A Two Step Approach in the Development of a Java Silicon Machine (JSM) for Small Embedded Systems

H. Ploog · R. Kraudelt · N. Bannow · T. Rachui · F. Golatowski · D. Timmermann
Department of Electrical Engineering and Information Technology
University of Rostock, Germany
E-mail : hp@e-technik.uni-rostock.de

Abstract

In current solutions a Java Virtual Machine executes Java byte code by interpretation or dynamic compilation. To increase the execution performance we propose our experiences in the development of a processor architecture that can directly execute JavaCard 2.0 compliant byte code.

1. Introduction

JAVA has been developed for desktop and internet based systems but there are several implementations in the embedded area where specific Java advantages can be reused, so the idea behind JAVA and its benefits is successively transported into the embedded system-industry. In this paper we focus on *static embedded systems* in which is no need for dynamic class loading during runtime (e.g. car radio, cola machine, smart card). But the user of such systems also has the possibility to choose from a set of different applications.

To execute Java byte code on a platform there are at least 3 possibilities which are different in terms of execution speed:

- Interpreted execution
  The byte code is interpreted by software.
- Compiled execution
  In an additional step the Java byte code is compiled to a specific processor architecture so that there is no runtime overhead for interpretation. It is also possible to compile the byte code just in time (JIT), the so called dynamic compilation-technique.
- Direct execution (JSM)
  Java byte code can directly be executed by a real Java processor.

By real Java processor we consider a processor not only optimized for executing JAVA-code but capable of executing Java-code without a software implemented Java Virtual Machine.

In the last few years the importance of SUN's Java-technology raised up and it becomes a topic on many university and commercial research programs but only a small number of projects dealing with the development of a Java Silicon Machine (JSM) are known.

Many of the existing chips [7],[8] remap the Java byte code to a new (reduced) instruction set to speed up the performance. Glossner et al. described a system which is based on the same idea but they also used a multithreading architecture [3]. Another theoretical description of a Java processor architecture can be found in [6]. At the University of Zurich an ongoing project is called JAMA [4]. It seems that this processor will be designed for direct execution of JAVA byte code.

To our knowledge by now only three existing (real) Java processors are known: picoJAVA-I [12], microJava (picoJAVA-II) and JEM-1[13] . The latter one is designed by Rockwell Collins Inc. and the former one by SUN itself. Since microJava is based on the intellectual property module PicoJava-II one anticipates an increasing number of custom Java processors.

In this paper we describe our experiences in the development of a JSM for small embedded systems like smart cards.

The project is split into two parts. The first part is a software-based Java Virtual Machine suitable for 8051-processor like systems for architecture exploration, and part two is the JSM itself. The second part is still under development since this is currently a work in progress.

2. Differences between Java and Java for smart cards

A smart card is a single-chip computer based on an 8-bit microcontroller. The two most commonly used chips are Motorola's 6805 and Intel's 8051. These systems contain three different types of memory: RAM, EEPROM and ROM. The RAM is only used to store intermediate results during calculation. The EEPROM holds private cardholder values such as a private encryption key or a bank account number. The ROM is used to store the program that runs on the smart card. Smart cards are connected via five pins to the smart card reader. So these systems are “on” only if they are inserted into a reader.

The size of the die is constrained to 25 mm^2. Therefore memory space is hard limited. In average, those systems contain 4 to 20 Kbytes of ROM, 0.1 to 1 Kbytes of RAM and up to 10 Kbytes of EEPROM.

Clock frequency is typically about 3.57 to 5 MHz and an external clock has to be supplied. Smart cards can be seen as special cases of embedded systems.

SUN proposed a specification for the usage of Java on smart cards [11]. Because of the limited memory on smart cards, SUN had to remove some memory-consuming op-codes and features, like

- string manipulation,
- floating point arithmetic and
- threads.

Neither the types char, float, double and long nor operations on those types are supported. Smart cards also do not support arrays with more than one dimension. Object usage is limited as there is no <clinit>-method.

Besides these obvious modifications there are other restrictions resulting from the behavior of a smart card. Java Card systems are not able to load classes dynamically. All classes used are masked into the ROM of the card during manufacturing. Installing through a secure installation process after the card has been delivered to the smart card producer is possible too. Programs executing on the card may only refer to classes which already exist on the card as there is no way to download classes during the normal execution of application code. For more details see [11].

Due to the lack of memory space on smart cards the JVM has to be split into two parts, one for offline preparation as loading, resolving, and linking all classfiles and the other one for online execution of the Java byte code as shown in Figure 1.

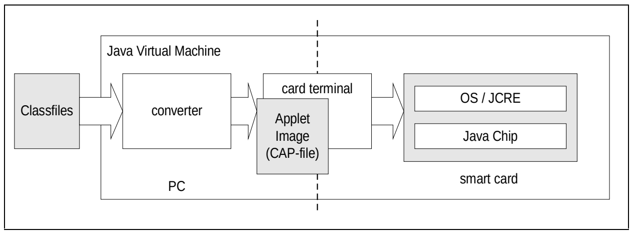


Figure 1. Separation of the Java Virtual Machine into two parts

During preparation the applets are converted to an applet images which can be directly executed on the smart card.

In conventional JVMs the byte code is verified before it is executed. In the smart card area the Byte-Code–Verifier is part of the offline block due to its time and area consumption. Therefore, a downloaded applet is not verified during runtime. To avoid illegal operating applets each applet is signed with a digital signature (CAP-file) so the card itself can determine whether the applet belongs to an environment it trusts or not.

But even a signature is no guarantee that an applet is always working correctly [10]. Unfortunately, there is no way to protect the card against transitive Trojan Horses but currently no such attacks are known. More details on Java Cards can be found in [16], [17] and [18].

3. Simulation-model based on 8051

The first step in the development process was to build a software version of a Java Virtual Machine according to SUN's JavaCard Specification 2.0, which is suitable for implementation on 8051-processor systems [5]. It is a cleanroom implementation and was used to understand the basic behavior of the JVM.

Since the goal was a Java processor some techniques which are used for describing hardware systems were used to implement the Java Virtual Machine. E.g., the execution engine of the JVM has one big “switch-case”-construct. It is figured out that such a switch-case-construct is not very well suited for a small footprint implementation. The next step was to identify groups of opcodes so that function calls like *ALU(opcode, A, B)* or *stackOp(opcode, value)* can be used. Thereby we were able to encapsulate those function-blocks.

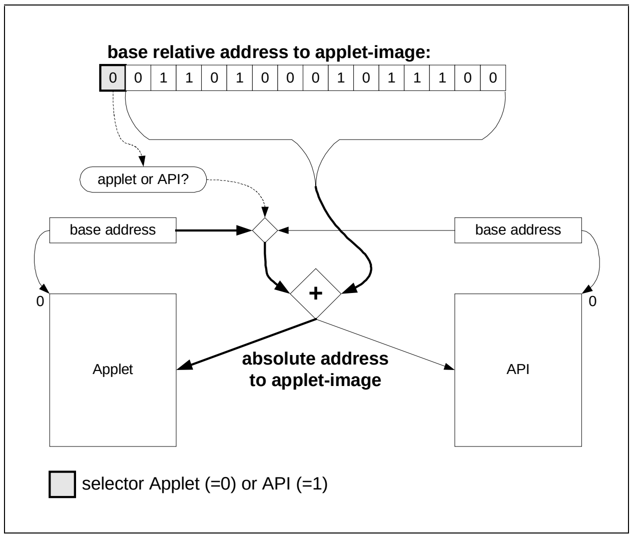
The advantage is due to the fact that the simulation model now looks similar to a structural description of a JSM so that the HW-designer just has to make minor modifications only.

On the other hand the disadvantage is that function calls result in a small software overhead.

To avoid nondeterministic and time-consuming searches in classes, methods, or fields in the constant pool it is recommended to have direct access to these structures. This can only be achieved if the addresses of all accessible objects are known in advance. Since resolved applets contain each class they need, each relative location is fixed. But instead of storing an absolute address an applet-relative address is stored. In this way applets can be moved inside the persistent memory (relocatible applets) for the cost of an additional adder.

Firmware or device specific software is located in the application programming interface (API). On smart cards the API-functionality is located in ROM and the applets are stored in EEPROM. To select an API or applet-method, the highest bit in the available address space is used as a selector (see Figure 2). This bit must be set by the converter while linking the class-files. The base address of the current active applet is stored in an applet base register. Native methods for direct hardware access are also located inside the API. These methods are necessary since different applets must be able to read data, e.g. a bank account number, or they have to increment the number of tries made to activate the smart card.

The implementation requires about 14 Kbytes of ROM on a 8051-system including the cost for the additional modularity of about 3 Kbytes. Since this is a simulation model we have not optimized the source code for the specified architecture.



base relative address to applet-image:
0 0 1 1 0 1 0 0 0 1 0 1 1 1 0 0
applet or API?
base address
base address
0
0
+
Applet
API
absolute address to applet-image
selector Applet (=0) or API (=1)

Figure 2. Access to applet-image or API

4. Moving from software to hardware

A Java Card–system is more then just a Java processor. Moreover, the virtual machine is part of the Java Card runtime environment (JCRE). The API, the executive (for handling different applets), and the native methods also belong to the JCRE.

Because of security reasons Java does not offer any byte code for direct hardware access.

To access IO using a Java Virtual Machine on standard or dedicated processor architectures, an API-function is called. Inside this function the Java-environment is left to access IO with the underlying processors IO-opcode, e.g. *mov port_adr, #val* (see Figure 3).

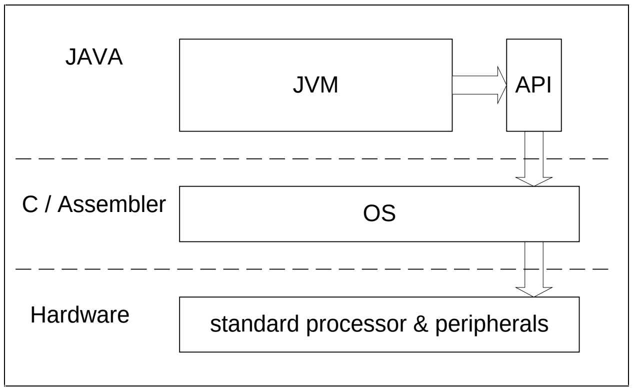


Figure 3. IO-access with JAVA

By using a Java processor there is no environment which could be left. To overcome the IO-access problem in Java the instruction set will be expanded. This is possible since two opcodes ($FE, $FF) in the opcode space are reserved for custom usage.

Due to a missing standard these solutions are proprietary. On PicoJava-II about 33% of the implemented opcodes can not be found in the Java specification.

Obviously, these new (hidden) opcodes can not be generated using an ordinary Java-compiler. Accessing these opcodes becomes somewhat difficult because a processor specific compiler has to be used.

Moreover, applets compiled in such a way are no longer interchangeable and one of the major benefits of Java gets lost.

As soon as implementation details become public knowledge, it should not be too hard to write malicious code, i.e. Trojan horses. This can not be accepted in the smart card area (revealing PIN's and POS's).

It can be shown [9] that only two additional opcodes are necessary: IO-Read and IO-Write.

In the proposed JSM the address of the opcode is traced to avoid illegal IO-access. Since application-applets are stored in reprogrammable memory the JavaCard–runtime-environment (JCRE) is located in ROM or in a *different* EEPROM. Therefore, it is possible to trace the address of the opcodes to-be-executed and a very small online-checker easily can allow or prohibit the execution of the opcode and may generate an exception in case of a fault [2].

Although IO-access is required in different native functions we only implemented IO-Read and IO-Write. Therefore we have no hardwired native functions and the functionality is realized by software inside the JCRE.

5. Java Silicon Machine

In Figure 4 the basic concept for the JSM is shown. Parts of it are already implemented using VHDL. For multi-applet cards one additional opcode has to be implemented: *set_applet*. It is used for selecting one of the applets and loads the *applet base register* with the corresponding start address of the chosen applet.

We have to adopt parts of the JSM to the new JavaCard specification 2.1, since the format of the downloadable applet is *now* specified and it is of great impact for some parts of the JSM.

The JSM is fully controlled by microcode. Therefore many changes may result in some 'software'-updates [1]. To get maximum independence between the submodules the state machines are just loosely coupled. Once these state machines are started, they run automatically until the end of the requested operation and feed back the result. The internal behavior of one state machine is completely hidden to its connected state machines.

To speed up the proposed architecture it is important to know about the dynamic probability of opcodes in a given applet. In [6] some values are given but they are based on benchmarks for complete JVM/JSM's. In the smart card area the situation is different So we are currently analyzing different applets to get the percentage distribution based on the dynamic instruction count. After the linking process the constant pool just contains constants of type integer. Those constants can only be addressed using the *ldc* or *ldc_w* opcode. So the index into the constant pool can easily be replaced with the constant itself. Consequently, we do not need a constant pool which results in some speed-up and less memory usage. Benchmarking different architectures is difficult, since execution speed depends on the compiler and coding style of an applet. We benchmarked the Dallas iButton by measuring the time to call methods and compared the results with first theoretical values of our architecture.

We assume that the iButton is a software implementation since implementation details are not published. The underlying processor itself is driven by an unstabilized ring oscillator operating over a range of 10 to 20 MHz [15]. Therefore the clock frequency of a iButton is not constant. The comparison shows speed up of 100 against the iButton (clocking the JSM @ 3.5MHz).

Techniques like caching promise some speed up but for the cost of additional hardware [6]. Since die size is limited we do not benefit from the usage of these speed-up techniques.

6. Conclusion and future work

We presented some of our experiences in the development of a JSM for small embedded systems like smart cards. We separate the development process into two parts. The experiences resulting from implementing the JVM on a 8051-system directed us to the proposed JSM-architecture. We described some problems and presented solutions for a JSM in the smart card area. It is planned to complete the architecture until the end of the year ´99. For functional verification an APTIX-system explorer M3PC containing four XCV1000-4 will be used.

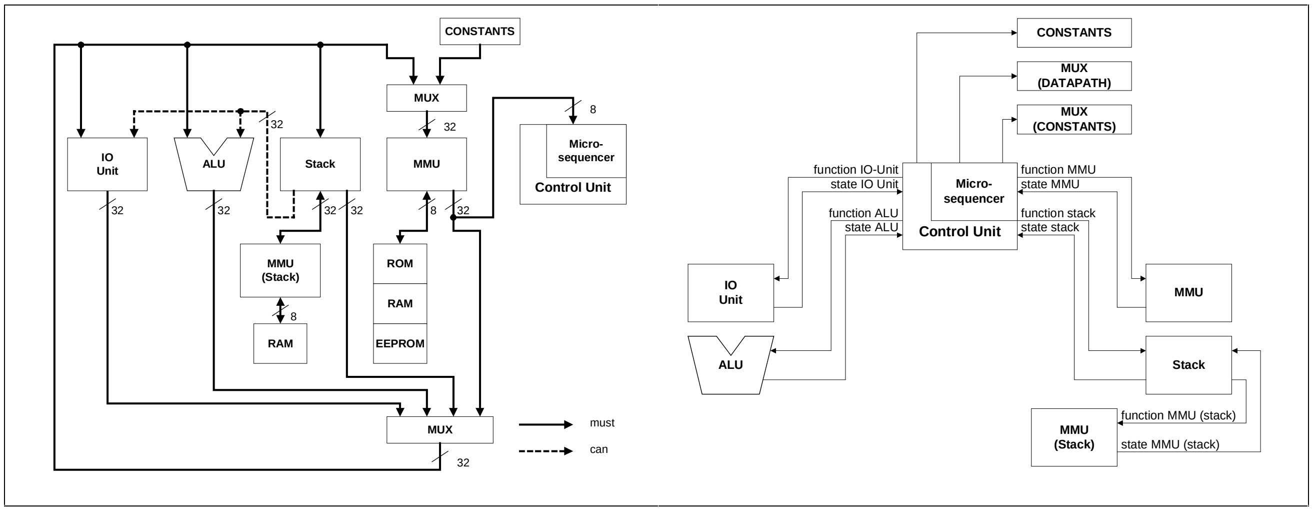


Figure 4. Datapath and control-path of the JSM

7. References

[1] N. Bannow, “*Concept of a Java-Processor,”* Technical Report, University of Rostock, Jan. 1999

[2] F. Golatowski, H. Ploog, R. Kraudelt, O. Hagendorf, and D. Timmermann, “Java Virtual Machines für ressourcenkritische eingebettete Systeme und Smart-Cards,” accepted for presentation at JIT’99, Frankfurt, Germany, Sep. 99

[3] C. J. Glossner, and S. Vassiliadis, “The Delft-Java Engine: An Introduction,“ Euro-Par 97, Conf. Proc., p.766-770, August 1997, Passau, Germany.

[4] http://www.tik.ee.ethz.ch/~jama/

[5] R. Kraudelt, „Entwicklung und Implementierung einer JAVA virtuellen Maschine (JVM) für den Einsatz in besonders ressourcenkritischen Systemen (Smartcards),“ diploma thesis, University of Rostock, 1999

[6] Vijaykrishnan Narayanan, “*Issues in the design of a JAVA processor architecture*,” PhD-thesis, University of South Florida, 1998

[7] Eric Nguyen, “*JAVA^{TM}-Based Devices from Mitsubishi”*, Java One, 1996, slides at: http://java.sun.com/javaone/javaone96/pres/Mitsu.pdf

[8] Patriot Scientific, *Java Processor PSC1000*

[9] H. Ploog, T. Rachui, and D. Timmermann, “*Design Issues in the development of a JAVA-processor for small embedded applications,”* ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA´99, Monterey, Feb 21-23

[10] J. Posegga, and H. Vogt, “*Byte Code Verification for Java Smart cards based on Model Checking,*“ 5th European Symposium on Research in Computer Security (ESORICS), Springer Verlag, 1998

[11] JavaCard 2.0 Language Subset and Virtual Machine Specification, http://www.javasoft.com/products/javacard, Sun Microsystems, Inc., 1997

[12] M. O’Connor and M. Tremblay, “picoJava-I: The Java virtual machine in hardware,” IEEE Micro, March-April 1997, pp. 45-47

[13] Rockwell, http://www.techweb.com/wire/news/1997 / 09/922java.html

[14] Dallas Semiconductor, http://www.ibutton.com

[15] Stephen M. Curry, “An introduction to the Java Ring,” http://www.javaworld.com/javaworld/jw-04-1998/jw-04-javadev.html

[16] Guthery G. B., Java card: Internet computing on a small card, Jan-Feb 1997, IEEE Internet Computing, pp/ 57-59.

[17] Michael Montgomery, “Get a jumpstart on the Java Card”, http://www.javaworld.com/javaworld/jw-02-1998/jw-02-javadev.html , 1998

[18] Zhiqun Chen , “Understanding Java Card 2.0,“ http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html, 1998