國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

嵌 入 式 J a v a 加 速 器 系 統 設 計

Design of Java Accelerator IP for Embedded Systems

研 究 生：郭瀚文

指導教授：蔡淳仁　教授

中 華 民 國 一 百 年 八 月

嵌入式 Java 加速器系統設計
Design of Java Accelerator IP for Embedded Systems

研　究　生：郭瀚文　　　　　　Student：Han-Wen Kuo

指導教授：蔡淳仁　　　　　　Advisor：Chun-Jen Tsai

國　立　交　通　大　學

資　訊　科　學　與　工　程　研　究　所

碩　士　論　文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年八月

Abstract

Java Runtime Environment (JRE) is becoming a popular application platform for complex multimedia embedded systems today. In this thesis, we present the architecture design of a reusable Java accelerator IP for application processors for embedded systems. The accelerator IP cooperate with a general purpose processor (GPP) core to support the JRE. The GPP core is responsible for running service routines to support Java tasks such as I/O requests, dynamic class loading, heap memory management, etc. The proposed Java accelerator IP is a double-issue Java core in charge of execution of the Java applications. More importantly, it is easy to integrate such Java accelerator IP into existing embedded systems both hardware-wise and software-wise. On the software side, it does not rely on any full-blown OS (such as Linux) running on the GPP. Only a thin kernel that maintains the execution of interrupt-driven Java service routines is necessary to support the JRE. On the hardware side, the communication between the Java accelerator IP and the GPP core is achieved using a memory sharing table and an interrupt-driven mailbox device. The proposed Java embedded platform with the Java accelerator IP has been implemented on the Xilinx Virtex-5 ML507 FPGA development board.

Acknowledgement

這篇論文的完成，首先必須要感謝我的指導教授蔡淳仁教授。在研究所的這兩年期間，除了老師提供了很多學習的經驗跟機會外，在與老師討論的過程中，我也了解到何謂邏輯的謹慎推演以及細節的注意。也感謝實驗室的學長帶領我們一起執行國科會計畫，透過計畫的合作互相研討，認識了不同的領域與研究。再與實驗室的同學與學弟一起進行研究時，讓我了解到自己仍有許多地方思考不夠嚴謹，也學習到如何清楚表達自己的想法給別人，以及如何與別人的溝通中提取重點。真的很高興在實驗室的夥伴願意和我一起努力，讓我自己在這兩年間成長了許多。感謝 MMES LAB 以及實驗室的大家。

Content

List of Figures

List of Table

Chapter 1. Introduction

1.1 Motivation

For the past few years, Java is getting considerable attentions from embedded system application platform langrage because of the strong portability of applications across different operating systems and processors. Adopting Java as a standard application langrage also can enjoy the miscellaneous API support. Not to mention the fact that the Java language is a well-designed object-oriented programming language and up to 40% more productive than C++ [19]. As a result, many embedded multimedia systems are Java-based platforms. There are many different Java middleware stacks selected by different organizations to support different service requirements. Take an example from Java's creator Sun Microsystems, the Java 2 Platform, Micro Edition (J2ME) [9] framework is one of the Java platforms that are designed for embedded devices. Another example in recent years is Google's Android platform. The Android platform includes Java class libraries based on Apache Harmony (a Java 5 API compliant but not binary-compatible middleware implementation) so most Android applications are written in Java.

1.2 Java OS Model for Java Accelerator IP Design

The Java runtime model adopted in this thesis is based on the Java OS model [12][13]. Although the intension of the original Java OS model is to use minimal native C code to support the complete JRE of a software-based VM, the concept is nevertheless very suitable for encapsulation of a hardware Java accelerator as a reusable IP for application processor SoCs. The SoC software stack shown in Fig. 1 illustrates this concept. The proposed Java platform is composed of a generic RISC processor and the Java Accelerator IP (JAIP). Since the Java VM model is based on a stack machine, many operations (e.g. I/O) are very inefficient. Therefore, the RISC-side software will handles such low-level Java service tasks. The typical service tasks include I/O controller management (i.e. management of device drivers), file system accesses, physical memory management, communication tasks, management of application accelerators (e.g. audio/video codecs), etc. That is, the software of the RISC-side provides a hardware abstraction layer (HAL) for the Java platform. On the other hand, the JAIP is responsible for all general bytecode execution except the bytecode execution relative to typical service tasks. Note that the RISC core does not prohibit adoption of a sophisticated OS kernel, and the intercommunication is achieved by memory sharing and interrupt. Therefore, the proposed Java platform has little dependency on any particular OS kernels and processors. More specifically, it is easy to integrate such Java system model into existing embedded systems (both hardware-wise and software-wise).

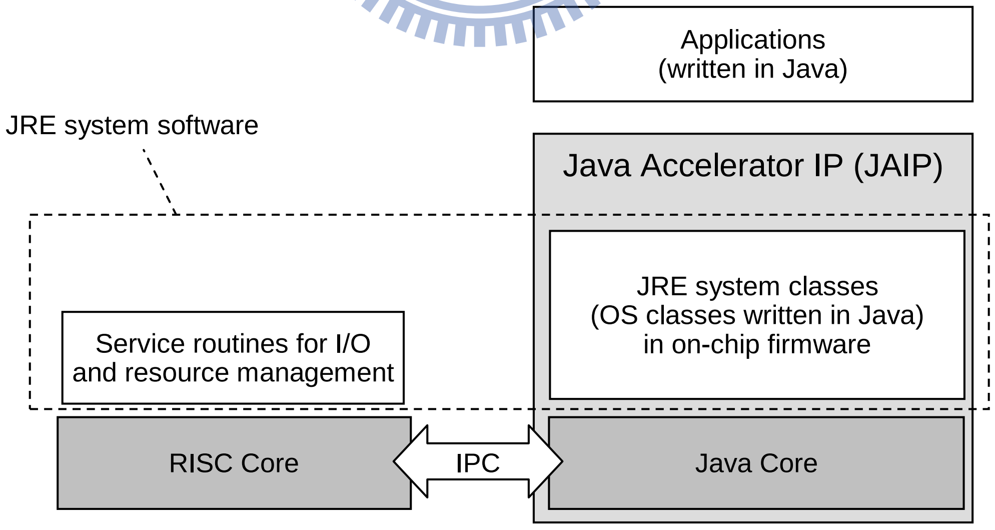


Fig. 1. The concept of JRE encapsulated inside a reusable IP. The RISC-side system software is minimized to facilitate integration of the Java accelerator into existing application processors.

1.3 Scope of this Thesis

In this thesis, we propose a hardware Java accelerator approach which is a reusable IP for application processor SoCs. The proposed JAIP in this thesis is based on the work in [2] [4] [6]. The organization of this thesis is as follows. Chapter 2 discusses some related Java runtime environment designs and describes the previous design of the Java embedded platform [2] [6]. Chapter 3 describes the hardware architecture of JAIP in detail, while chapter 4 discusses the design of dynamic symbol resolution mechanism. The implementation of the proposed JAIP using an FPGA is presented in chapter 5, along with some performance analyses and benchmark results. Finally, conclusions and discussions are given in chapter 6.

Chapter 2. Previous Work

2.1 Java VMs for Embedded Systems

One important characteristic of Java is portability, which means that the compiled binary code of Java classes should be executable on any JRE systems regardless of its underlying hardware or OS platforms and producing exactly the same behaviors. This is achieved by compiling the Java language programs to the intermediate codes called Java bytecodes. Java bytecode instructions are analogous to machine codes, but usually are executed by a virtual machine (VM) written specifically for the host hardware. End-user will use a JRE installed on the target device for Java applications. To make clear distinction among various Java devices, Sun Microsystems develops three different JRE, Java Platform, Enterprise Edition (Java EE), Java Platform, Standard Edition (Java SE), and Java Platform, Micro Edition (Java ME). Java ME was formerly known as Java 2 Platform, Micro Edition (J2ME)[9]. J2ME is designed specifically for embedded systems. Target devices range from industrial controllers to mobile phones and set-top boxes. Furthermore, the J2ME has been divided into two base configurations, one to fit small mobile devices and one to target towards more capable devices like smart-phones and set top boxes. The configuration for small devices is called the Connected Limited Device Configuration (CLDC) [10] and the more capable configuration is called the Connected Device Configuration (CDC) [11].

Traditional JRE is composed of a Java Virtual Machine (JVM) [14] and a set of standard class libraries. The software-based Java VM is implemented on non-virtual hardware and on standard operating systems. JVM adopts stack architecture and its organization is shown in Fig. 2. In the Java virtual machine specification, the three major components are the class loader subsystem, the runtime data area, and the execution engine [7]. The class loader is a mechanism for loading classes or interfaces given fully qualified names. The runtime data area is the major memory space that the Java VM organizes to execute a program. The execution engine is another mechanism that is responsible for executing the instructions contained in the methods of loaded classes. More details of the Java class file format is described in the VM specification [14].

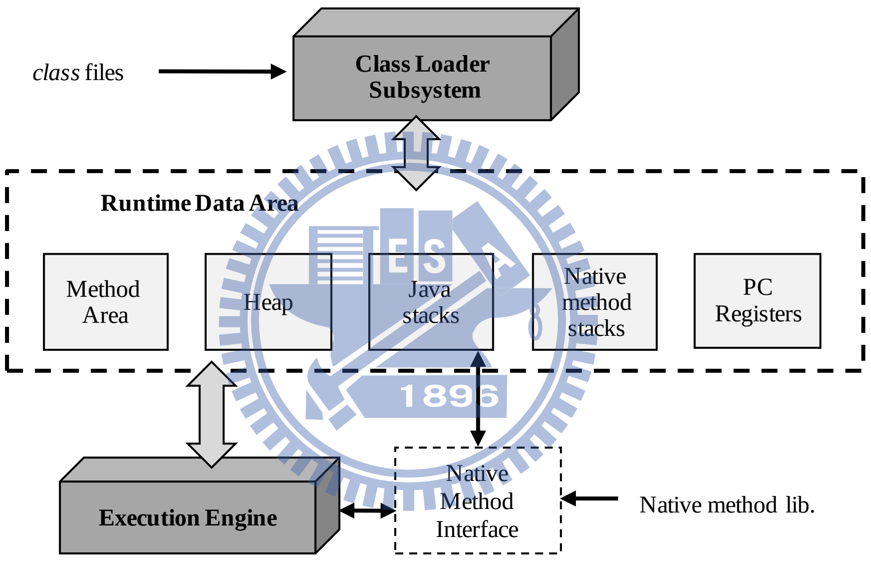


Fig. 2. The block diagram of Java virtual machine.

The Java VM will interpret and execute the Java bytecodes at runtime. However, implementing a stack-based virtual machine using an interpreter has a great impact on the performance, especially for the embedded processors. The interpreters are slow (up to an order of magnitude slowdown) and require memory to store the interpreter code. In addition, many operations are expensive for embedded processors, such as simulation of a stack-machine, dynamic symbol resolutions, and heavy dynamic memory allocations. Therefore, there are many performance and memory constraint issues for embedded systems with a regular processor. The solutions for improving the time and space overhead of JRE can be roughly divided into three approaches. The Just-in-Time (JIT) compilers, the hardware-based co-processors (e.g. ARM Jazelle), and sand-alone Java processors (e.g. picoJava) are common measures for embedded environments.

The execution speed improves significantly by using JIT compilation techniques [16] [17] to translate Java bytecodes to native codes at runtime. Although, the speedup by the JIT compiler is high, JIT requires extra memory [17] and imposes extra compilation overhead for class loading. The compiler itself along with the memory footprint for the compilation may require a few megabytes of storage [16]. Therefore, this approach is less suitable for embedded devices, which have strong memory constraints. An interesting effort is taken on by Google when picking a solution for their Android application environment. The Dalvik VM [8] is a register-based virtual machine which is not binary compatible with the JVM. Java application class files must be converted into Dex file format before execution. It is shown that a register-based VM can be 32.3% more efficient than a stack-based VM when executing standard benchmarks by an interpreter, at the expense of 25% larger binary code size of the benchmark programs [20].

It is important to point out that, the implementation of most JRE heavily relies on the underlying operating system. However, most Java middleware stacks of JREs have already included main functions of a typical operating system. Therefore, adopting a complete operating system underneath a JRE is a duplication of system functions, which is not a good design philosophy for embedded devices with resource constraints.

Building a purely hardwired Java processor is a nontrivial task, as the Java VM instruction set is quite complex. Some instructions are even more complex than the instructions of a traditional CISC CPU such as the Intel x86 and Motorola 680x0 families. There are several Java processor solutions such as picoJava [22], Komodo [23], jHISC [24], and JOP [25]. Because the processor is custom-designed to match the stack machine model of the JVM, it can deliver better bytecode execution performance than that of a general-purpose processor running a Java interpreter. However, a single core stack machine designed to support Java complex low-level communication and multimedia controls may not be the most proper architecture for JRE. After all, many communication and multimedia tasks can be executed more efficiently using traditional register-based processor architecture.

The final approach is the Java co-processors approach used in, for example, ARM Jazelle [15]. These proposals use improved stack machines to execute the Java bytecodes and used resister-based processor to handle other tasks. Unfortunately, such architectures are tied to specific processor architecture and are not generally available for other host processors.

2.2 Double-issue Java Core Proposed by Ko

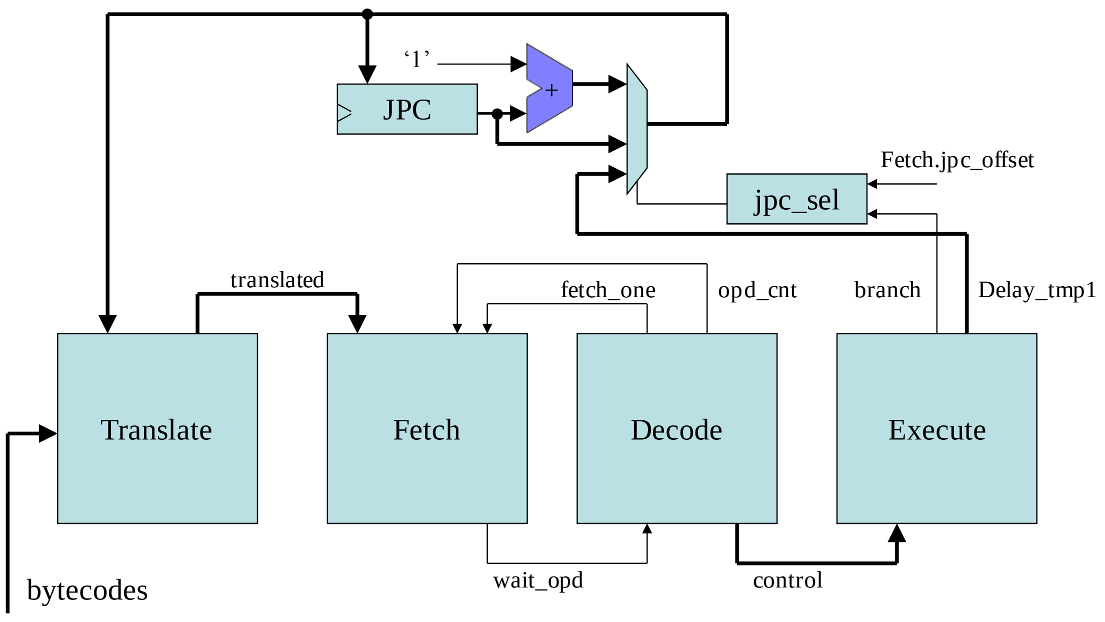


Fig. 3. Overall Java Processor Architecture.

Ko et al. [1] [2] proposed a design of double-issue Java Bytecode Execution Engine (BEE) core. The architecture of Java bytecode execution engine is shown in Fig. 3. The BEE core adopts a four-stage pipeline architecture with translate, fetch, decode, and execute stages. The Java BEE core is a stand-alone IP not tied to any host processor architecture. Therefore, it is easy to integrate the BEE core into any processor that supports interrupt-driven inter-processor communications. The Java BEE core chooses two-level stack architecture with three registers stored top three of stack items. The stack architecture is shown in Fig. 4. For a double-issue core, the BEE executes two microcode instructions per cycle. The Java BEE does not execute bytecodes directly. Instead, Ko defines a RISC-like instruction set architecture (ISA), and the Java BEE translates a Java bytecode either into one or more instruction on-the-fly.

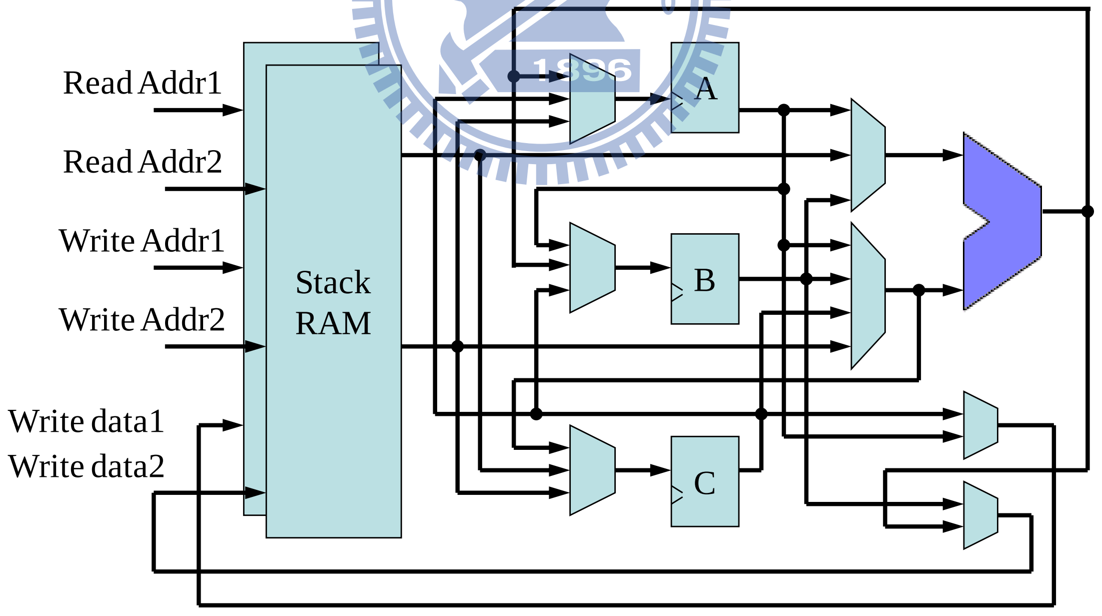


Fig. 4. A Two-level Stack Cache with Double Issue

2.3 Dynamic Class Loading Mechanism Proposed by Hwang

Hwang et al. proposed a dynamic class loader [5] [6] for the heterogeneous dual-core SoC system [3] [4]. In the previous system [1] [2] [3] [4], the class loader parses all class files and convert them into runtime information images and reserves all resolution information in the constant pool of the image. Hwang also designed a dynamic resolution state machine to handle symbol resolution of constant pool data. The states of method invocation and field data access do resolution and get information which produced by class loader before execution as shown in Fig. 5. This state machine controls and changes the program counter for referenced method bytecode as well.

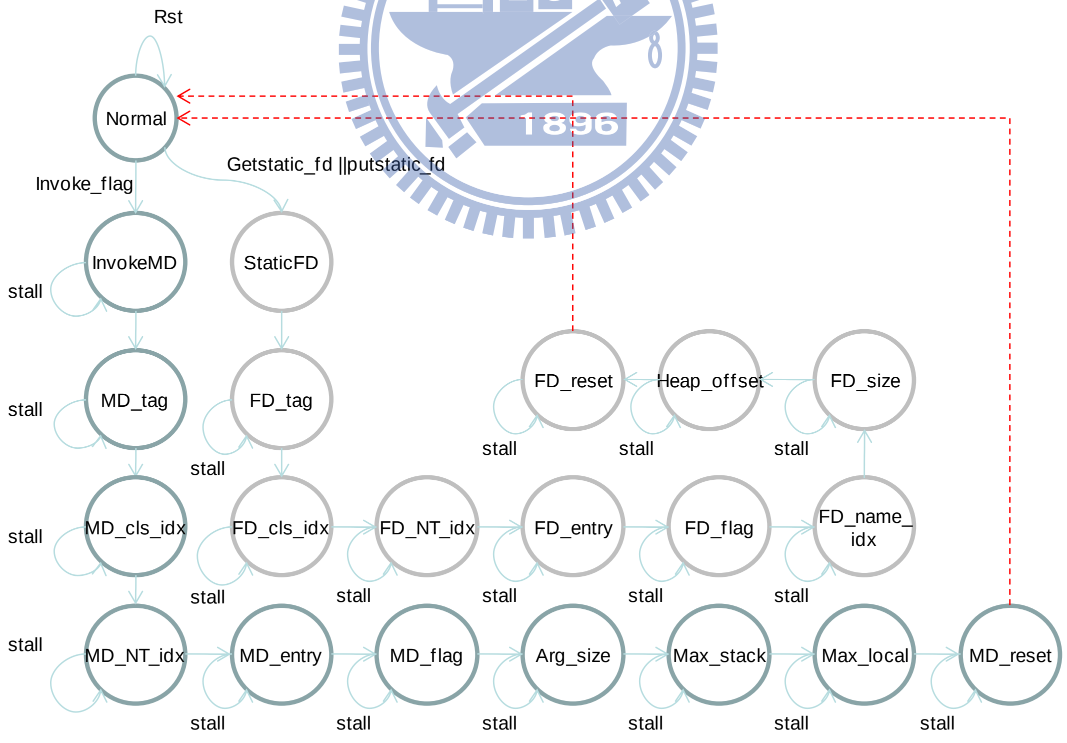


Fig. 5. Controller state machine for method invocation resolution in [6].

2.4 Contributions of this Thesis

Although, the double-issue Java Bytecode Execution Engine (BEE) core designed by Ko et al. [1] [2] has many advantages, the BEE‘s architecture (e.g. stack structure and hazard detection) cannot bear some common combination of instruction (e.g. a special combination of two load instructions). The prototype of the dynamic class loading mechanism and method area management was first proposed in [6]. However, the support of dynamic symbol resolution is not complete.

In this thesis, we propose a hardware Java accelerator approach which is a reusable IP for application processor SoCs. We also propose a Java embedded platform with a heterogeneous dual-core design which is composed of a generic RISC core and the proposed JAIP. The proposed JAIP in this thesis is based on the BEE [2]. Nevertheless, we redesign the BEE to increases both reliability and capability. The new design also has a powerful dynamic symbol resolution mechanism which supports the interface method invocation and system native method invocation.

Chapter 3. The Hardware Architecture of the Java Accelerator IP

3.1 Overview

In this chapter, we present the architecture detail of the proposed Java accelerator IP (JAIP). Although the JAIP is derived from the architecture described in [1] [2], we have redesigned the whole accelerator architecture. Only the double-issue architecture design concept, four-stage pipeline design, and part of the instruction set architecture are maintained in this new design. The JAIP increases the reliability of the Java core and supports more Java langrage's object-oriented features.

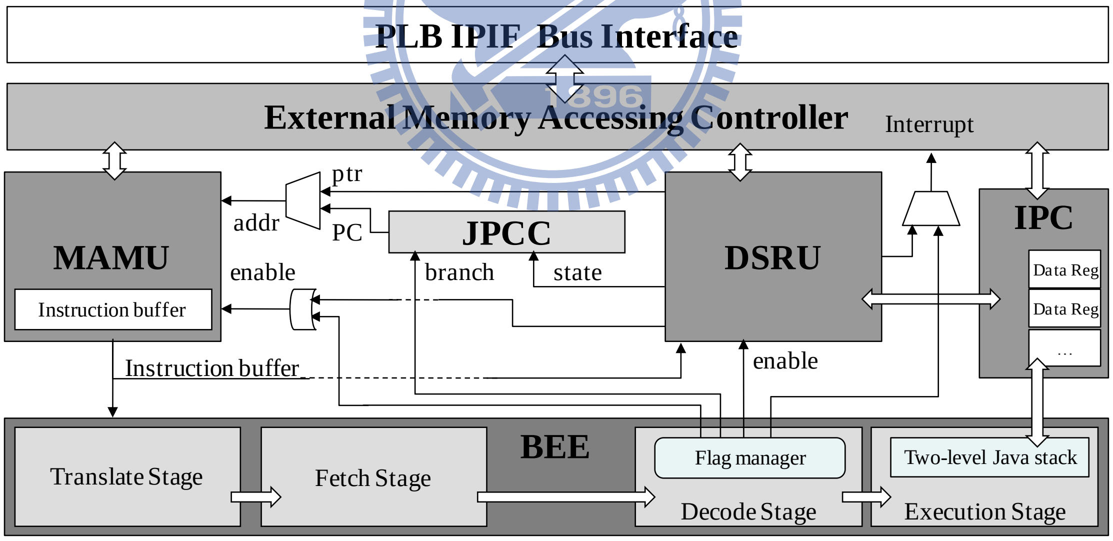


Fig. 6. Architecture Diagram of the Proposed JAIP.

This IP is comprised of several parts as shown in Fig. 6. The key components include a Java Program Counter controller (JPCC), the Method Area Manager Unit (MAMU), the double-issue Bytecode Execution Engine (BEE), the Dynamic Symbol Resolution Unit (DSRU) and the Inter-Processor Communication (IPC) Unit. The

JPCC is a simple logic just like other processor’s program counter controller. It controls a register containing current program count, and conditionally switches the input source of the counter register. For performance issue, there is an instruction cache like unit named MAMU in the JAIP. The difference between the MAMU and a traditional instruction cache is that the MAMU caches a complete class image at a time. The implementation of dynamic class loading is achieved by the MAMU and the DSRU. The DSRU is another key architecture to support the dynamic resolution of the Java language. Java stores the runtime linking-and-loading information in the constant pool, but resolving a runtime symbol to a physical address usually takes several indirect references. We design a software parser to simplify and reorganize the constant pool, so that the JAIP can easily get the resolution information through the DSRU. The BEE is the most important circuit in the JAIP. Since all the Java bytecode are executed by the BEE, the design of the data path dominates the performance. Therefore, we adopt double-issue architecture and pipeline to the BEE. The final part is the IPC, an interrupt communication interface between the RISC core and the JAIP. When the JAIP requests the RISC core for interrupt services, the IPC prepares the arguments or information for the RISC core during the services. The IPC also can return the data from the RISC core after the service routine finishes its execution.

The remaining sections of this chapter describe more details of the hardware architecture and discuss the design and implementation principal that makes JAIP more solid. Because a complete description of the dynamic resolution mechanism involves the design of the software service routines, it will be described in next chapter.

3.2 Method Area Manager Unit

This section will introduce the hardware architecture, function, and design principal of the MAMU. The MAMU is comprised of three parts, the controller, the Method Area Circular Buffer (MACB), and the instruction buffers. The concept of the MAMU is similar to an instruction cache. Although the cache block size is of two Kbytes, but a class runtime image will be loaded into cache in contiguous cache block once it is referenced. The main function of the MAMU is to manage the class runtime images stored in the MACB. To achieve this goal, we design the Class Information Table and the Circular Buffer Allocation Table, which is maintained by a dynamic class loading C code running on the RISC core. And the MAMU loads class runtime images into the MACB dynamically base on the two tables. The details of the class runtime images and the dynamic class loading C code will be discussed in next chapter.

3.2.1 Two-Level Method Area Design

In order to keep a balance between performance and resource usage, the JAIP adopts a two-level method area design. It treats the DDR-SDRAM as a L2 method area, and the MACB is the L1 method area. The MACB is composed of 32 small memory blocks that store dynamically loaded class runtime images in FIFO. The Java core is designed to only fetch method bytecodes, access non-string constant values, and lookup dynamic resolution information from the MACB. That is, a class runtime image has to be loaded into the MACB before execution. Since all the local branches in a Java method use relative references and are confined within the class image, the MACB has to be updated only upon inter-class invocations and returns. Therefore, we will encounter one of three conditions in switching to another class: 1. the target class runtime image is store in the MACB, 2. The target class runtime image is not in the MACB but in the DDR-SDRAM, and 3. The target class runtime image does not even exist upon invocation. The first two conditions are similar to hit and miss of a cache, and it will be described clearly in section 3.2.2. The third condition occurs because the JAIP supports dynamic class loading. However, due to the DSRU, the situation will be eliminated. The class runtime image has already been established by the DSRU before the switching operation. More details will be described in section 4.3.1.

3.2.2 Method Area Manager Unit

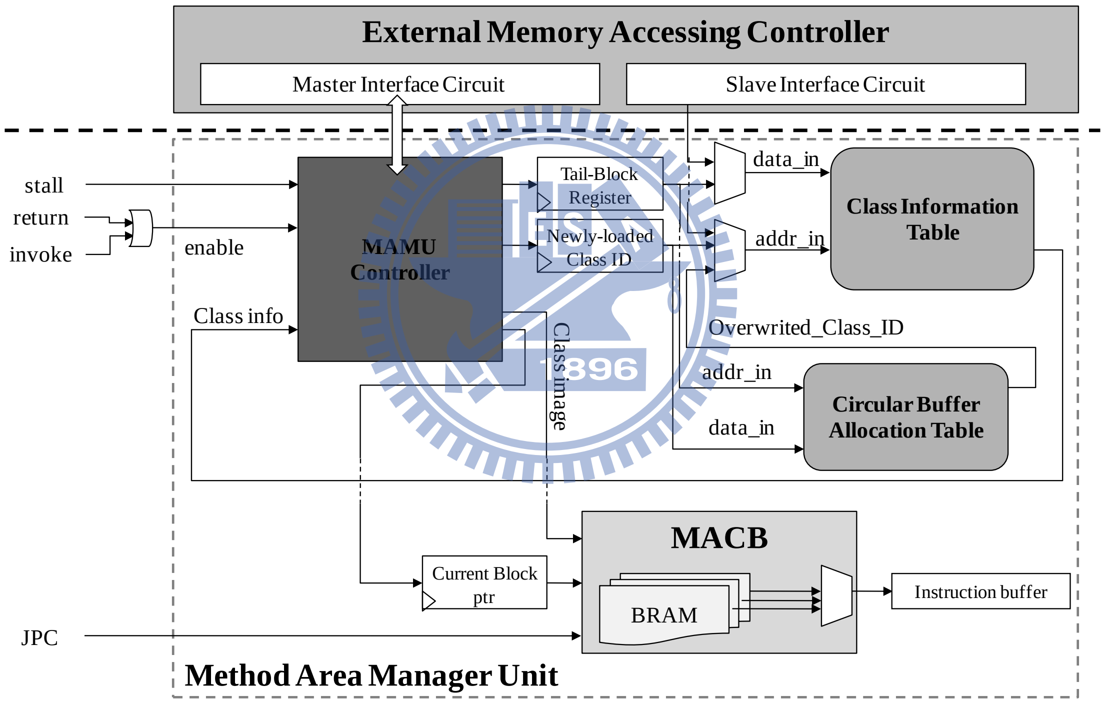


Fig. 7. Architecture Diagram of MAMU.

We design two tables to check hit or miss of the MACB. The Class Information Table is indexed by the class ID, and it stores each class's location in the DDR-SDRAM, size and occupied the MACB memory block's ID. The Circular Buffer Allocation Table records the class ID in each MACB memory block. When the JAIP must switch to another class, the controller inside the MAMU will be activated. The controller state machine is shown in Fig. 8. The controller will obtain the initial

MACB memory block ID occupied by the target class from the Class Information Table. If the number is not 0xFFFF, it means that the class image has been loaded into MACB and the MAMU simply sets the current block pointer to the new one. If the number is 0xFFFF, then a cache miss occurs. The controller then gets the class's address and size, and loads the images from the DDR-SDRAM. The cache replacement policy used is the FIFO policy due to its simplicity and reasonable performance for embedded applications. When overwriting an MACB memory block, the controller renews the Class Information Table base on the Circular Buffer Allocation Table. The victim class is eliminated from the Circular Buffer Allocation Table as well. The flow chart is showing in Fig. 9. Note that a class runtime image might occupy one or several contiguous MACB memory blocks according to its size, so the Class Information Table and the current block pointer simply records the first MACB memory block's ID. The MACB will add the significant 5 bits of the JPC to the current block pointer to get the proper block number.

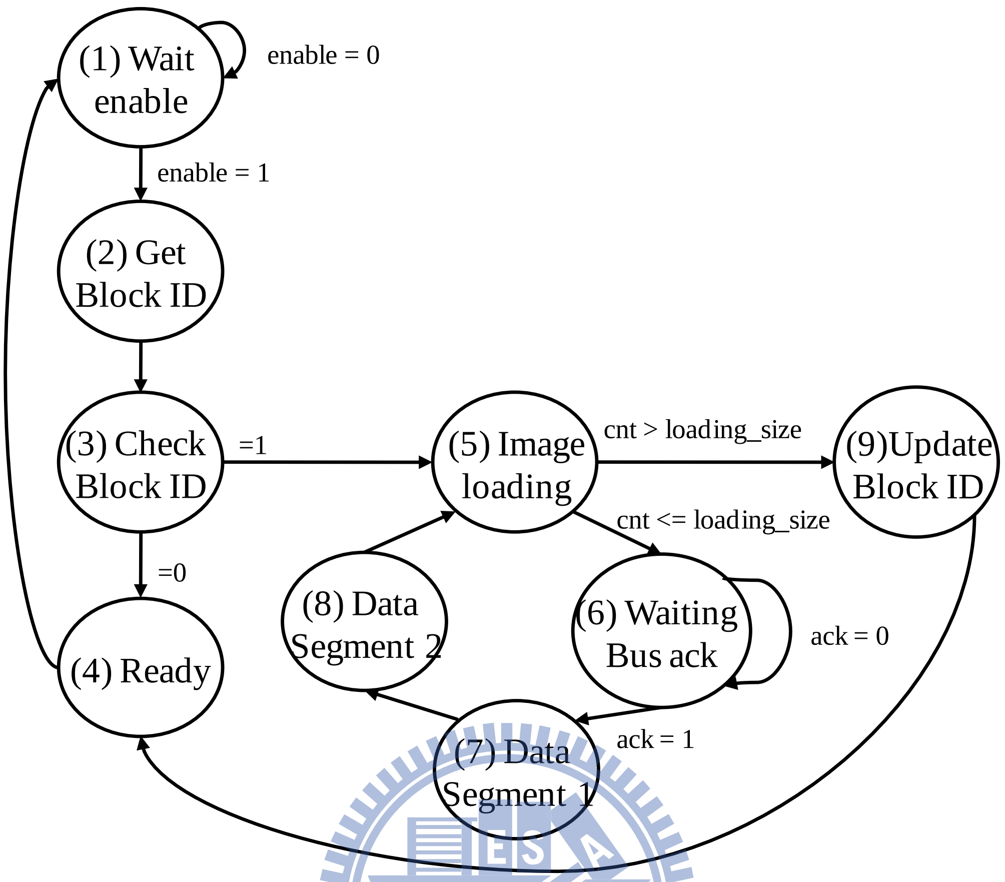


Fig. 8. The state machine of MAMU.

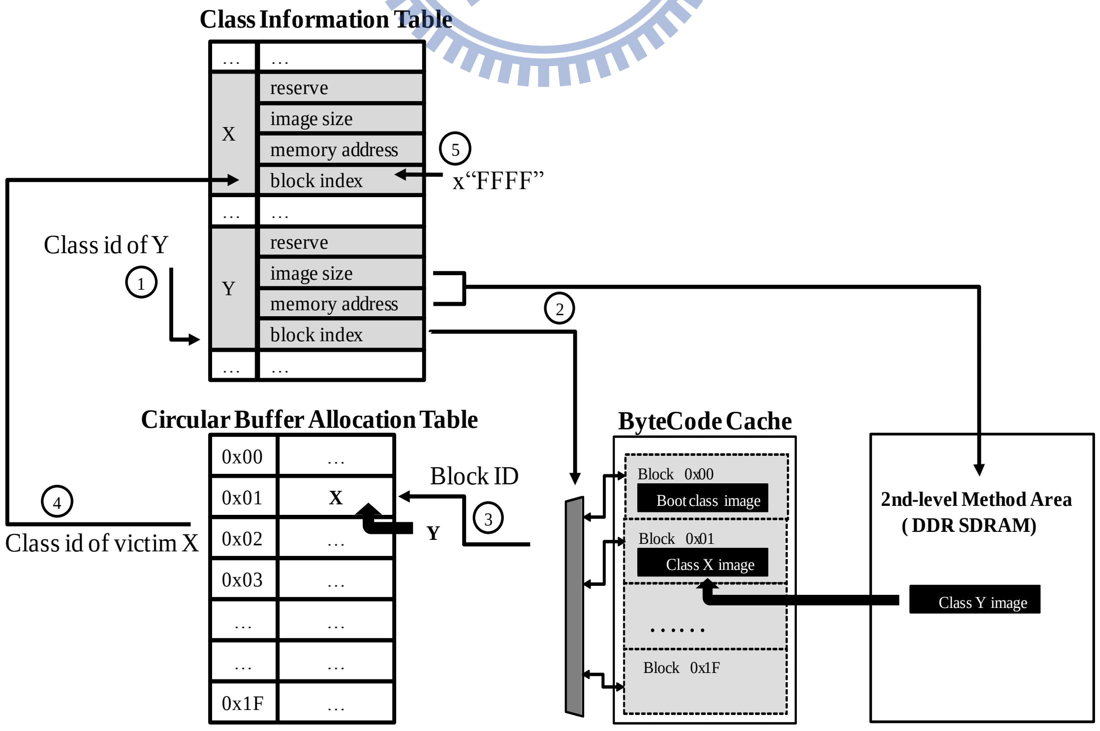


Fig. 9. The state machine of MAMU.

3.2.3 Instruction buffer

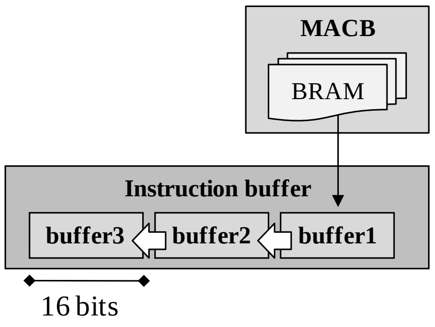


Fig. 10. The Instruction buffer controller.

Since Java bytecodes are variable length instructions, and the longest instruction is 5-byte long, the total length of the instruction buffer must be longer than 40 bits. The JAIP's bytecode execution engine supports double-issue of instructions. Two instructions will be fetched from the instruction buffer per clock cycle. The minimal size of two instructions combined together is 16 bits. Therefore, we design an instruction buffer with the total length of 48 bits, and it is separated into 3 buffer cells, each of 16 bits. The buffer controller fetches 16 bits bytecode (possibly includes operands) from the MACB each cycle renewing the buffer cell 1, and shifting the whole buffer by 16 bits (one buffer cell long). (Fig. 10)

3.3 Bytecode Execution Engine

This section will introduce the kernel of the JAIP -- Bytecode Execution Engine (BEE). The BEE is based on the double-issue four-stage pipeline micro-architecture proposed by Ko [2]. The overall block diagram of it is shown in Fig. 11.

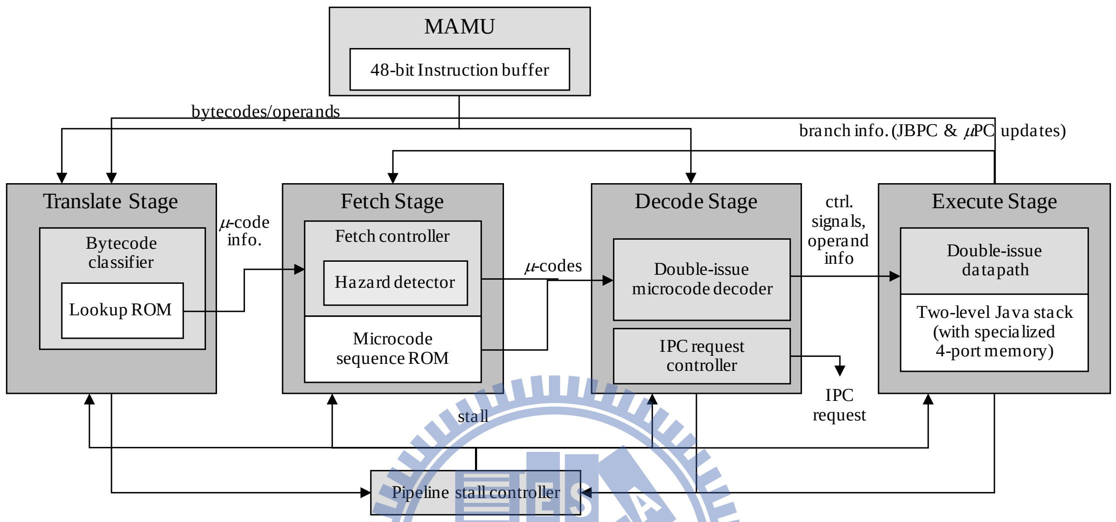


Fig. 11. The pipeline architecture of the bytecode execution engine.

The BEE executes all the method bytecode of Java. However, some of the Java bytecode instructions are quite complex, and their hard-wired implementation costs are quite high. Therefore, they have to be implemented using either a microcode sequencer or the execution of a sequence of hardwired instructions. Another technique for speeding up bytecode execution is instruction folding. The proposed BEE will try to fold two instructions into one on-the-fly. That is, it is, in essence, a simplified double-issue architecture with a single ALU. We also design an instruction set architecture for simplifying the circuit complexity. We classify the Java instruction byte sequence into three types, Simple Instructions, Complex Instructions and Operands. Most of Java bytecodes are mapped, one-to-one, to the native BEE instructions, which are referred to as the Simple Instructions. For the second case, the BEE translates a complex Java bytecode (e.g. invoke) into a predesigned microcode sequence and such bytecodes are called Complex Instructions. Java instruction byte sequence may also contain some operand data following a operation bytecode, and those operand data bytes are classified as Operand type.

3.3.1 Translate Stage

This section describes the first stage of the proposed pipeline architecture. The Translate Stage merely translates the bytecodes inside the instruction buffer into the microcode information at every clock cycle. The task of Translate Stage is quite simple but essential. Because of the variable-length instruction nature of the Java bytecode instructions, it is not trivial to fetch two bytecode instructions per cycle. Therefore, Java bytecode must be decoded to some degree before the Fetch Stage. There are two advantages of performing such early partial decoding of bytecodes: 1. The Fetch Stage following the Translate Stage would know how many bytes it has to fetch in order to execute a complete instruction with operands. 2. The circuitry for the hazard detection unit, operand number controller, etc., can be simplified.

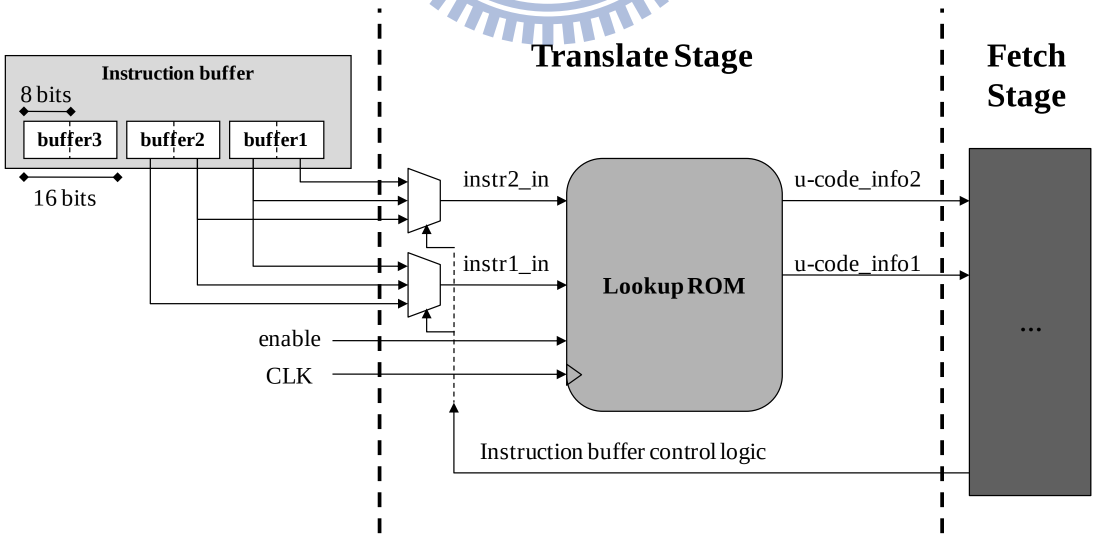


Fig. 12. Architecture Diagram of Translate Stage.

The translation is implemented using a ROM, addressed by the numerical values of the Java bytecode instructions. Each entry inside the lookup ROM includes three pieces of information: the number of Operands behind the OP code (4bits), the IsComplex bit (1bit) and a mapping information (8bit). The mapping information presents different meanings depending on the instruction type. If the incoming byte is a Simple Instruction, the mapping information will be a single microcode instruction which passed directly to the Fetch Stage. If the incoming byte is a Complex Instruction, the mapping information will be a starting address that points to the corresponding microcode sequence in the microcode ROM. Despite the fact that an operand data byte does not need to be translated, the translate stage will still map it to either a microcode instruction or a microcode ROM address. However, such mistakes will be resolved at the Fetch Stage and the corresponding microcode information will simply be discarded. The operand values will be extracted directly from the instruction buffer at the Decode Stage and the Execute Stage.

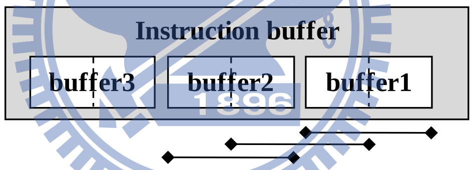


Fig. 13. Possible inputs to the translation logic and the decode logic.

The translation logic reads 16-bit of data from the instruction buffer for mapping operations. There are three possible sources for the 16-bit input data as shown in Fig. 13. The 16-bit input data can come from buffer one alone, from the high byte of buffer one and the low byte of buffer two, or from buffer two alone.

3.3.2 Fetch Stage

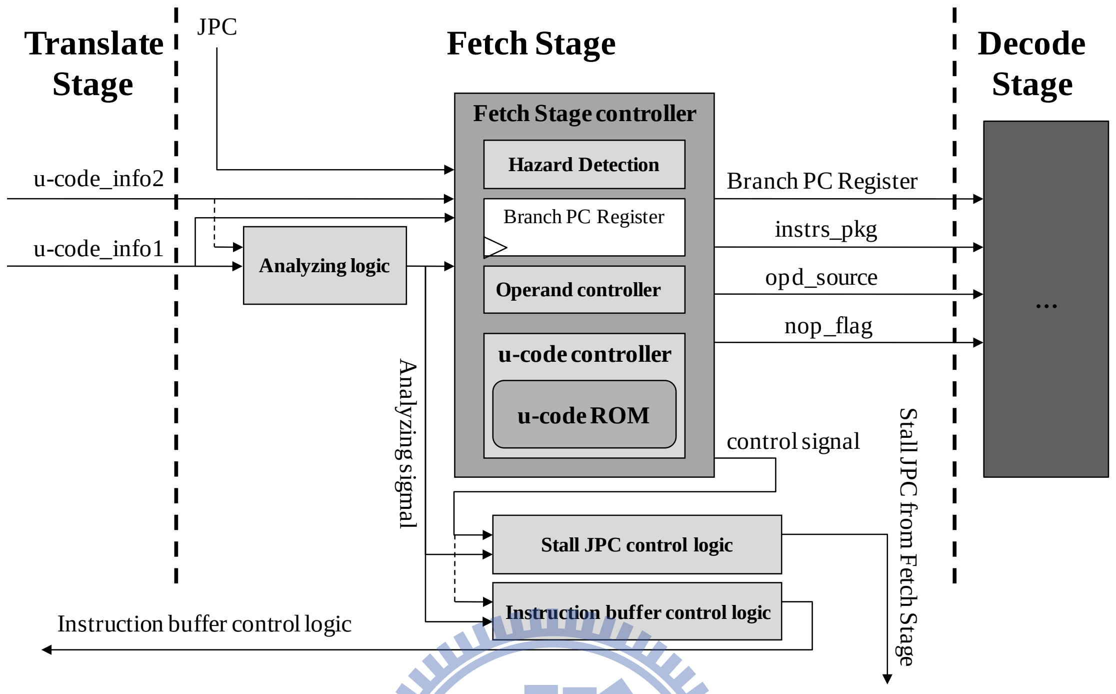


Fig. 14. Architecture Diagram of Fetch Stage.

The Fetch Stage is a complex circuit, and Fig. 14 shows its top-level architecture diagram. The primary tasks of Fetch Stage are as follows:

1. Sending two executable microcode instructions to the decode stage per clock cycle.

2. Determining the location of the input data in the instruction buffer.

3. Recording the program counter when encountering a branch-like instruction.

Based on the microcode information passed by the Translate Stage, the Fetch Stage analyses the type of the two translated instructions. If the two instructions do not belong to one of the cases shown in Table 1, a double-issue hazard may occur. For the cases shown in Table 1, the Fetch Stage checks and see if each instruction is of the operand type or complex type. If the instruction is of the operand type, it will be replaced by a 'NOP' instruction. If it is of the complex type, the Fetch Stage will switch to microcode sequence fetching mode.

1st Instr.	**2nd Instr.**	**Output instr. combination**
S	**S**	1stInstr. + 2ndInstr.
S	**O**	1stInstr. + nop
O	**S**	nop + 2ndInstr.
O	**O**	nop + nop.
C	**O**	complex mode
O	**C**	

Table 1. Possible translated instruction combinations after the Translate Stage. Note that S stands for Simple type, C for Complex and O for Operand.

If the two instructions received from the Translate Stage cannot be handled in one cycle (i.e. a pipeline hazard occurs), the second instruction has to be stalled until next cycles. There are two kinds of hazardous cases:

1. There is a structure hazard between two Simple Instructions.
2. One of the two instructions is of the complex type, and the other one is not of the operand type.

1st Instr.	**2nd Instr.**	**Output instr. combination**	
		Current cycle	**Next cycle**
S	**S**	1stInstr. + nop	2ndInstr. + 3rdInstr.
S	**C**		
C	**S**	complex mode	2ndInstr. + 3rdInstr.
C	**C**		

Table 2. The translation when hazard occurs. Note that S stands for Simple type and C for Complex.

The situation will be resolved by applying another instruction translation phase using the table shown in Table 2. The first instruction will be sent to Decode Stage with a ‘NOP’ instruction if it is a simple instruction. If it is a complex instruction, the Fetch Stage will switch to complex mode and fetches two simple instructions from the microcode sequence ROM of the complex instruction. For the second hazardous instruction received directly from the Translate Stage, it will be saved in a register and combined in next cycle with another instruction (the 3rd instruction shown in Fig. 3.3-6) obtained from the Translate Stage. The combination of the 2nd and 3rd instructions will go through the same hazard removal process in next cycle.

There are two modes in the Fetch Stage, complex mode and normal mode. The Fetch Stage Usually is in normal mode. In this mode, The Fetch Stage passes the two microcodes, which are obtained directly from the translate stage, to the Decode stage. When the Fetch Stage encounters the situation where the first instruction is of the complex type or the two instruction bytes are a operand byte followed by a complex instruction, it switches to the complex mode. The Fetch Stage will stall JPC and instruction buffer properly base on the number of operands so that the following operands will be store in buffer completely. Because one of the instructions is of the complex type, the corresponding mapping information will be a microcode ROM address. The Fetch Stage sends two microcodes fetched from the microcode sequence ROM at this address to the Decode stage. The following fetch operations will continues until all the microcodes of the corresponding complex instruction are fetched. Currently, we do not allow mixed double-issue of a simple instruction microcode and a complex instruction microcode. Note that the hazard detection logic only processes the two instructions passed from the Translate Stage. Therefore, the programmer of the microcode ROM should make sure that there is no hazard situation for every consecutive pair of microcodes in the ROM. The programmer should insert proper ‘NOP’ instructions whenever necessary.

3.3.3 Decode Stage

When the Fetch Stage sends two non-hazardous microcode instructions to the Decode Stage and the Decode Stage then generates the corresponding control signals and flags. The Decode Stage consists of the operand source MUX, the special-instruction decode unit, the normal-instruction decode unit, the flag manager, and the branch destination calculation unit as illustrated in Fig. 15.

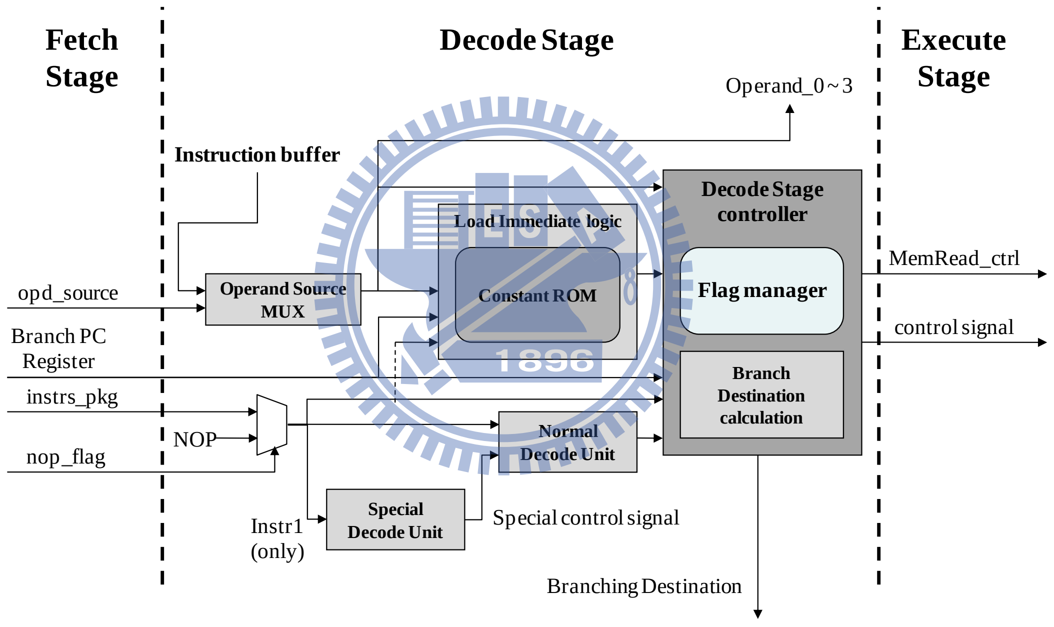


Fig. 15. Architecture Diagram of Fetch Stage.

If a decoded instruction requires operands, the operands’ location in the instruction buffer can be determined using the following policy. If the instruction is currently stored in the upper byte of buffer cell 3, the corresponding operands will follow in order. The sequence of operands is described as following: lower byte of buffer cell 3 – opd0, upper byte of buffer cell 2 – opd1, lower byte of buffer cell 2 – opd3 and upper byte of buffer cell 1 – opd3. Fig. 16 shows all possible locations of each operand.

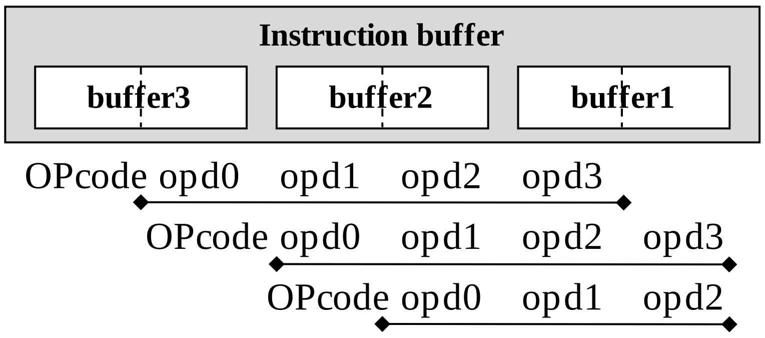


Fig. 16. Operand positions in the instruction buffer.

Most of the instructions are decoded by the normal-instruction decode unit, but some of them are decoded by the special-instruction decode unit because of their sophisticated behavior. For example, the branch instructions may change the JPC conditionally. More details of special instructions will be described in Appendix A. The flag manager raises the flag according to the instruction to enable other circuits or mark a situation. The branch destination calculation is performed in the Decode Stage. The calculation usually involves the operands and the stack data, so performing the branch destination calculation in Decode Stage is reasonable.

Due to the nature of on-chip static RAM, the read data will be available in the output port one cycle after the addresses are generated. In order to prepare all the data before the Execute Stage, the preloading technique is adopted. The memory reading control signal is generated in the Decode Stage and associated directly to the address port of the stack RAM in the Execute Stage. In the next cycle, the calculation is executed without any memory delay because the value in the stack RAM has already been extracted. Other signals, such as data path control and store control are registered as the traditional pipeline design.

3.3.4 Execute Stage

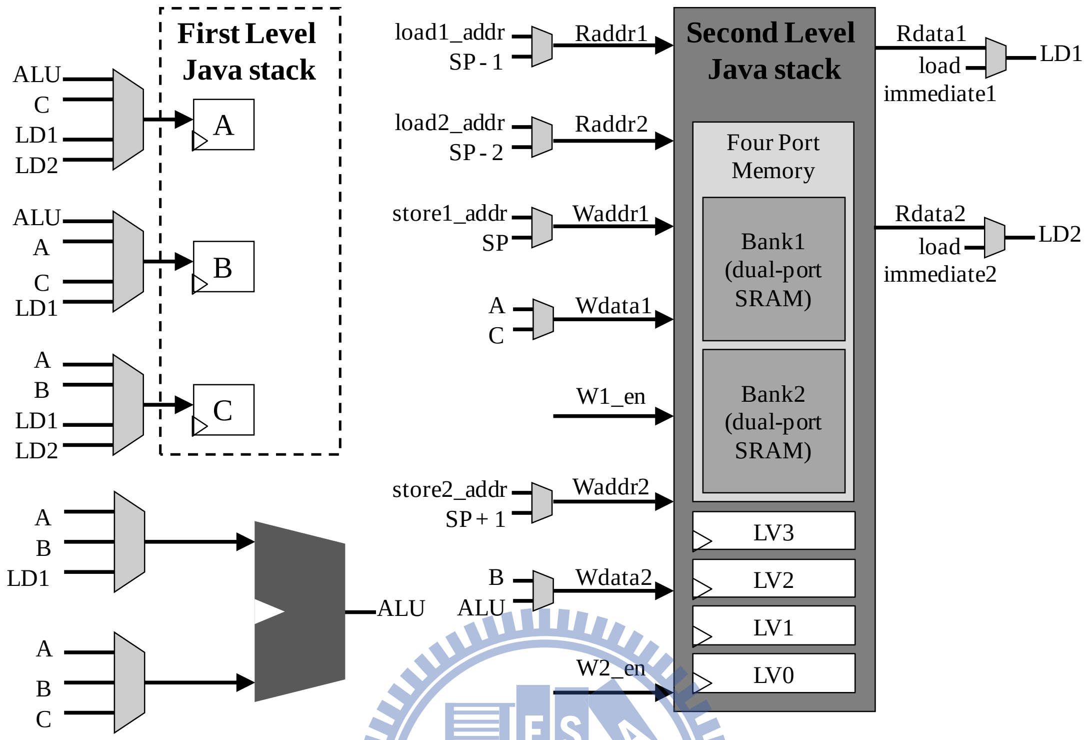


Fig. 17. Data Path of Execute Stage. Note that LD represents the load data from stack memory or immediate data. The capital R stands for read and W for write. The LV represents local variable register.

The Execute Stage just performs the control signal from the Decode Stage accurately. The data path of it is shown in Fig. 17. Since the Java virtual machine is stack machine, the implementation of stack and stack operation is very critical. To achieving double issue data path, we design a special two-level stack architecture. This stack structure consists of 7 registers and a four-port memory as shown in the right part of Fig. 17. Note that storing Java stack directly in memory is different from the stack cache architecture which is popular for Java processors. It is necessary to update both registers and the stack memory every time the stack pointer is updated due to some stack operations.

1.3.1.1. Two-level Stack Structure

The ISA is classified into four functional types: load type, store type, ALU type, and special type. Except special type instructions and some ALU type instructions, all the instruction can be double-issued. Most of the combinations will merely use the top two of the stack elements, but the combination of store and ALU will need three stack items (the store stores the first item of stack and ALU needs the second and the third). In short, it needs at least three items from stack to perform all the combination. Therefore, there are three register store the top three of stack called A, B, and C respectively, and there are also the first level of the Java stack. Fig. 18 illustrates how each type of the two-instruction combination can be resolved by such design. There are two points worth attention. The first one is that the combination of two ALU operations is not supported because the latency will be too long if adding an extra ALU and the probability of such combination is very low. And the other is that the special type has its own data path for each instruction.

The second level of Java stack is composed of a four-port memory and four local variable registers. The four instruction type will affect the stack respectively. The load type adds an item to stack. The store type consumes an item from top of the stack. The ALU type consumes two items from the stack and adds a new item to the stack. For the special instruction, it is not combined with other instruction. Therefore, any instruction combinations will update two top-of-the-stack items at most. Take the example of the combination of load and load instructions; it will load two local variables from stack memory. And it will also store two items into the operand stack of the stack memory. As a result, double-issue of two load instructions at the same cycle requires the stack memory to be a four-port memory.

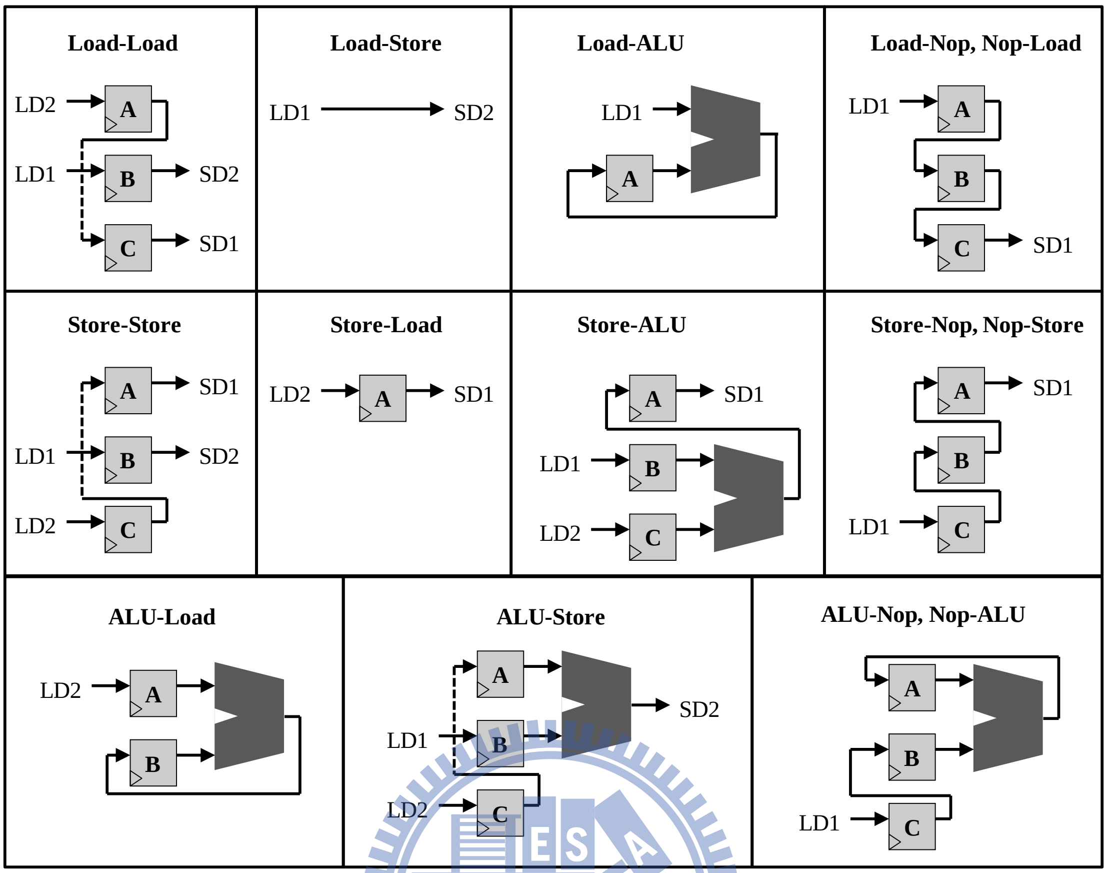


Fig. 18. Data Path Combination. Note that LD represents the load data from stack memory or immediate data, and SD represents the data which will be stored in stack memory.

A general-purpose four-port memory is expensive and not a common feature in current VLSI target technologies. The four-port memory we used in the design of JAIP is a special-purpose four-port memory. It is constructed by using two dual-port on-chip memory blocks organized in an interleaving structure. It has two read ports and two write ports. Therefore, if the decode and the execute stages have to open three read/write ports of the same memory bank at the same cycle, a pipeline hazard could happen. To solve such structure hazard due to memory-port limitation, we have added a 4-register local variable cache to the two-level stack architecture. The local variable registers will cache the first four local variables of the current stack frame. In Java bytecode, the load or store of local variable 0 to 3 is a single OP code without any operand behind, and the others are a combination of OP code and an operand. If the double-issue instruction combination is two single loads for example, the operation will be loading two data from the local variable register, and store the second and third items to each memory bank respectively. If the instruction combination is an OP code with an operand, the operation will simply be one load and one store. That is, with this design, accesses to the most frequent local variables (based on the Java VM model) will not cause structure hazard to the proposed double-issue data path. Note that the initialization of the local variable cache for current stack frame happens upon method invocation, and restoring operation occurs in return.

3.4 Inter-processor Communication Unit

The communication between the JAIP and the RISC core is achieved using two mechanisms. The first one is a memory sharing table called the cross reference table, and it will be described in chapter 4. The second one is an interrupt-driven mailbox device for low-bandwidth control data exchange. Mailbox is a common inter-processor communication (IPC) device for SoC with multiple processor cores.

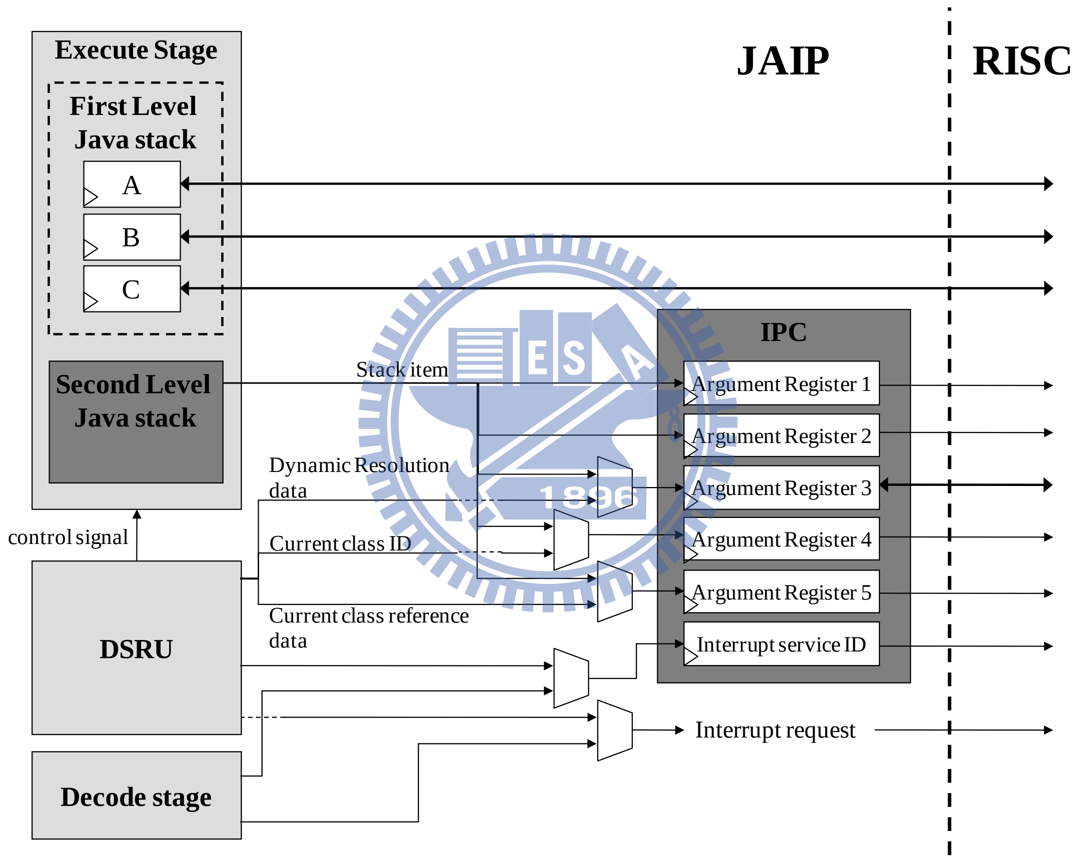


Fig. 19. Architecture Diagram of IPC Note that the single arrow stands for one-way data flow direction, and the double arrows stands for the register can be read/written by both sides.

Since it is not efficient to run certain tasks, such as I/O and class parsing, on a stack-based machine, the JAIP will pass these operations to the RISC core by requesting RISC services through ISRs (interrupt service routines). Whenever the JAIP wants to request some services from the RISC core, it will use IPC for argument passing. The architecture of the mailbox IPC module is shown in Fig. 19. The register file of the mailbox is composed of five argument registers and a service ID register. The argument registers are copies of the first five of arguments stored in the stack memory. Note that the argument register number five can be read or written by both sides because some service will return a value from the RISC core (e.g. The service for the bytecode "new" returns an object reference address). The service ID register tells the RISC which ISR services it should provide, and it is maintained by the Decode Stage (for local branches) or DSRU (for requesting parsing operation), and the two circuits also raise interrupt signal. If a complex Java bytecode instruction requires a RISC-side service routine to perform some tasks (e.g. a "new" operation), the bytecode instruction will be implemented using a microcode sequence that involves writing the service ID register in the Decode Unit. The interrupt signal is triggered by the 1st instr., and the interrupt function ID is the 2nd instr. The DSRU triggered the interrupt signal only for requesting parsing operation, so the interrupt function ID is not changed. Note that only the DSRU fills the mailbox register. If the requesting service is a parse-load operation, the DSRU will fill directly the register itself. If not, the DSRU will control the stack memory in Execute Stage to fill the argument registers with the first five arguments.

Chapter 4. Design of Dynamic Symbol Resolution Mechanism

4.1 Runtime Environment

We have described in detail the hardware design of the propose JAIP. In this chapter, we present how the propose Java accelerator IP (JAIP) performs the dynamic symbol resolution and how dynamic resolution is accomplished jointly by the JAIP and the RISC core. A complete Java runtime environment (JRE) is a sophisticated system. There are many key components such as bytecode execution engine, class loaders, the class libraries, and so on. If all the key components are implemented in software, the integration of Java virtual machine with the rest of the software components is simple. Before a class file can be used by a Java application, it must go through a process of loading, linking, and initialization [7]. It is not trivial to implement a hard-wired Java processor core to perform such a complex process. Therefore, for a hardware-assisted JRE, only the bytecode execution engine can be reasonably implemented in hardware, and that is the approach taken by the JAIP. To construct a complete JRE, we design a series of system-level services running on the host RISC processor, which is customized to handle the loading and linking process, heap memory management, and I/O operations. These services are implemented as interrupt service routines (ISRs). When the JAIP requires some services, an interrupt signal will be sent to the RISC core through IPC to enable the corresponding services.

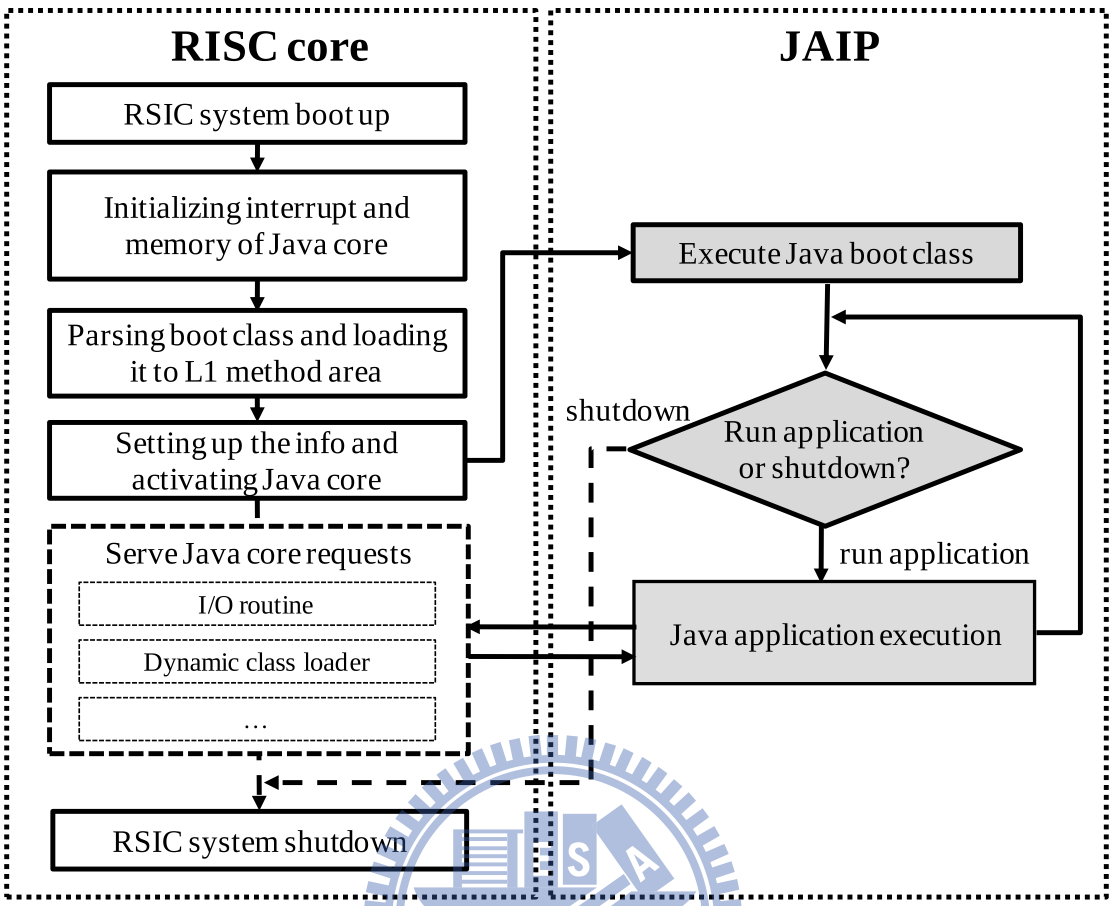

RISC core
RSIC system boot up
Initializing interrupt and memory of Java core
Parsing boot class and loading it to L1 method area
Setting up the info and activating Java core
Serve Java core requests
I/O routine
Dynamic class loader
…
RSIC system shutdown
JAIP
Execute Java boot class
shutdown
Run application or shutdown?
run application
Java application execution

Fig. 20. **The life cycle of proposed Dual-Core Java SoC.**

Fig. 20 shows the system life cycle. At the beginning of the system, a system initialization process will be run on the RISC core. The initialization process consists of three steps. It will register the interrupt table and initialize the interrupt handling routine first. The heap memory and the L2 method area are also initialized. In the second step, the RISC core will try to locate the jar file which contains all the Java classes in the mass storage (a Compact Flash card in our implementation). The boot class in the jar file will be loaded first. Once the boot class is found, the RISC core will then generate the run time image of the boot class and store the run time image in both L1 and L2 method area. The final step of the initialization process is to set up the essential information (e.g. the initial signals of the JAIP registers: INIT_PC, INIT_SP, and LVreg_valid, etc.) and enables the Java core to execute the boot class of the Java program. During the execution of a Java application, the JAIP can access both the heap memory and the L2 method area and call ISRs if necessary. After the Java application terminates, the JAIP would set a status register bit to inform the RISC core to shutdown the whole system.

4.2 Dynamic Class Loading Mechanism

This section describes the proposed solution for dynamic class loading which is a difficult issue for a hardware-based Java accelerator. As mentioned in section 4.1, a class must go through sequential processes of loading, linking, and initialization before it can be referenced by other classes. The Java virtual machine specification defines that the three processes must taken place in order. For typical computer languages, the linking process also resolves all symbols in the object files to physical addresses. However, for the Java language, some of the symbols cannot be statically resolved during the linking process. This is called dynamic resolution. That is, on the class's first active use, it must be initialized. Before it can be initialized, it must be linked. And before it can be linked, it must be loaded. Note that it is not necessary to wait until the class's first active use to load and link that class; the class can be loaded and linked well ahead of time of its first usage [7].

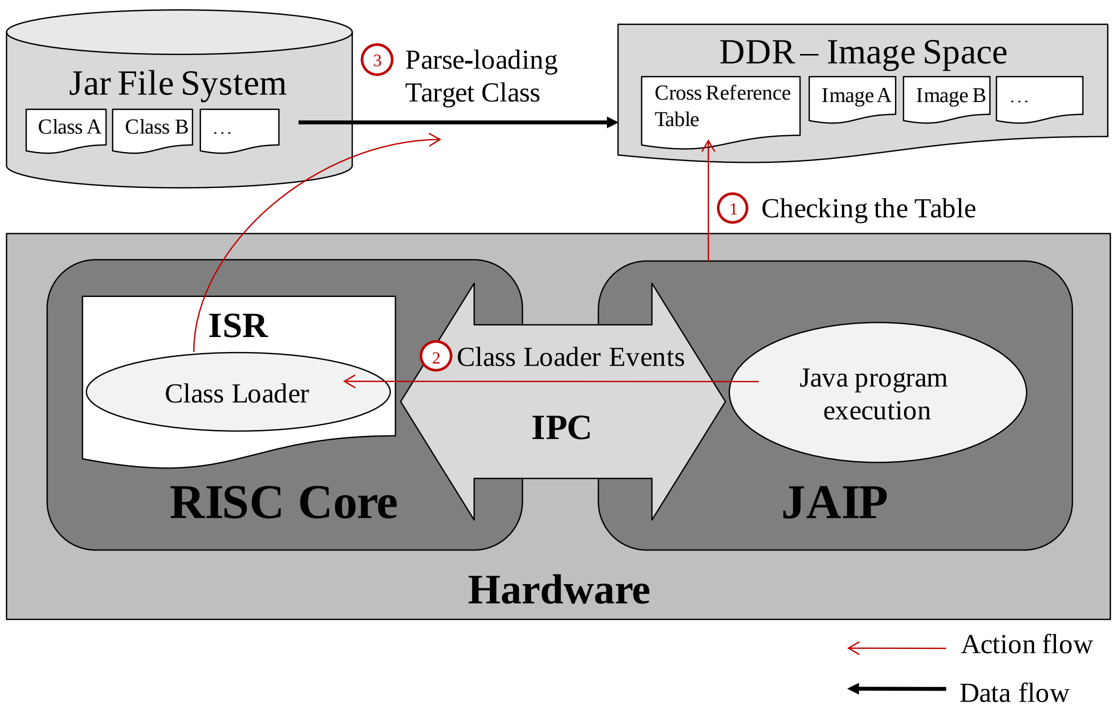


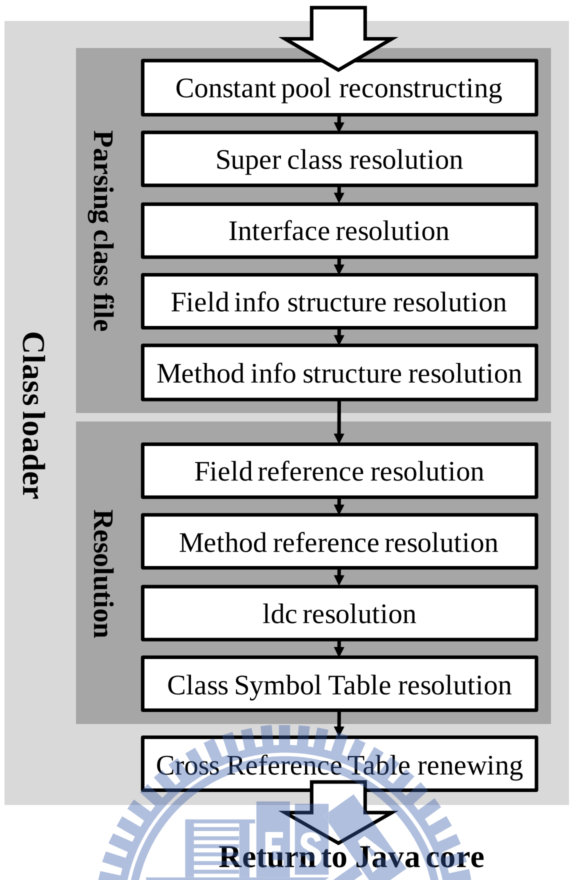
Fig. 21. Runtime dynamic class loading mechanism.

Although there are some implementation flexibility in the timing of loading and linking. JAIP define that the processes of the three must take place in strict order on the class's first active use since this policy consumes less memory and is more suitable for embedded applications. We implement the proposed dynamic class loading and symbol resolution mechanism with hardware-software co-designed approach. Dynamic class loading is performed on the RISC core using software routines as shown in Fig. 21. We merge the loading and linking process into a customized system class loader as shown in Fig. 22. There are two types of resolution policies in Java VM implementations: early and late resolutions. We adopt late resolution for JAIP since this approach requires smaller memory footprint and lower (and smoother) overall class loading overhead. Therefore, the system class loader also includes the resolution process. The system class loader (executed by the RISC core) will parse the class files and collect the runtime information from the class's constant pool and the Cross Reference Table. The software then generates a corresponding class runtime image with some reference information and renews the Cross Reference Table. The format of class runtime image and the detail of Cross Reference Table will be described in the following paragraphs.

Ideally, the class loading operations will take place on all the classes' first active use. And, according to the Java specification, there are six situations [7] that qualify as the active use of a class:

1. A new instance of a class is created.
2. The invocation of a static method declared by a class.
3. The use or assignment of a static field declared by a class or interface, except for static fields that are final and initialized by a compile-time constant expression.
4. The invocation of certain reflective methods in the Java API, such as the methods in class Class or in classes defined in the java.lang.reflect package.
5. The initialization of a subclass of a class.
6. The designation of a class as the initial class (with the main()< method) when a Java virtual machine starts up.

Note that the static field and part of the Java API are not supported in the current JAIP, and the class loading operation will be executed when condition 1, 2, 5, and 6 occurs.



Class loader
Parsing class file
Constant pool reconstructing
Super class resolution
Interface resolution
Field info structure resolution
Method info structure resolution
Resolution
Field reference resolution
Method reference resolution
ldc resolution
Class Symbol Table resolution
Cross Reference Table renewing
Return to Java core

Fig. 22. The flow diagram of the customized system class loader.

Dynamic symbol resolution is executed completely by the hardware. The JAIP can resolve all runtime information with the reference information in class runtime image by the Dynamic Symbol Resolution Unit (DSRU). The state machine of DSRU is shown in Fig. 22. Furthermore, the first step of this mechanism is a one-time software execution overhead for each class throughout the life cycle of the JRE. When a class is invoked for the second time, the JAIP will simply load the previously created class runtime image from the L2 method area without trigger any software overhead. The hardware detail of the DSRU's state machine will be presented in sections 4.3 and 4.4.

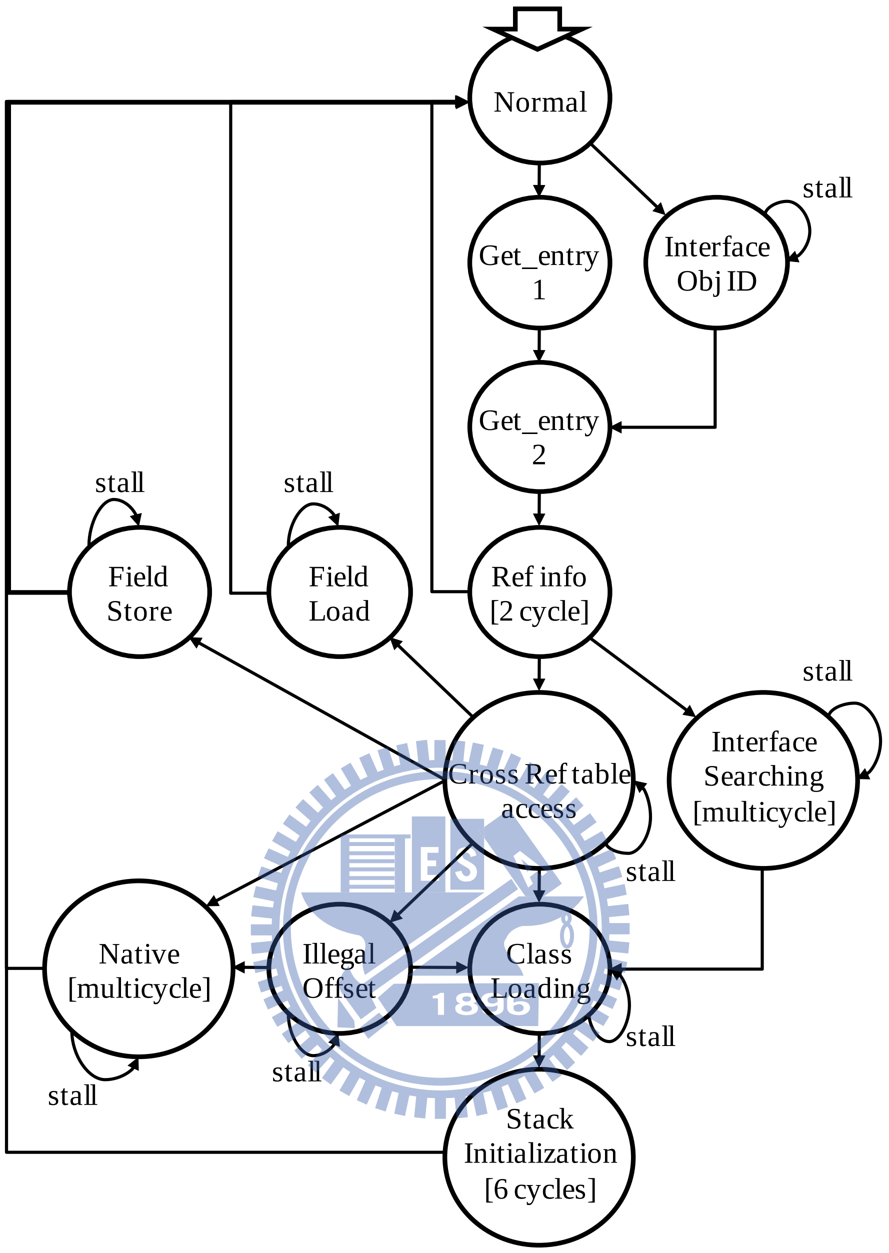


Fig. 22. The overall state machine of DSRU.

Although the process of loading and linking is carried out by the system class loader software, the initialization is not. There are two types of initializations for a class: the first one is static field initialization, which is not supported by the JAIP currently, and the other one is non-static object field initialization. The non-static object field‘s initialization is simply triggered by Java invocation bytecode (*invokespecial*), so it can be perform in JAIP perfectly. The initialization of static field is more complicated and hence we leave static field initialization for future work. This issue will be discussed in chapter 6.

4.2.1 Cross Reference Table

The Cross Reference Table is the key of the whole proposed dynamic class loading mechanism since it records the detail symbol information for each class. The table helps not only the system class loader to collect the runtime information, but also gives helps to method invocation, field data access, ldc instruction, new object and string operation. The table can be accessed by both the JAIP and the RISC core. The entry of Cross Reference Table is shown in Table 3. This table has first been proposed in [6]. We improve the structure of the table in order to support more Java language features.

Cross reference table				
Class [x]	**Attributes**	Class ID , Class name, image address, etc		
	Parent's ID	Class ID		
	Object Size	Size (byte)		
	IsParsed	1 or 0		
	Interface Info	IsInterface (1 or 0)	Interface Count	Interface List
	Field Data[i]	Field Name	Class ID & Field Offset	Static Field Address
	Method[i]	Method Name	Class ID & Method Offset	
	Ldc[i]	Data	Type	
	String Pool	The whole string data stored in constant pool		
	String Pool Offset[i]	The offset for each string		

Table 3. Cross reference table structure. Constant data in the literal pool of the class file are also stored in this table.

When the system class loader recognizes a new class, it will allocate a new global class ID and creates a table space for the class. A basic table field includes class name, class path name, parent's ID, image start address of external memory, and image size. A class's table will be filled with new information in one of two situations. The first one is that the system class loader has just parsed the class. And the other one is that the class that is being parsed refers to an unparsed class. During the execution, the JAIP can directly access the expected information for usage. Sections 4.3 and 4.4 will describe how the JAIP use the information of the Cross Reference Table to do method invocations and field data accesses.

4.2.2 Class Runtime Image Format

The detail structure of the class runtime image of a single class is shown in Fig. 23. This image format can be divided into four parts. The first part contains the image header and it always is 0x4D4D4553 which is the ASCII codes of “mmes”. The second part is the Class Symbol Table. The entry size of this table is 16 bits, and each entry corresponds to a constant pool entry of the original class. Therefore, the number of entry is the same.

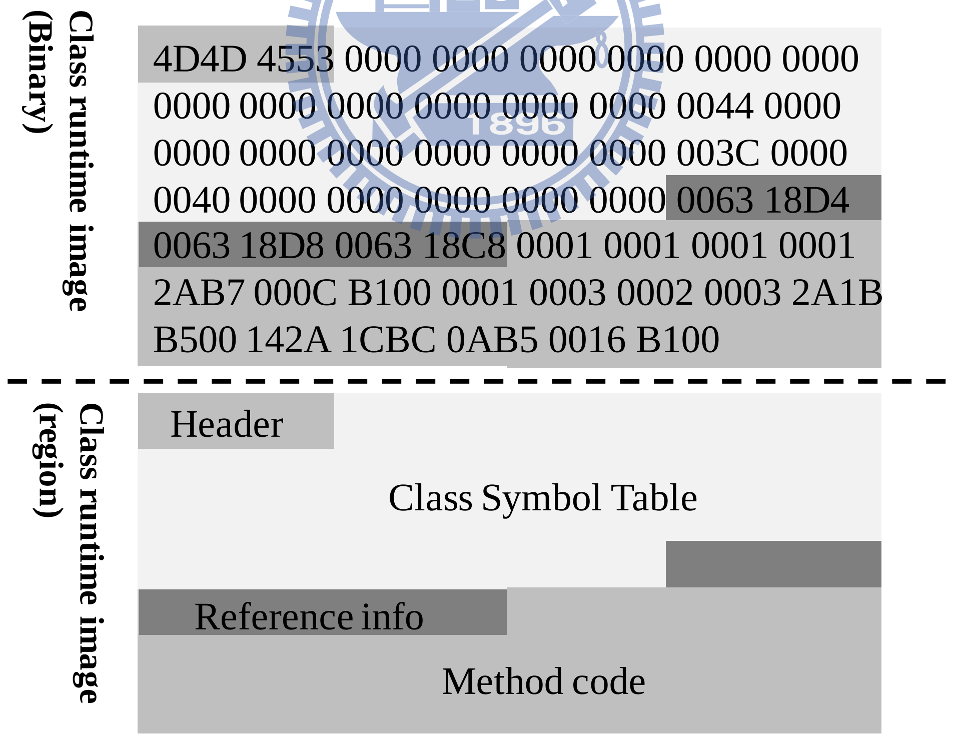
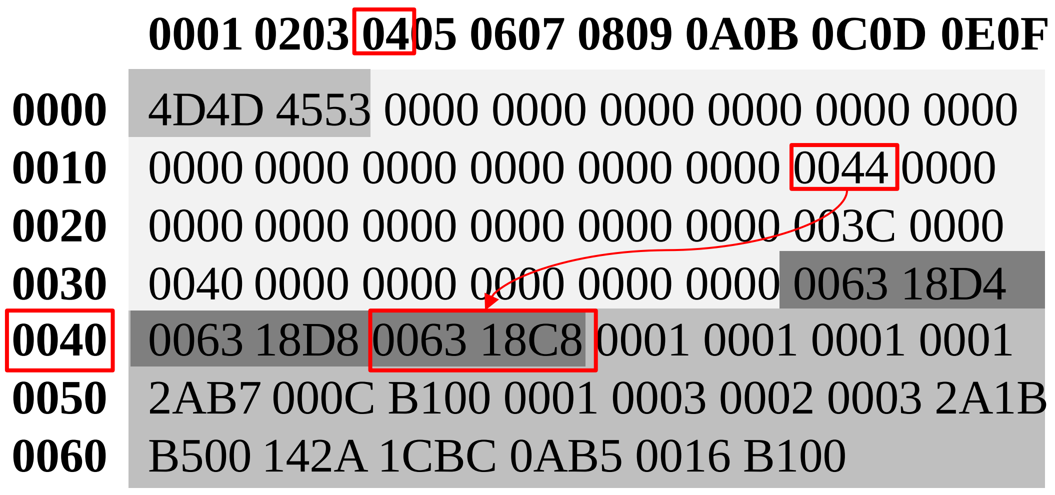

```
4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB7 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100
```

Fig. 23. The region diagram of Class Runtime Image format.

Most of the Class Symbol Table entry stores nothing because most data of the constant pool are reconstructed into the Cross Reference Table. However, if the entry's corresponding constant pool entry is of CONSTANT_Class type, CONSTANT_Fieldref_info type, CONSTANT_Methodref_info type, CONSTANT_InterfaceMethodref_info type, CONSTANT_String_info type, or CONSTANT_Integer_info type, the Class Symbol Table entry will store an offset to the image and indicates corresponding reference information below. Therefore, the DSRU can directly identify the target reference information. The content of reference information depends on the corresponding constant pool entry. If the constant pool entry is of CONSTANT_Fieldref_info type, CONSTANT_Methodref_info type, or CONSTANT_InterfaceMethodref_info type, the reference information will be a 32 bits memory address pointing to the Cross Reference Table entry. And, by indirect reference, the DSRU can obtain the resolution information. If the constant pool entry is of CONSTANT_Class type, CONSTANT_String_info type, or CONSTANT_Integer_info type, the reference information will be a 32-bit class information which will be sent to the RISC core as an argument. The dynamic reference flow is shown in Fig. 24. The last part is the method code data. The method bytecode, argument numbers, local variable numbers, and max stack numbers all are stored in this part.



```
      0001 0203 0405 0607 0809 0A0B 0C0D 0E0F
0000  4D4D 4553 0000 0000 0000 0000 0000 0000
0010  0000 0000 0000 0000 0000 0000 0044 0000
0020  0000 0000 0000 0000 0000 0000 003C 0000
0030  0040 0000 0000 0000 0000 0000 0063 18D4
0040  0063 18D8 0063 18C8 0001 0001 0001 0001
0050  2AB7 000C B100 0001 0003 0002 0003 2A1B
0060  B500 142A 1CBC 0AB5 0016 B100
```

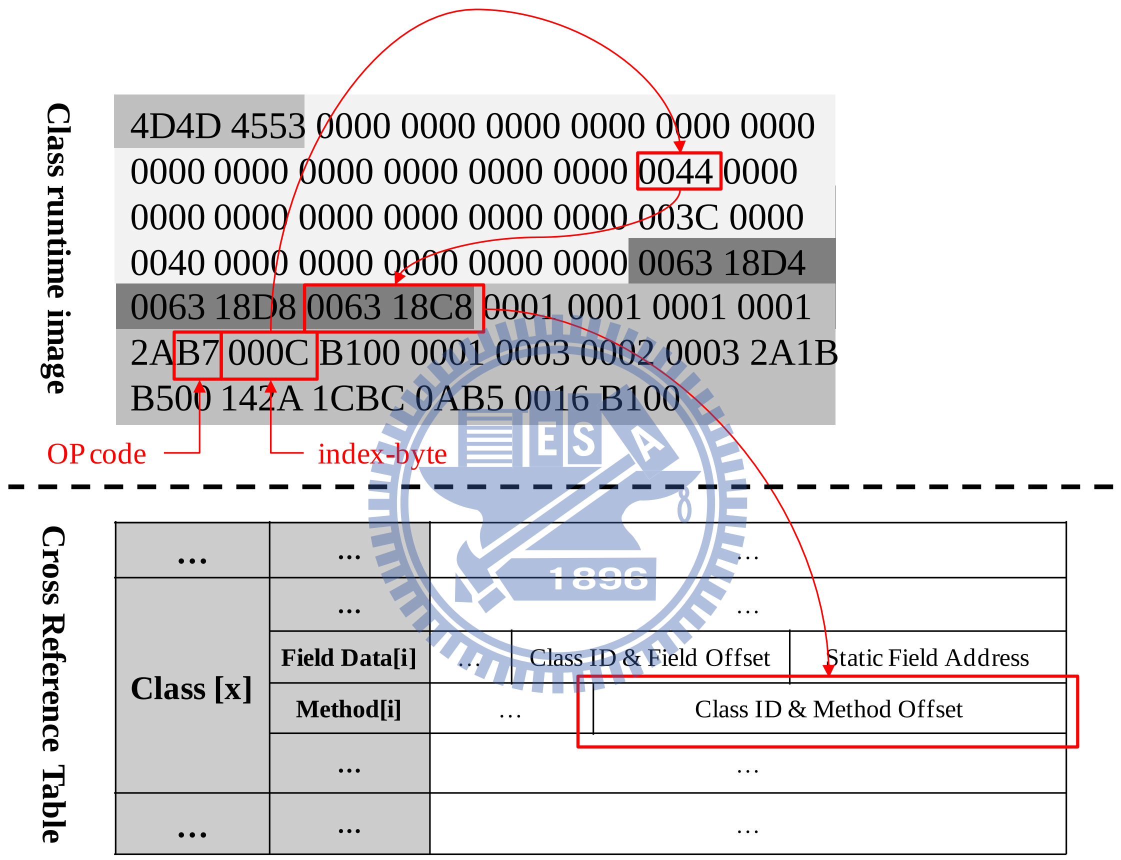
Fig. 24. The reference flow of Class Runtime Image.

4.3 Method Invocation Mechanism

This section introduces the method invocation mechanism of the JAIP. If a class method refers to another, the will be a method invocation instruction in the class method bytecode region. A method invocation bytecode is composed of an OP code and a 16 bits index-byte indicating a CONSTANT_Methodref_info or CONSTANT_InterfaceMethodref_info entry in the constant pool. The traditional Java virtue machine will distinguish the target class and method depending on the entry's data structure. There are four types of method invocation bytecodes, including invokestatic, invokeinterface, invokespecial, and invokevirtual. The invokestatic instruction refers to a class's static method directly. The other three depend on their object references. The invokeinterface instruction refers to an interface method only. The invokespecial instruction invokes a super-class method, private method or instance initializing method. And the invokevirtual instruction is used for a normal class method reference. The proposed JAIP provides three approaches to support these invocations. Although, the functions of the invocations are different, the JAIP executes them with similar procedures. Because JAIP's system class loader has reconstructed the constant pool and replaced it by the Class Symbol Table and the reference information. Therefore, all runtime method reference information is resolved by the Dynamic Symbol Resolution Unit (DSRU). It also is the second step of the proposed dynamic class loading mechanism.

Most of the invocations are classified into the normal method invocation, because the system class loader has completed most static symbol resolutions. As a result, the remaining operations of the invokestatic, invokespecial and invokevirtual instructions are the same. Nevertheless, the invokeinterface has a critical difference from the others, which will be explained in section 4.3.2. Another method invocation challenge is that Java supports calling C code from Java class method called native invocation, and many system class implementations also require native method invocations. The details of the native method support will be described in section 4.3.3.

4.3.1 Normal Method Invocation



```
4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB7 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100
```

…	…	…		
Class [x]	…	…		
	Field Data[i]	…	Class ID & Field Offset	Static Field Address
	Method[i]	…	Class ID & Method Offset	
	…	…		
…	…	…		

Fig. 25. The reference flow of Normal Method Invocation.

After static symbol resolutions performed by the system class loader, most method invocations are normal method invocations. Therefore, optimization of normal invocations is crucial for the efficiency of Java application execution. The reference flow of normal method invocation is shown in Fig. 25. The method invocation bytecode is composed of an OP code and a 16-bit index, and the 16-bit index now indicates a Class Symbol Table entry. A 32-bit reference pointer in the

Class Symbol Table will be located. The reference pointer is a 32-bit memory address pointing to a Cross Reference Table entry. And, by indirect reference, the DSRU get the 32-bit resolution information which is a class's ID and a method offset. If the significant 8 bits of the resolution information are 0xFF, this method invocation is a native method invocation. If not, then the DSRU checks the method offset. If the method offset is zero, it means that the target class's runtime image has not been generated yet. Then the DSRU will trigger the system class loader through the IPC. After the parse-loading is finished, the system class loader will return the correct method offset to the DSRU. No matter where the DSRU receives the method offset, it will get into next state, and check whether the target class is cached by the MACB or not. The MAMU will load the target class if necessary. After renewing the SP, VP and IPC, the whole invocation completes.

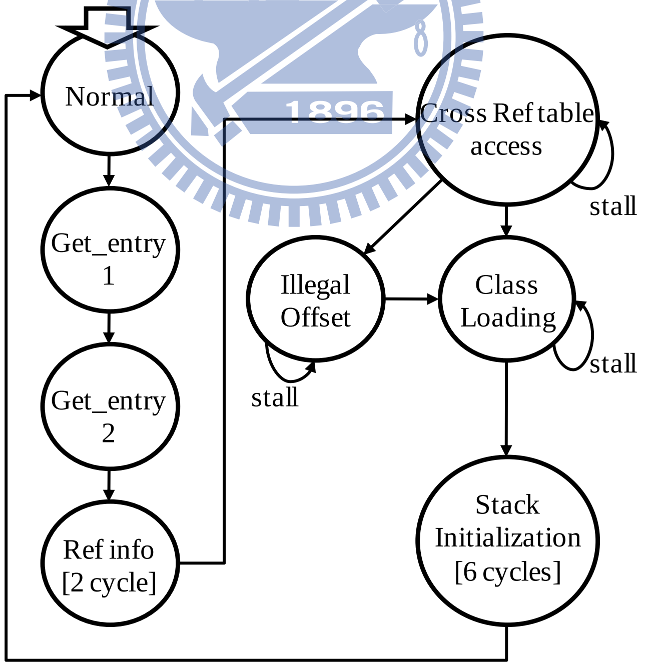


Fig. 26. The Normal Method Invocation state machine of DSRU.

The Fig. 26 describes the state machine of the DRSU during normal method invocation. The first two stages are the Get_entry1 and Get_entry2, which locates the Class Symbol Table entry. The Ref info stage then loads the reference information.

The Cross Ref table access stage access the Cross Reference Table entry. If the method offset is zero, the stage machine will enter the Illegal Offset and trigger the system class loader. If the method offset is non-zero or the parse-loading is finished, the state machine then enters the class loading stage and loads the target class image if necessary. The final stage is Stack Initialization, and the stage renews the SP, VP, and IPC.

4.3.2 Interface Method Invocation

The main difference between an interface method invocation and others is that the target class cannot be identified until runtime. The traditional Java virtue machine will distinguish the target class and method depending on two pieces of information. The first one is the CONSTANT_InterfaceMethodref_info entry's data structure, and the other one is the object instance. The JAIP's class loader still can obtain the method name of the interface from the constant pool. However, the object instance is established during runtime. Obviously, the system class loader cannot complete symbol resolution and locate the physical address of the method. Therefore, we design an interface method linking list to resolve this problem. The interface method linking list is shown in Fig. 27.

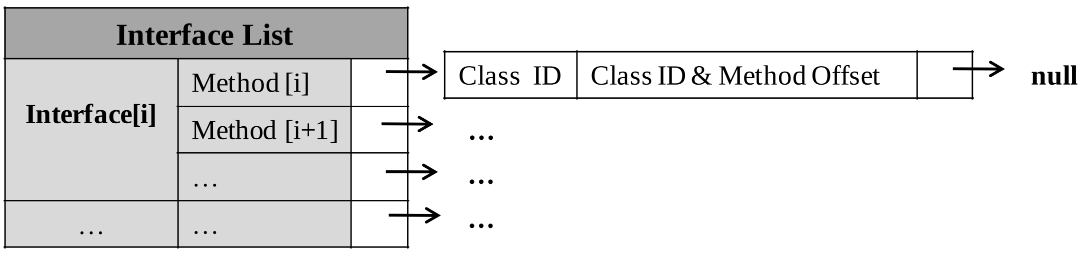
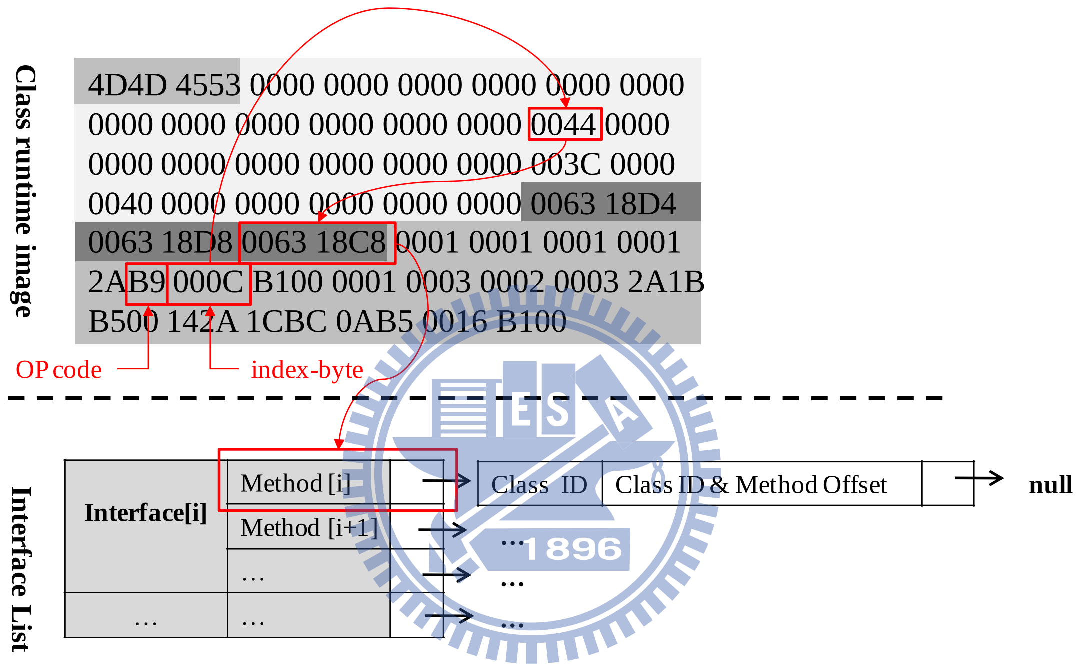


Fig. 27. The Interface Method Linking List format.

The Interface list is maintained by the class loader. When the class is first referred, the system class loader will analyze the class's constant pool. If the class implements any interface, the system class loader will create a list node for each method in the interface and fill out the node. The first class ID in a list node content is the class which implements the interface. The second class ID is the class which contains the physical implementation of the interface method. The JAIP can renew the JPC according to the method offset. The last entry is pointing to the next node of the link list.



Class runtime image
4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB9 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100
OP code
index-byte
Interface List
Interface[i]
Method [i]
Method [i+1]
…
Class ID
Class ID & Method Offset
null

Fig. 28. The reference flow of interface method invocation.

Similar to the normal method invocation, the interface invocation bytecode is composed of an OP code and a 16-bit index, and the 16-bit index now indicates a Class Symbol Table entry. The DSRU will recognize the object instance's class ID first, and then locate the 32 bits reference information. The reference information is a 32-bit memory address pointing to the first node of the interface list. After the two operations are done, the DSRU will then start searching the interface list. By comparing the object instance's class ID with the first entry of link list node, the DSRU locate the correct interface list node and takes the data from its second entry.

After renewing the SP, VP, and IPC, the whole invocation completes. The reference flow of interface method invocation is shown in Fig. 28.

The Fig. 29 describes the state machine of DRSU during interface method invocation. In the first stage "Interface Obj ID," the DSRU will recognize the object instance's class ID. The following three stages are the same as in the normal method invocations. The "Interface list ID" stage compares the object instance's class ID with the first entry of link list node. If the ID is not the same, the DSRU will enter the "Interface next list" stage to move on to the next node. If the ID hits, the DSRU will enter "Interface Offset" stage to get the class ID and method offset. And the remaining stages are the same as in the normal method invocations.

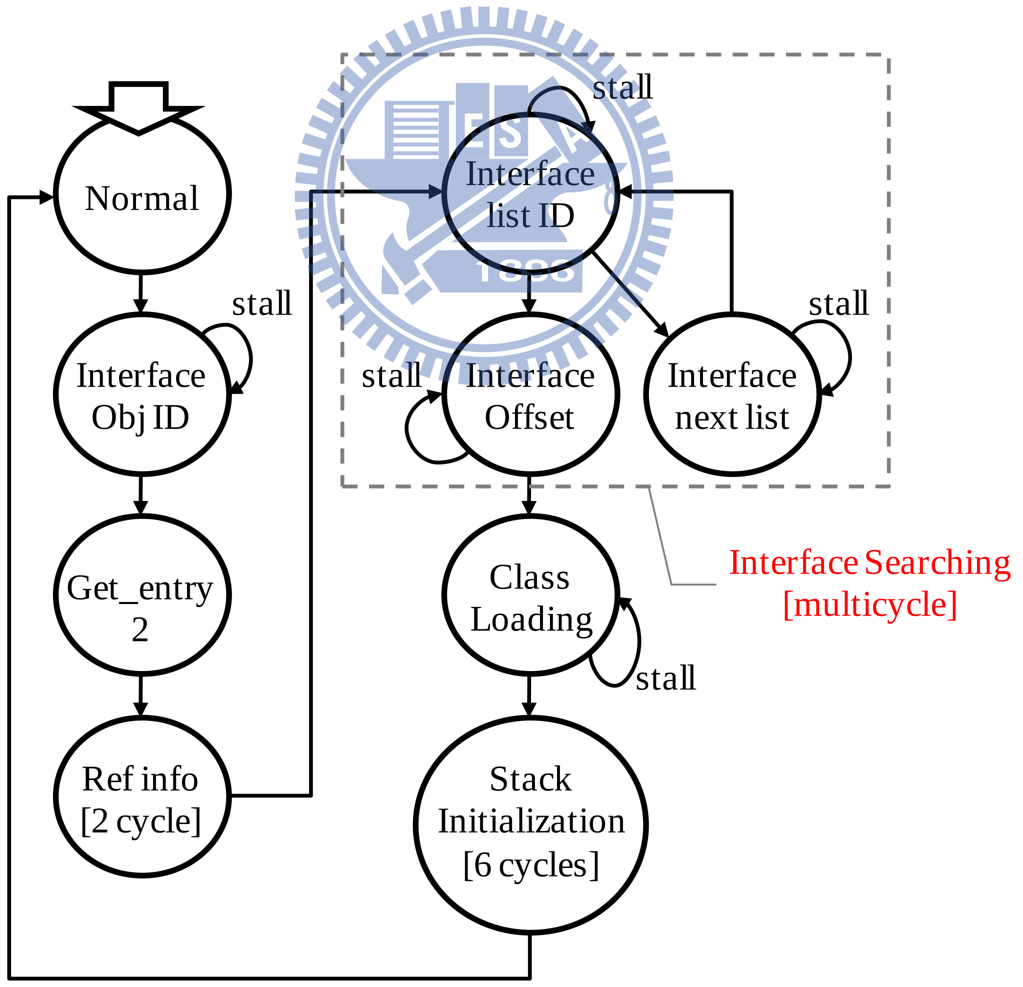
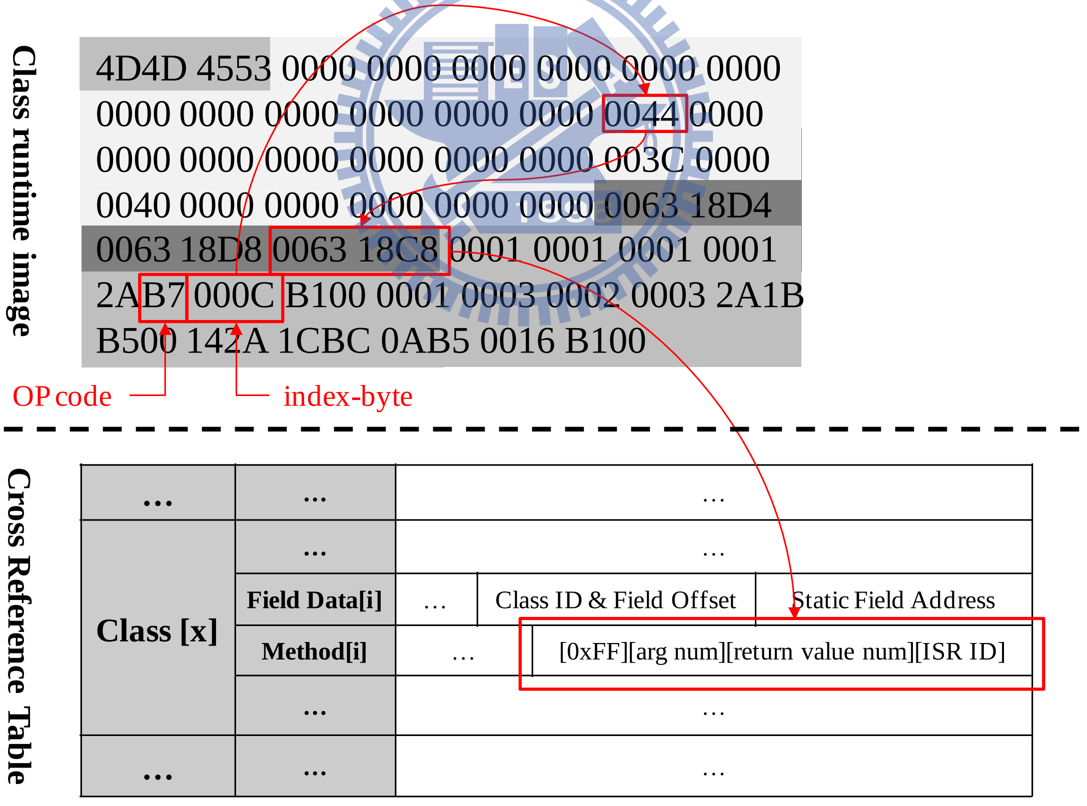


Fig. 29. The interface method invocation state machine of DSRU.

4.3.3 Native Method Invocation

The beginning of the native method invocation is exactly the same as in the normal method invocations. And the reference flow of it is shown in Fig. 30. The difference is in the resolution information which the DSRU receives. If the most significant byte of the resolution information is 0xFF, the DSRU will initiate native method invocation. The resolution information is composed of 4 parts. The first part is the start byte 0xFF which signals a native method invocation. The follow byte is the number of arguments the native call needs. The third part is the number of arguments the native call will return. The final part is the interrupt service ID of the native call because currently all native calls are implemented in ISRs.



Class runtime image
4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB7 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100
OP code
index-byte
Cross Reference Table
…
Class [x]
Field Data[i]
Method[i]
Class ID & Field Offset
Static Field Address
[0xFF][arg num][return value num][ISR ID]

Fig. 30. The reference flow of Native Method Invocation.

Since the beginning is the same as in the normal invocations, the top three items of stack are the return frame. The stack needs to be adjusted depending on the argument number of the native call. If the native call needs no argument, then the stack does not need adjustment. If the native call needs 1, 2, or 3 arguments, then the stack needs to pop 1, 2, or 3 items. If the native call needs more than 3 arguments, it needs to pop 3 items from stack and export the arguments (except the top three of stack) to the inter-processor communication unit (IPC). After the argument is handled properly, the DSRU will raise the interrupt with the interrupt service ID of the native call. Once the interrupt returns, the stack needs to pop the arguments that have been used if the number of argument is more than three. This operation is implemented by adjusting the stack SP. The stack needs another adjustment because the native function may have returned values. If there is no return value, it will pop the top three items of stack. And if there are 1or 2 return values, the stack needs to pop 2 or 1 item. After that, the whole native invocation process completes.

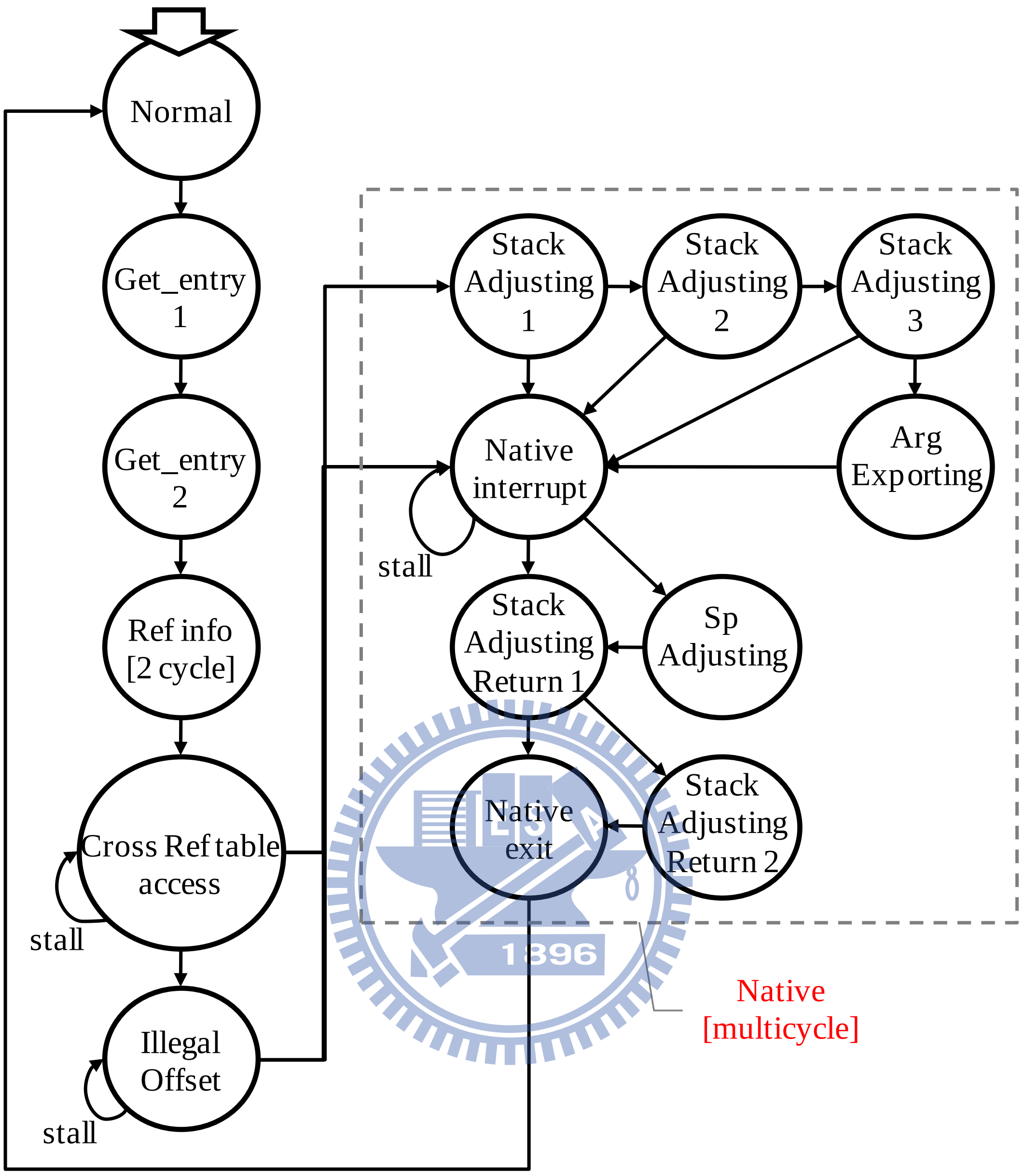


Fig. 31. The native method invocation state machine of DSRU.

Fig. 31 shows the state machine of the DRSU during a native method invocation. The "Stack Adjusting 1, 2, and 3" stage are the pop operation. The DSRU will export arguments to the IPC in the "Arg Exporting" stage. During the native call execution, the DSRU will be stalled in the "Native interrupt" stage. The "Sp Adjusting" stage is the operation of popping the argument that has been used. The "Stack Adjusting Return 1 and 2" stages perform stack adjustment operations to handle the return values.

4.4 Field Data Access Mechanism

Because of the JAIP's system class loader, the operation of field data accesses is very similar to that of normal method invocations. The reference flow of it is shown in Fig. 32. The difference is in that the resolution information which the DSRU receives is an object field offset. By adding the object field offset to the object reference address, the JAIP can easily access the object field data. Fig. 33 shows the state machine of DRSU during a field access. The DSRU will raise master-store signal in the "Field Store" stage and raise master-load signal in the "Field Load" stage.

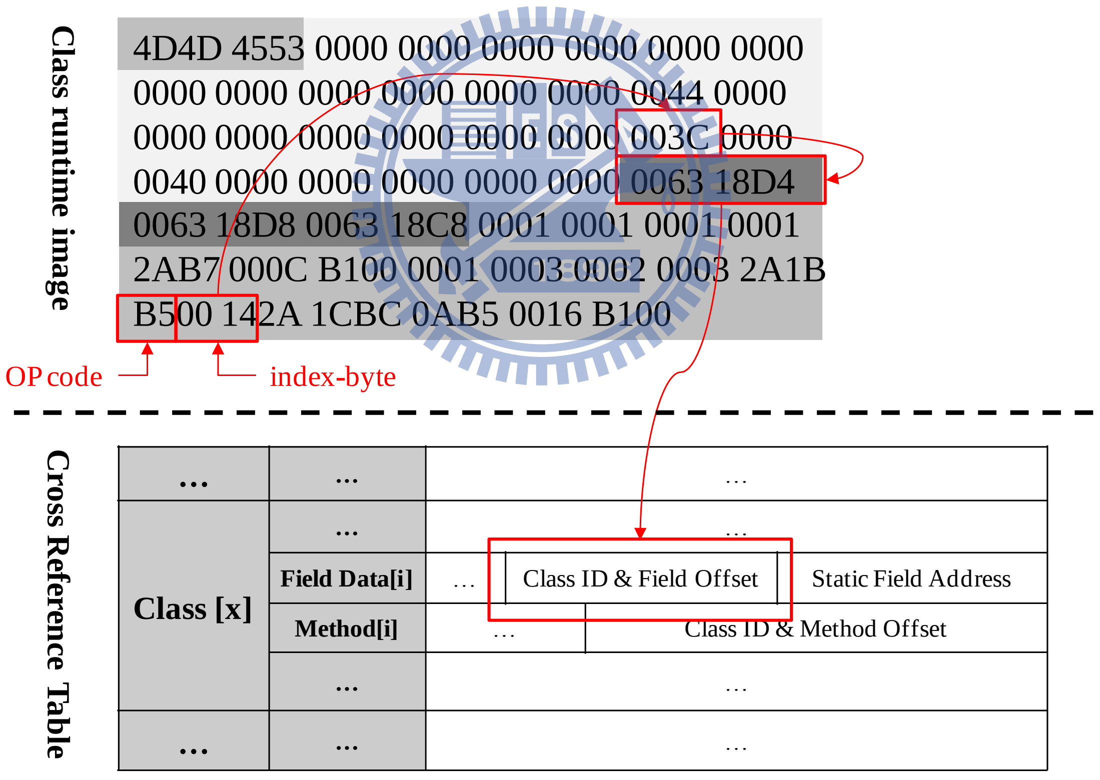

Class runtime image
4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB7 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100
OP code
index-byte
Cross Reference Table
…
Class [x]
Field Data[i]
Method[i]
Class ID & Field Offset
Static Field Address
Class ID & Method Offset

Fig. 32. The reference flow of field data accessing.

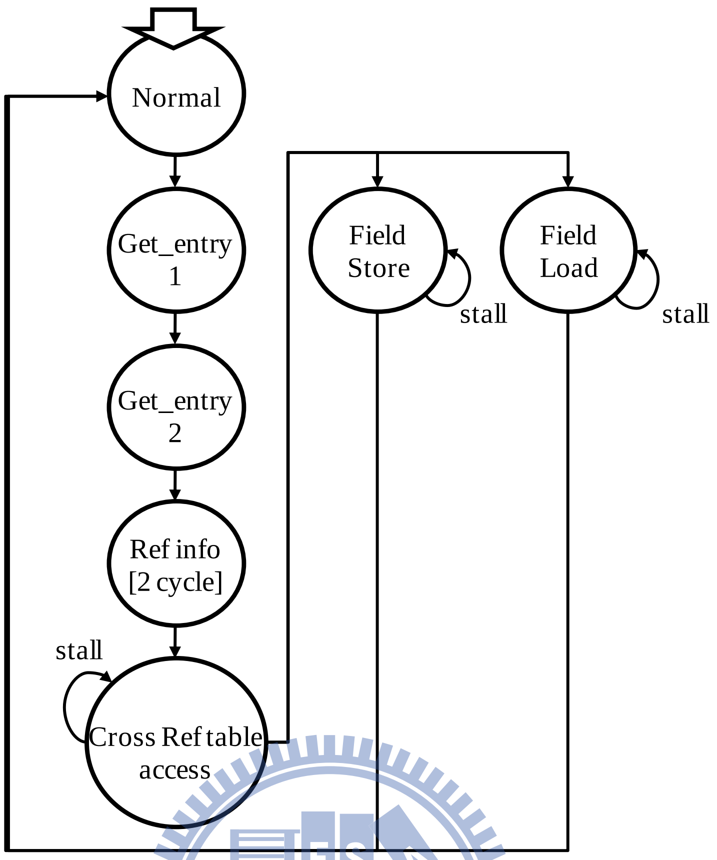
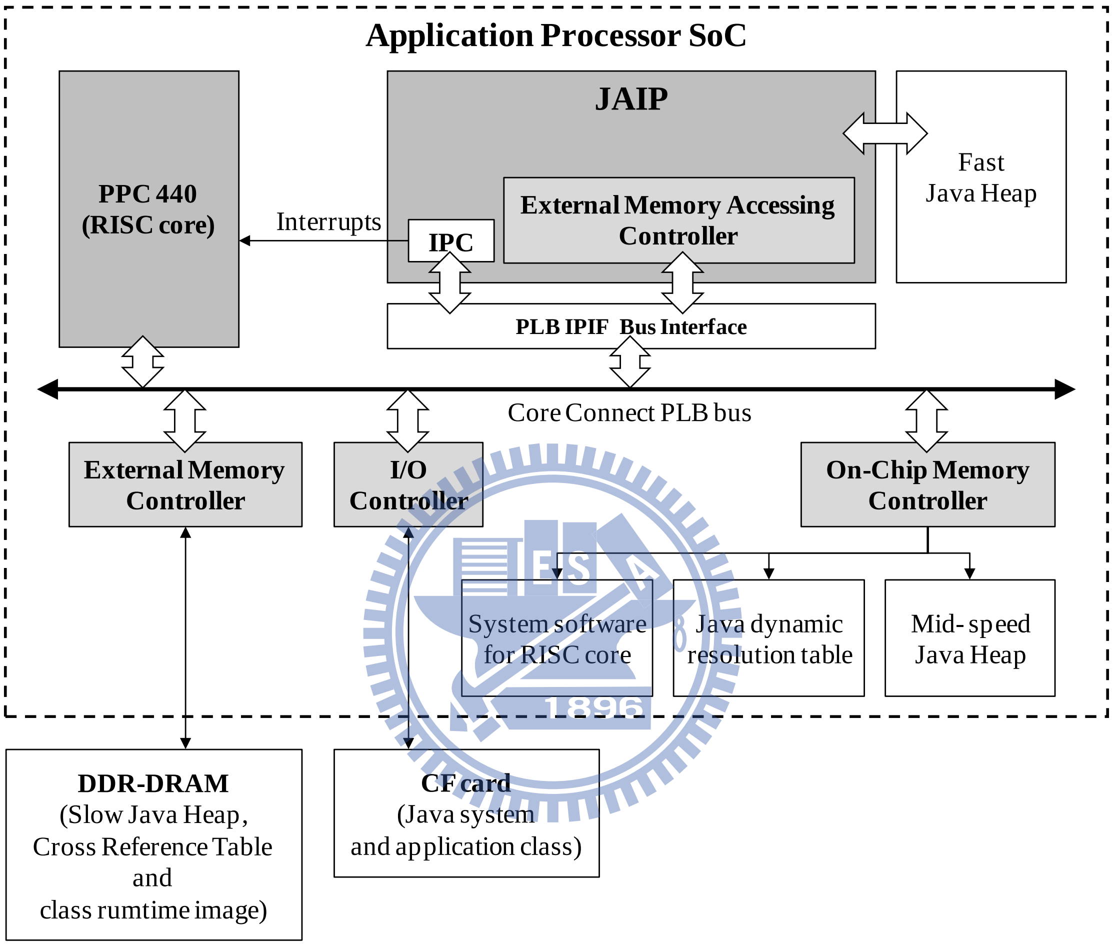


Fig. 33. The field data accessing state machine of DSRU.

Chapter 5. Experimental Results

5.1 The Proposed System Implementation



Application Processor SoC
PPC 440 (RISC core)
Interrupts
JAIP
IPC
External Memory Accessing Controller
Fast Java Heap
PLB IPIF Bus Interface
Core Connect PLB bus
External Memory Controller
I/O Controller
On-Chip Memory Controller
System software for RISC core
Java dynamic resolution table
Mid-speed Java Heap
DDR-DRAM (Slow Java Heap, Cross Reference Table and class rumtime image)
CF card (Java system and application class)

Fig. 34. The field data accessing state machine of DSRU. Note that there are three possible connections between Java heap and JAIP. However, only one of them will be adopted when synthesizing.

The complete Java embedded platform is implemented on an SoC emulation platform, the Xilinx Virtex5 ML507. The FPGA XC5VFX70T contains hardcore PowerPC 440, 44800 slices, 128 DSP8E functional units and 5328 Kbits BRAM with 256 MB DDR2 memory. We use Xilinx Embedded Development Kit 13.1(EDK) as the development tool and Xilinx® Synthesis Technology (XST) as the FPGA synthesis tool. The design suite also provides full system simulation verification for EDK development platform and ISE. We create an implementation platform from Base System Builder (BSP) wizard of Xilinx Platform Studio (EDK XPS) as shown in Fig. 34. We adopt PowerPC 440 as the RISC host processor. The RTL model of the JAIP is written in VHDL and synthesized by Xilinx XST. The synthesis report is shown in Table 4. Both the JAIP and the system bus (PLB [27]) frequency are set to 100 MHz.

Selected Device : 5vfx70tff1136-1		
Number of Slices:	9252(3528) out of 44800	20(7)%
Number of Slice 6 input LUTs:	8755(4044) out of 44800	19(9)%
Number used as logic:	8390(4404)	
Number of IOs:	212(0)	
Number of bonded IOBs:	120 (0) out of 640	18%
Number of Block RAM/FIFO	35 (19) out of 148	23(12)%
Number using Block RAM only	35(19)	
Number of PPC440:	1 out of 1	100%
Minimum period: 10.681ns	Maximum Frequency: 93.624MHz The value of parentheses is only Java execution engine	

Table 4. Synthesis report of the design on an XC5VF70T device.

5.2 Performance Evaluation of JAIP

This section compares the performance between the CVM running on the RISC core alone and our proposed Java core. We use the Embedded Caffeine Mark (ECM) 3.0, two programs that perform a long-chain of method invocations (CHAIN_40 and CHAIN_28), and a program that calculates   to 500 digits (PI) for benchmarking. Sun's CVM with Just-In-Time (JIT) compilation acceleration is used as a comparison point against the proposed JAIP. JIT is a common software VM acceleration technique The CVM-JIT interpreter also executes on the same Xilinx ML-405 development board using the PowerPC 405 core at 100 MHz.

5.2.1 ECM Benchmark Analysis

However, in order to approach the real application testing, we have performed instruction distribution analysis on five benchmark programs (LOGIC, LOOP, SIEVE, METHOD, and STRING) from ECM according to the classifications in Table 5.The distribution of bytecode instructions are shown in Fig. 35.

Bytecode classification	Description	Examples
Local variables accessing	loads or store data from local variables into stack	iload_1 astore_0
Array accessing	loads or store data from array (heap)	iaload iastore
Field accessing	loads or store data from field (heap) (need Dynamic resolution)	getfield putfield
Stack	allows for push or pop data into the stack	iconst_1 pop
ALU	arithmetic or logic instruction	iadd iinc
Method invocation	method call (need Dynamic resolution)	invokevirtual
Branches	condition branch or jump, and return	ifne goto return
Instructions that invoke RISC service	The instruction is implemented with interrupt but does not belong to any upper class.	new ldc

Table 5. Classification of Java bytecode instructions.

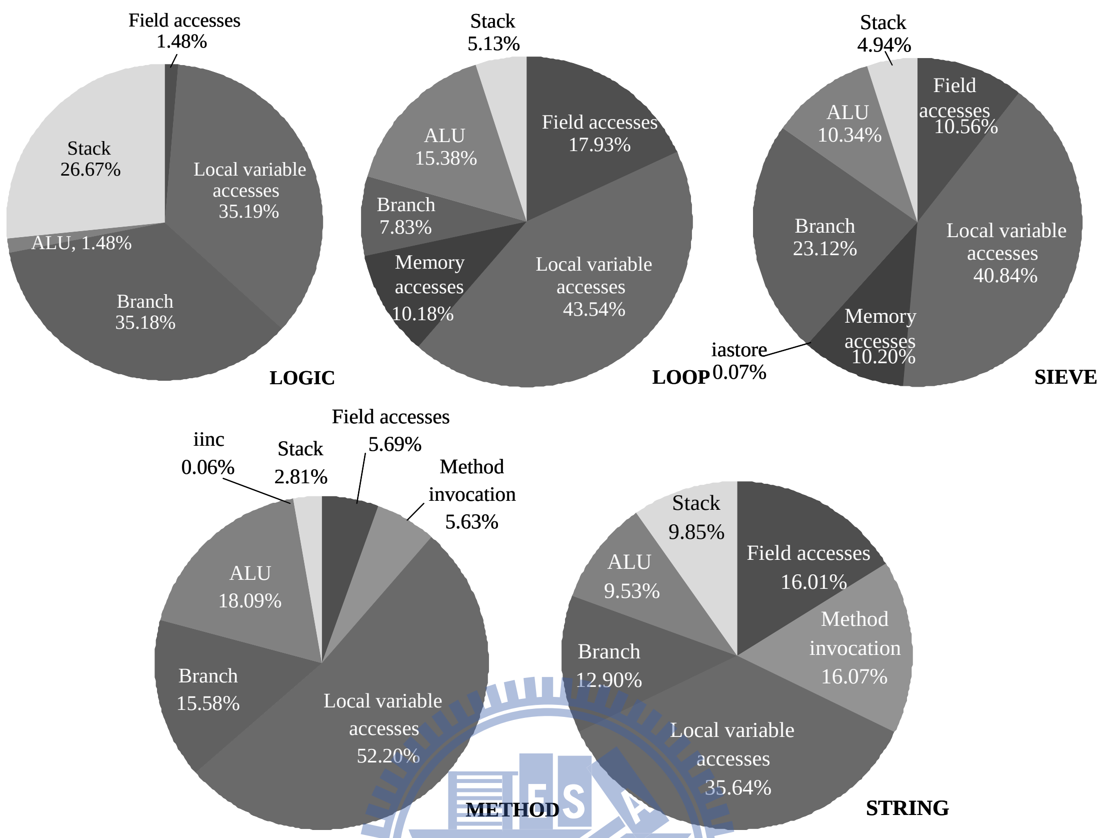


Fig. 35. Instruction distribution of ECM programs.

5.2.2 Performance of ECM Benchmarks

The benchmark result is shown in Fig. 36. The CVM Java VM interpreter is running on the PowerPC 405 CPU under MontaVista Linux. It is important to point out that, on the target platform, the PowerPC 440 core has a 32 KB data cache for CVM-JIT. Since the proposed JAIP does not have any data cache, it would be in great disadvantage when executing a benchmark program which accesses Java heap stored in SDRAM frequently. However, the proposed architecture has the flexibility of allocating Java heap to one of three possible memory areas: external SDRAM, on-chip SRAM accessible via system bus, or on-chip SRAM connected exclusively to the Java core (i.e., the fast Java heap illustrated in Fig. 34). We use the fast Java heap for object storage in this thesis to simulate the case with a data cache.

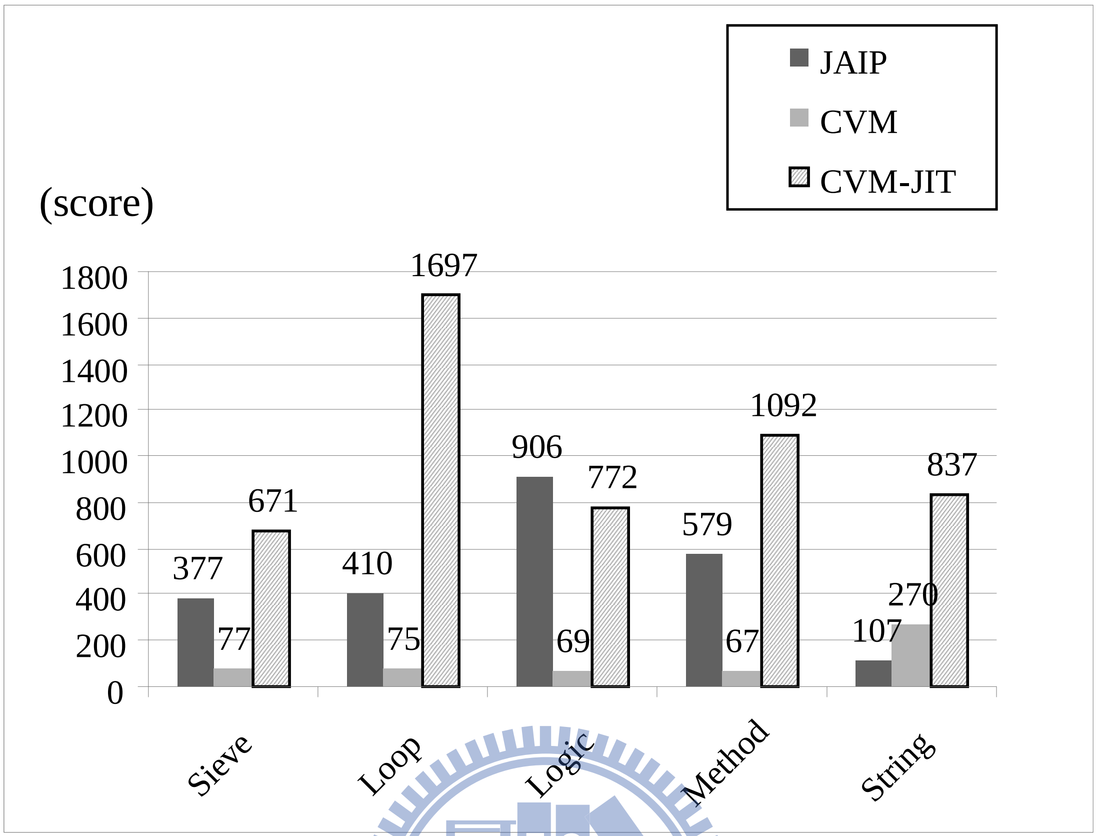


Fig. 36. ECM benchmark results. Higher number means better performance.

Fig. 36 shows the ECM performance of CVM (Java interpreter-only), CVM-JIT (Java interpreter with JIT enabled), and the proposed JAIP, respectively. The calculation of each benchmark is described as follow: Sieve compute prime number below “512”. Logic changes the Boolean logic state 2400 times. Method calls recursive method invocation for a total of 10060 times. Loop counts the Fibonacci sequence below 64 for a total of 4036 iterations.

In the Fig. 36, the JAIP only outperforms CVM-JIT on the LOGIC benchmark. The reason is that the three benchmarks, SIEVE, LOOP, and METHOD, have control structure that can be optimized by the JIT compiler very efficiently.

The reason for the low performance in STRING benchmark is quite simple; the string manipulation operations are implemented on the RISC side as ISRs in current implementation,. That is, every string operation is an IPC request to the RISC core, and the overhead is significant. Therefore, if the target application of JAIP requires a lot of string manipulation, the instruction set of the microcodes of JAIP should be extended to facilitate direct string manipulation within the Java core.

5.2.3 Performance of the long-chain of method invocation and the PI test program Benchmarks

The performance of CVM, CVM-JIT, and JAIP on these programs is shown in Fig. 37.

According to the performance result of the ECM, we assume that JIT does not work well for short burst, sporadic program behaviors (for example, the control branches in the LOGIC benchmark). Therefore, we added two more test programs here to demonstrate that the proposed JAIP does perform better than CVM-JIT when the control structure of the program is hard to optimize by a JIT compiler.

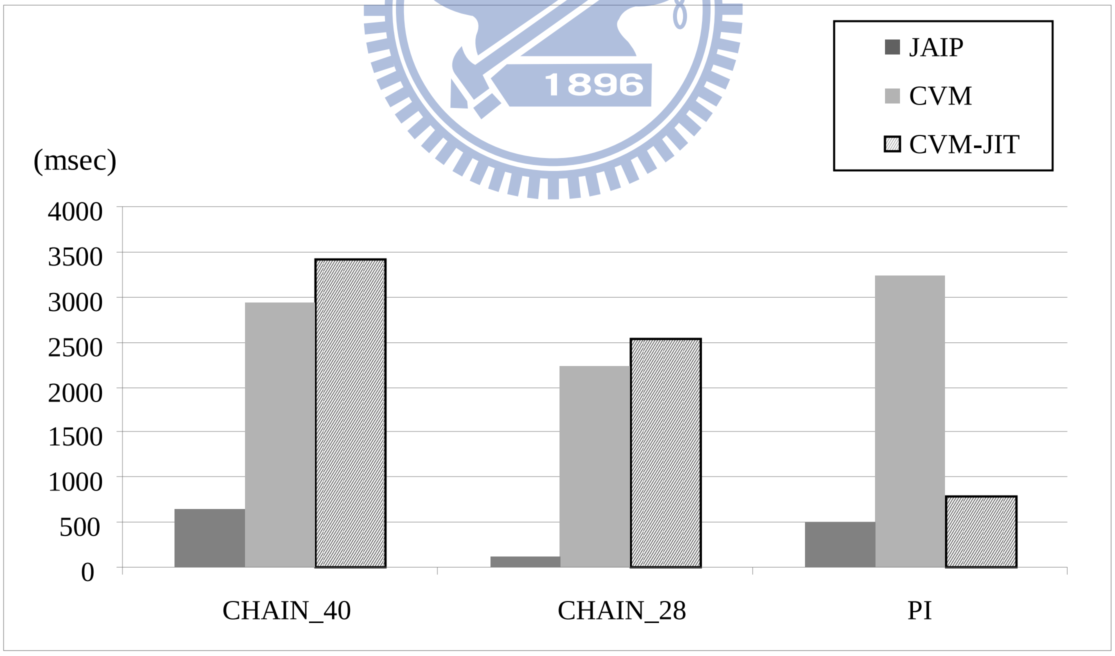


Fig. 37. CHAIN and PI benchmark results. Lower number means better performance.

The first one is similar to the METHOD benchmark of ECM. However, instead of performing recursive method invocation, we perform a long chain of method invocations across 40 (CHAIN_40) or 28 (CHAIN_28) small classes repeatedly. For the CHAIN_40 test, there is a main class and forty different small classes. Each small class has only one tiny method. The main class calls the only method in class one, which calls the only method in class two, and so on. Until the fortieth class' only method is called, this method will call back to the method in class one. This cyclic invocation behavior continues until 100 method calls have been made. Then, the 100th calls returns to its caller and so on until the main functions is reached. The main class repeats the cyclic call for 1000 times. This test is designed to test the efficiency of the proposed method area circular buffer design. Since current implementation has 32 blocks, it can accommodates up to 32 classes (if each one is smaller than 2KB) before some class images get flushed. Hence, for CHAIN_40, every method calls involves a runtime image transfer from SDRAM to MACB. For the CHAIN_28 test, the call numbers are the same as CHAIN_40 (i.e. 100 × 1000), but there are only 28 different classes in the cyclic call so only the first 28 calls involves runtime image transfer.

In this case, JAIP runs much faster than both CVM and CVM-JIT. More importantly, CVM-JIT performs even worse than the CVM interpreter. It is evident that if the control structure of a program cannot be optimized by the JIT compiler and the amount of computation in the program is little, JIT compilation overhead would not be negligible. For JAIP, CHAIN_28 runs much faster than CHAIN_40 deservedly because of constant flushing (only 28 times for each of the small class).

The second program is a PI test program. The PI test program calculates to 500 digits and repeats the calculation for three times. This program has higher double-issue ratio of the instructions. For the PI benchmark, JAIP still performs the best. This program has intense calculation that can be speed up by double-issue architecture. To make the point clear, we have recorded the percentage of (microcode) instructions that are double-issued in JAIP for the benchmark programs. The results are shown in Table 6. Note that this double-issue gain is achieved with simple, inexpensive circuitry.

	SIEVE	LOOP	LOGIC	METHOD	PI
% of double-issue	8.67	4.86	9.75	29.41	36.99

Table 6. Percentage of double-issued instructions.

Chapter 6. Conclusions and Future Works

In this paper, we have proposed the detail of architecture design to facilitate the encapsulation of a Java core as a reusable accelerator IP. With the proposed architecture, the Java accelerator IP can be integrated into an application processor SoC easily. The key architecture that enables host processor invocation of the Java accelerator is the Method Area Manager Unit and its associated two-level method area memory hierarchy.

For system software integration with the RISC core, mailbox-driven ISRs on the RISC side are used to modularize the integration effort. The design of the software stack of the JRE follows the concept of the JavaOS model.

For the bytecode execution engine, we have also proposed the two-level stack architecture with local variable cache. With this stack architecture, we can implement a double-issue pipeline with small hardware cost. The implementation out-performs CVM-JIT when the computation and/or control structure of the program is difficult for JIT compilers to optimize.

In the future, there are many issues for complete support of JVM for embedded systems. We will improve the Method Area Manager Unit and the two-level method area memory hierarchy by a “single method caching mechanism”. The main difference is that in current design the Method Area Manager Unit manages the method area by a unit of a complete class. However, some Java classes will be parse-loaded into JAIP just to execute one (or few) method in the class. If the cache block of Method Area Circular Buffer stores just a single method, the cache memory will be used more efficiently.

Another issue is that the JAIP is not capable of multi-threading so far. To support multi-thread in hardware directly, hardware-based context switching should be implemented within JAIP. To decrease the overhead of context switching, multiple copies of the original two-level stack architecture will be used. Each copy of the original stack architecture will contain only a thread stack. The original two-level stack architecture is maintained as a working frame, and the complete stack data is stored in external memory. The swap-in and swap-out of working frame will occur and overlap with the execution time of another thread so that the physical overhead of a context switch will be reduced to just a few cycles. The RISC-side will be responsible for thread scheduling. On the other hand, multiple copies of register files will be used to save the internal registers (e.g. JPC, current class ID, etc.).

Reference

[1] H.-J. Ko and C.-J. Tsai, "A Double-issue Java Processor Design for Embedded Application," ***Proc. of IEEE Int. Symp. on Circuits and Systems****(ISCAS'08),* Seattle**, May. 2007.**

[2] H.-J. Ko, *A Double-issue Java Processor Design for Embedded Application, Mater thesis, NCTU, 2007.*

[3] K.-N. Su and C.-J. Tsai, "Fast Host Service Interface Design for Embedded Java Application Processors," ***Proc. of IEEE Int. Symp. on Circuits and Systems*** *(ISCAS'09)* ,Taipei, May, 2009.

[4] K.-N. Su, *Design of Heterogeneous Dual-Core Java Application Processor for Embedded Applications, Mater thesis, NCTU, 2009.*

[5] C.-F. Hwang, K.-N. Su and C.-J. Tsai," Low-Cost Class Caching Mechanism for Java SoC," ***Proc. of IEEE Int. Symp. on Circuits and Systems****(ISCAS'10),* Paris**, May. 2010.**

[6] C.-F. Hwang, *Design of Dual-Core Java Processor for Interactive 3-D GUI Platform, Mater thesis, NCTU, 2010.*

[7] Bill Venners, *Inside the Java 2 Virtual Machine*, New York: McGraw-Hill**, 2001, ch.5 ch.6 ch.7 ch.8.**

[8] Dan Bornstein, "Dalvik VM Internals," *Googol Developer Conference (Google I/O 2008)*, San Francisco, May 2008.

[9] Sun Microsystems, *J2ME Technology*, Sun Developer Network URL: http://java.sun.com/javame/technology/, 1994-2009.

[10] Sun Microsystems, *Connected, Limited Device Configuration Specification*, ver. 1.0a, Sun Microsystems White Paper, May 2000.

[11] Sun Microsystems, the Java Community Process Program, JSR 36: *Connected Device Configuration*, ver. 1.0b, Dec 20, 2005.

[12] S. Ritchie, "Systems Programming in Java," *IEEE Micro*, 17, 3 (Mar.), 1997, pp. 30-35.

[13] B. R. Montague, "JN: OS for an Embedded Java Network Computer," *IEEE Micro*, 17, 3, 1997, pp. 54-60.

[14] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd Ed., Addison-Wesley, 1999.

[15] C. Porthouse, *High performance Java on embedded devices, Jazelle DBX technology: ARM acceleration technology for the Java Platform*, White paper of ARM Ltd., Oct. 2005.

[16] C.-H. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," *Proc. of 29th Annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO'29)*, pp. 90-99, Paris, Dec. 1996.

[17] A. Krall, "Efficient Java Just-in-Time Compilation," *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 205-212, Paris, Oct. 1998.

[18] J.M. O'Connor and M. Tremblay, "picoJava-I: the Java virtual machine in hardware," *Micro, IEEE*, vol. 17, no. 2, pp. 45 – 53, , Mar./Apr. 1997.

[19] H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine for Java Bytecode," *Computer*, Vol. 31, Issue 10, pp. 22-30, Oct. 1998.

[20] Y. Shi, D. Gregg, A. Beatty, and M. Anton Ertl, "Virtual Machine Showdown: Stack versus Registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 4, Issue 4, pp.153-163, Jan. 2008.

[21] E. Quinn and C. Christiansen, "Java Technology Pays Positively," *IDC Bulletin #W16212*, May 1998. http://wellscs.com/robert/java/productivity.htm

[22] Sun, *picoJava-II Microarchitecture Guide*, Sun Microsystems, March 1999.

[23] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "*A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling*," *Proc. of 1999 Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99)*, pp. 34-39, Newport Beach, Oct. 1999.

[24] Y. Y. Tan, C. H. Yau, K. M. Lo, W. S. Yu, P. L. Mok, and A. S. Fong, "Design and Implementation of a Java processor," *IEE Proceedings*, Vol. 153, pp. 20-30, 2006.

[25] M. Schoebel, "Evalution of a Java Processor," *Tagungsband Austrochip* 2005, pp. 127-134, Oct. 2005.

[26] Xilinx LogiCore, *LogiCORE IP Multi-Port Memory Controller (MPMC)*

(v6.03.a), Xilinx Production Specification DS643, March, 2011.

[27] Xilinx LogiCore, *LogiCORE IP Processor*
Local Bus(PLB) v4.6 (v1.05a), Xilinx Production Specification DS531, Sep, 2010.

[28] P. S. Corporation, Embedded Caffeine Mark 3.0, URL:
http://www.benchmarkhq.ru/cm30/info.html, 1997.

[29] Xilinx LogiCore, *PLB IPIF (v2.02a)*, Xilinx Production Specification DS448, April, 2005.