A DOUBLE-ISSUE JAVA PROCESSOR DESIGN FOR EMBEDDED APPLICATIONS

Hou-Jen Ko and Chun-Jen Tsai

Department of Computer Science
National Chiao Tung University, Hsinchu, Taiwan

Abstract—**Java applications for embedded systems are becoming popular today. CLDC/MIDP is the standard application platform for mobile phones while CDC/PBP is the emerging application platform for next generation digital TV set-top boxes. Although software-based Java Virtual Machines (VM) are prevalent, most of these VMs require a host processor running at much higher clock rate than 300MHz to reach reasonable performance. This is beyond the recommended specification of handsets and set-top boxes. In this paper, we have proposed a double-issue java processor for embedded systems. The design is not tied to any host processors and can be used as an efficient binary execution engine for a full Java Runtime Environment implementation. When synthesized on a Virtex IV FPGA (4VFX12FF66-10), the RTL model can reach over 100MHz and consumes less than 22% resources of the device.**

I. INTRODUCTION

Java Runtime Environment (JRE) is adopted by many organizations as the portable application platform for embedded systems such as mobile phones and set-top boxes. In order to support a large variety of devices while maintaining interoperability, Sun Microsystems has created the Java 2 Micro Edition (J2ME) specification and, under this framework, define different profiles and configurations for different applications [1]. For mobile phones, the Connected Limited Device Configuration (CLDC) with Mobile Information Device Profile (MIDP) has become the standard environment for Java applications. The virtual machine (VM) underneath CLDC/MIDP is a reduced-capability version of Java VM, called KVM. For DTV set-top boxes, the Connected Device Configuration (CDC) with Personal Basis Profile (PBP) are adopted as the de facto standard application environment [2]. The VM underneath CDC/PBP is a full capability VM. However, the reference implementation of CDC/PBP from Sun Microsystems is a specially engineered VM, named CVM, to facilitate porting to various embedded platforms.

There are many performance issues for adopting Java for embedded systems. First of all, object-oriented programs rely a lot on dynamic memory allocation/de-allocation which is very inefficient for embedded devices. Secondly, the Java VM model is based on a stack machine [3]. Excessive access of stack memory to store intermediate computation results is very inefficient. Finally, most embedded systems use a RISC CPU running at less than 300MHz as the host processor. The RISC architecture is usually not efficient for the execution of a software interpreter of a byte-oriented machine language [4][5].

There have been many efforts to improve the performance of a Java VM [4]. For embedded devices, software-based approaches such as Just-in-Time (JIT) compilation are less suitable since JIT compilers requires extra memory and the overhead of the on-the-fly compilation process is more noticeable and intrusive for embedded systems with slow RISC processors. For hardware-based solution, there are co-processor approaches (such as ARM Jazella) and java processor approaches [5][6]. An interesting work is the Java processor, JOP, designed by Schoberl [5] since the complete RTL model (written in VHDL) is available to general public. JOP defines its own application profile/configuration, which is closer to CLDC than to a full JVM. The RTL model of JOP has been ported to many devices. However, the performance still has a lot of room for improvement.

In this paper, the design of a double-issue Java processor is proposed. The advantage of designing a stand-alone Java processor instead of a co-processor is that the design will not be tied to certain host processor. However, since a stack machine is not efficient for I/O and control tasks, a general purpose processor is still required to complete the system. The paper is organized as follows. Section II provides an overview to a full Java Runtime Environment design and discusses how a Java processor can be integrated into the environment. The proposed double-issue Java processor is presented in section III. Section IV describes the target platform and shows the synthesis report of the RTL model. Finally, some discussions are given in section V.

II. JAVA RUNTIME AND INTEGRATION OF A JAVA PROCESSOR

A complete JRE is a sophisticated software system. The key components of a JRE include a bytecode execution engine (BEE), a dynamic class loader, a garbage collector, and standard class libraries (Fig. 1). Among these components, only the BEE can be reasonably implemented in hardware. For software-based VM, the BEE is implemented as an interpreter. The integration of this "virtual hardware" with the rest of the software components is simpler since everything is implemented in software. However, for a hardware-assisted JRE, the BEE will be replaced by a Java processor. In this case, the JRE becomes a highly integrated hardware/software system. The link between a Java processor and the rest of the JRE is the dynamic class loader.

When the JRE is assigned to run a Java program, the initial class file will be loaded and parsed. All the static content of the classes inside the class file (e.g. method codes and data field information) will be registered in the method area. An object will be allocated on the heap to instantiate the root class. The object will contain a copy of the private data fields of the root class. At this point, the program counter of the Java processor will be set to point to the initial method in the method area. During execution, the Java processor will fetch bytecodes from the method area and access data fields of the object in the heap and in the method area.

In general, the class loader is responsible for locating/loading the class files (Java application images) and setting up the method area for the BEE. Therefore, it is more suitable to execute the class loader on the host processor. The proposed JRE is shown in Fig. 2. In this paper, we only focus on the design of a Java processor that can be used to replace the BEE.

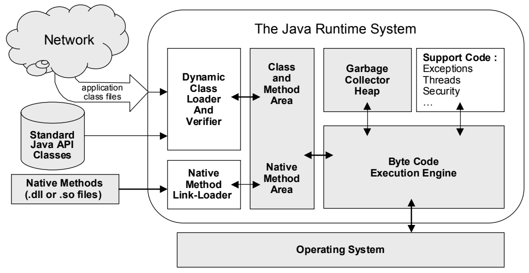
Network
application class files
Standard Java API Classes
Native Methods (.dll or .so files)
The Java Runtime System
Dynamic Class Loader And Verifier
Class and Method Area
Garbage Collector Heap
Support Code : Exceptions Threads Security …
Native Method Link-Loader
Native Method Area
Byte Code Execution Engine
Operating System

Fig. 1. A standard Java runtime system

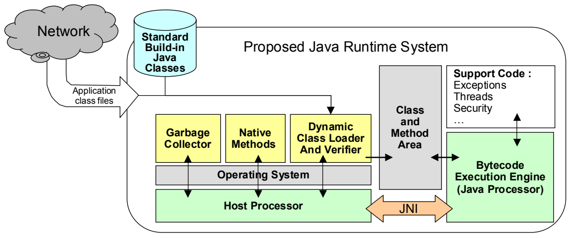
Network
Standard Build-in Java Classes
Proposed Java Runtime System
Application class files
Garbage Collector
Native Methods
Dynamic Class Loader And Verifier
Class and Method Area
Support Code : Exceptions Threads Security …
Operating System
Bytecode Execution Engine (Java Processor)
Host Processor
JNI

Fig. 2. The proposed Java runtime system

III. PROPOSED DOUBLE-ISSUE JAVA PROCESSOR

In this section, the detail design of a double-issue Java Processor is presented. For a double-issue processor, two machine instructions are executed per cycle. It is important to point out that a Java processor in general does not execute bytecodes directly because some bytecodes are much more complex than a traditional machine instruction. Therefore, for the proposed processor, the native instruction set (referred to as the microcodes, following the convention in [5]) is different from the bytecode instruction set. A bytecode will be translated into one or more microcodes on-the-fly. The proposed processor has a four-stage pipeline which is shown in Fig. 3.

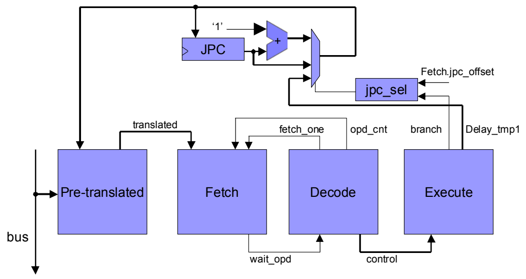

Fig. 3. Overall Java processor architecture

III.1 Pre-translation Stage

The Java bytecodes are divided into simple bytecodes and complex bytecodes. At the pre-translation stage, each simple bytecode is translated into a microcode, while a complex bytecode is translated into a pointer that points to the address of a microcode sequence stored in ROM. A Java bytecode instruction may be followed by zero, one, or more operand bytes. Therefore, it is not trivial to fetch two bytecode instructions per cycle (along with the operand bytes) due to this variable length instruction nature of Java bytecodes. Obviously, the instruction must be decoded to some degree before the fetch stage so that the processor knows how many bytes it has to fetch in order to retrieve two complete instructions with operands. The pre-translation module is designed to classify-and-tag the bytecode streams so that the fetch module can identify the number of bytes to fetch.

As shown in Fig. 4, the pre-translation module fetches four bytes at a time from the bytecode section of the method area. The bus_wr signal triggers the counting of the Pre-Fetch Program Counter (PFPC). The PFPC signal will select one byte per cycle. Each byte is sent to the Translation ROM and the Operand Count ROM. These two modules classify the bytecode into one of three types, namely, one-to-one mapping, one-to-many mapping, and operand. For the first two cases, the translation ROM produces instruction data which could be a native microcode (for one-to-one mapping) or an address (for one-to-many mapping). If the translated instruction data is a microcode, it means that the Java bytecode can be mapped to this Java Processor microcode. If the translated instruction data is an address, the address will be used in the fetch stage to retrieve the corresponding microcodes. At the same time, the value retrieved from the Operand Count ROM is decreased by one, which indicates the number of remaining operand bytes.

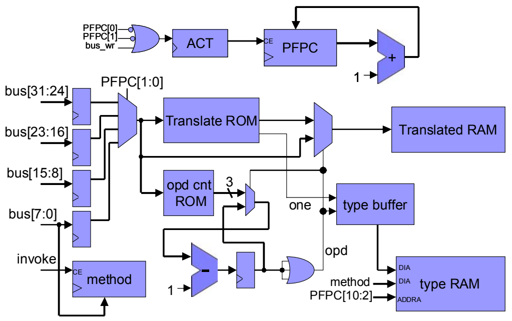

Fig. 4. Pre-translation stage

III.2 Fetch Stage

After the pre-translation stage, the translated instructions and the tags are stored in the Translated RAM and Type RAM, the fetch module (see Fig. 5) retrieves the translated values. At the fetch stage, the Type Management module determines the type of the next two translated instruction/operand data to be decoded.

First of all, the fetch stage must ensure that the data fetched from the RAM have been translated. Otherwise, it will trigger an exception to translate the corresponding method area and lock the current JPC (Java Program Counter) until the translated data arrive. For every two instructions fetched, the second one is always stored in a register first in case the processor could not execute two instructions simultaneously. When this happens (when the LSB of the JPC is one), the registered instruction will be send to the decode unit, alone with the next translated instruction fetched from the RAM.

Secondly, the mode register stores the current status to distinguish between "simple bytecode mode" and "one-to-many mapping mode." In the simple bytecode mode, the fetch stage always fetches two translated values from the pre-translation module. A translated value could be a microcode or an operand value. The translated value is stored to the operand buffer if they are of operand type. In the "one-to-many mapping" mode, the instruction is extracted from the one-to-many instruction ROM table, indexed by the corresponding translated instruction data, namely, an address. At the same time, this address also adds to the offset value to index the next address and stores the result to the address register. During one-to-many translation mode, the instructions are fetched from the one-to-many instruction ROM. This mode is maintained until the next signal is extracted from the one-to-many ROM indicating that the microcode sequence of the complex bytecode instruction is complete (This design is similar to that in [5]).

Finally, the last two signals, "decode.opd_cnt" and "decode.fetch_one," are the signals from the decode stage. The signal "opd_cnt" indicates the number of bytes of the microcdes the decode stage needs. The Type Management module will determine the opd_cnt value and update the operand buffers. The other signal "fetch_one" indicates that the microcodes of the decode stage encountered a structure hazard that the Java processor can not execute this combination of the two microcodes in one cycle.

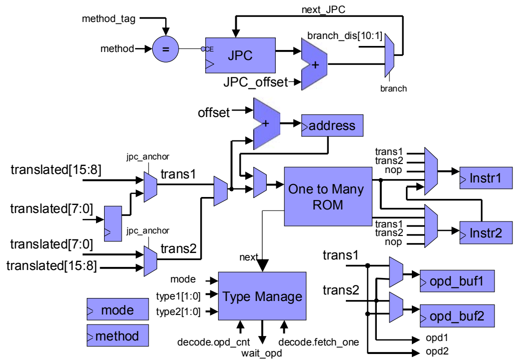

Fig. 5. Fetch stage

III.3 Decode Stage

At the fetch stage, two complete microcode instructions and the operands that these microcodes need are fetched into the processor. The next stage is the decode stage which is shown in Fig. 6. The opd_cnt signal is the number of bytes of operands that the microcode instructions needed and the fetch.wait_opd signal indicates that the operands is not ready and these microcodes should wait until the fetch stage fetches enough operands.

There are two immediate value ROMs at the decode stage because we must support two immediate load operations. The tmp1 and tmp2 signals could represent various items: an immediate value, a stack address of the RAM and an address of register bank. There is an advantage to generate these addresses at the decode stage. Due to RAM read pipelining, if the addresses are prepared early, the data can be read from RAM without any wait cycle. Finally, for store operations, it takes one cycle delay to store the tmp1 and tmp2 values in the registers.

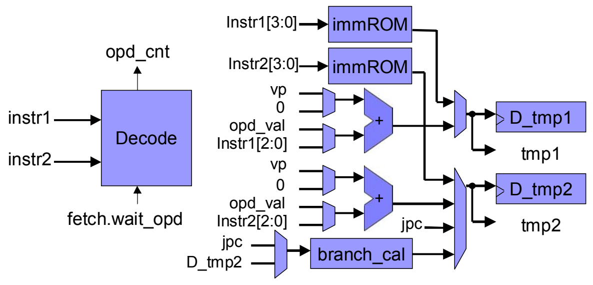

Fig. 6. Decode stage

III.4 Execution Stage

The data path of the execution stage is shown in Fig. 7. The top of stack is store in the register labeled A. The top-1 and top-2 entries of the stack are labeled B and C, respectively. Each operation is performed with registers or load values as sources. This data path can handle parallel execution of any combinations of two instructions except two ALU operations because of the structure hazard. The load values could be from the local variables or the stack data. When the stack pointer decreases, the registers should update the values and the stack value needs to load from the memory for more top values. On the other hand, when the stack point increase, the new value store to the top registers and the value that spill from the register should be write back to the memory.

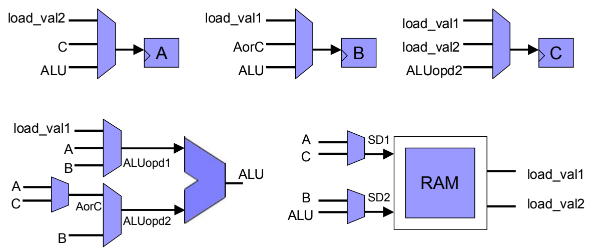

Fig. 7. Data path of Execution Stage

In order to execute two instructions per cycle, the memory bandwidth requirement would also increases. In the proposed design, two RAM devices are used to serve this purpose (Fig. 8). One of the RAM handles memory requests for addresses with LSB 0, and the other one handles requests for addresses with LSB 1. The read address or the stack pointer is generated at the decode stage without any delay. There is a condition that causes conflict between these two RAM devices. When two read and write addresses have the same LSB value, it would try to access the same RAM devices. Fortunately, the only condition for this case to happen is when two operations try to load or store the local variables with the same LSB. The probability of this scenario is relatively low, so we do not add extra logics to support it. We simply avoid this condition at the decode stage, and it will not happen at the execution stage.

TABLE I. Synthesis report of the Java Processor

Device utilization summary:				
Device: 4vfx12ff66-10				
Number of Slices:	1190	out of	5472	21%
Number of Slice Flip Flops:	398	out of	10944	3%
Number of 4 input LUTs:	2237	out of	10944	20%
Number of bonded IOBs:	67	out of	320	20%
Number of FIFO16/RAM16s	4	out of	36	11%
Number of GCLKs:	1	out of	32	3%
Number of DSP48s:	3	out of	32	9%
Minimum period: 9.485ns (Maximum Frequency: 105.430MHz)				

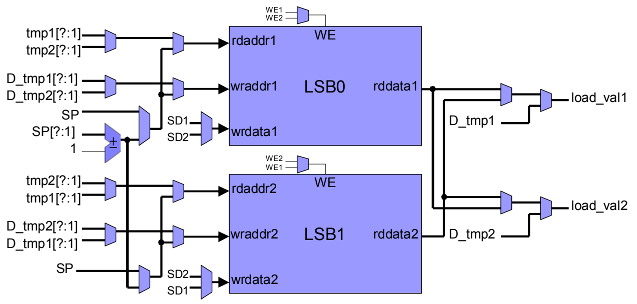

Fig. 8. Memory Architecture

IV. IMPLEMENTATION RESULT

The proposed Java processor is implemented on an SoC emulation platform, the Xilinx ML-403. The platform is based on a Virtex 4 FPGA with a PowerPC core running at 300MHz. The RTL model of the Java processor is written in VHDL and the synthesis report using SynplifyPro for the Virtex IV device is shown in TABLE I. The full JRE software system proposed in section II is still under development so the integration of the Java processor and the JRE is not done yet. However, the proposed JRE will be based on the CVM implementation, which has been ported to the target platform already.

V. CONCLUSIONS

For embedded systems with host processors running under 300 MHz, hardware-assisted JRE is the most efficient way of supporting Java applications. This paper proposes a double-issue Java processor and the design have been implemented on a Xilinx Virtex-4 FPGA. We also propose the architecture of a full JRE that can be integrated with this Java processor. Future works will be focusing on modifying CVM to fit the proposed system.

VI. ACKNOWLEDGEMENT

This research is partly funded by National Science Council, Taiwan, R.O.C., under grant number NSC 95-2219-E-002-012.

REFERENCES

[1] Q. H. Mahmoud, *J2ME for Home Appliances and Consumer Electronics Devices*, *Sun Microsystems White Paper*, Jan. 2003.

[2] Digital Video Broadcasting (DVB), *Multimedia Home Platform (MHP) Specification 1.0.2, ETSI TS 101 812*, June, 2002.

[3] T. Lindholm and F. Yelling, The Java Virtual Machine Specification, Addison-Wesley, 1996.

[4] A. Krall, K. Ertl, and M. Gschwind, Java VM Implementation: Compilers versus Hardware, *John Morris (ed.), Computer Architecture (ACAC '98)*, Perth, pp. 101-110, 1998.

[5] Martin Schoberl, *JOP: A Java Optimized Processor for Embedded Real-Time Systems*, *Ph.D. Thesis*, Tech. Universitaet Wien, Jan 2005.

[6] A. Kim and M. Chang, "Designing a Java Microprocessor Core Using FPGA Technology," *Computing & Control Engineering Journal*, June 2000, pp.135-141.