.
The processor is a contribution to meet the requirements of different manufacturers to provide Java implementations for smart devices. For these devices the usage of JSM processor saves power consumption, chip costs and is also valuable because of its security features.

6 Literature

1] Advancel Logic Corporation, Tiny2J Microprocessor Core for Javacard Applications, http://www.advancel.com

[2] JazelleTM – ARM® Architecture Extensions for Java Applications, White Paper, www.arm.com

[3] N. Bannow, Java-processor for SmartCards and small embedded systems. (in german) Diploma thesis, Institute of Applied microelectronics and computer engineering, University of Rostock, Dec. 2000

[4] Bhaskar Bose, M. Esen Tun; LavaCORE - A Configurable Java Processor; http://www.xilinx.com/xcell/xl37/xcell37_20.pdf

[5] F. Golatowski, H. Ploog, R. Kraudelt, T. Rachui, O. Hagendorf: Java Virtual Machines for resource critical embedded systems and SmartCards,. (in german), Java Informationstage JIT 99, ITG/GI-Fachtagung, Düsseldorf, September 1999

[6] F. Golatowski, N. Bannow, D. Timmermann, Hardwareunterstützung für JavaCards: Der Javaprozessor JSM, 14. Mikroelektroniktagung 2001, ÖVE-Schriftenreihe Nr.26, pp. 141-146, Vienna, October 2001

[7] F. Golatowski, S. Preuss, H. Ploog, T. Geithner, C. Cap, D. Timmermann, Integration of Java Processor Core JSM into SmartDev(ices), 8th IEEE International Conference on Emerging Technolgies and Factory Automation, Proceedings, pp. 699-702, Antibes Juan les Pins (France), October 2001

[8] H. Ploog, R. Kraudelt, N. Bannow, T. Rachui, F. Golatowski, D. Timmermann: A Two Step Approach in the Development of a Java Silicon Machine (JSM), Workshop on Hardware Support for Objects And Microarchitectures for Java. Austin, Texas, October 1999

[9] A. Kim and J.M. Chang, Designing A Java Microprocessor Core Using FPGA Technology, Proceedings of 1998 IEEE International ASIC Conference, Rochester, NY, Sep. 13-16, http://csl.cs.iit.edu/~java/publication/asic98.htm

[10] J. Kreuzinger, R. Zulauf, A. Schulz, Th. Ungerer, M. Pfeffer, U. Brinkschulte, C. Krakowski, Performance Evaluations and Chip-Space Requirements of a Multithreaded Java Microcontroller, Workshop on Hardware

Support for Objects And Microarchitectures for Java. Austin, Texas, October 2000

[11] Moon Java Processor Core, http://www.vulcanasic.com/eda.htm, Vulcan ASIC Ltd.

[12] Java Coprocessor, Product brief, http://ww.nazomi.com

[13] O'Connor, J., Tremblay, M.: picoJava-I: The Java Virtual Machine in Hardware. IEEE Micro, 17 (2), pp. 45-53, 1997

[14] Sun Microsystems: picoJava-I Microprocessor Core Architecture. data sheet, http://www.sun.com/microelectronics/picoJava/ 1998

[15] Sun Microsystems, Inc., Java Card Technology Home Page http://java.sun.com/products/javacard.

[16] Sun Microsystems Inc., JavaCardTM 2.1.1 Virtual Machine Specification, 2000

[17] Instant Java for your smartcard, SGS Thomson, www.st.com/stonline/prodpres/smarcard/insc9901.htm