在翻譯爪哇中介碼成 X86 組語環境下的異常實作

(Exception Implementation based on Translating Java Bytecode to X86 Assembly)

學生：陳宗強　　指導教授：楊 武 博士

國立交通大學資訊科學研究所碩士班

摘要

隨著 Java 程式的普及，其執行效率愈來愈受到重視。最原始的 Java 模擬機器是以解譯 bytecode 來達成。解譯程式通常效率較差，它適合於小程式但不適合於大型的應用軟體。即時編譯器的效率比解譯的效率比解譯器好很多。但仍有不適用的地方，例如當一個程式裡存在有許多只被執行一次的函數時，即時編譯器的效率就會低落。

本論文提出之方法，將使用者所寫的 class 都轉成 Assembly 的型式，包含呼叫自己定義的 class。若呼叫的是 Java 內建 Library 中的 class 檔，則透過 JNI 去命令 JVM 幫使用者執行呼叫之內建 class 檔。此方法跟 JIT compiler 不同的是，JIT 在執行時，才會將某些常常呼叫的函式轉成 Native code，而本論文的方法是將使用者所寫的都轉成 Native code，進而達到加速執行的速度。

誌謝

首先，我要感謝我的指導教授　楊武博士的指導，由於老師專業知識的傳授與持續不斷的幫助和指導，並督促我做實驗和寫論文，假如沒有老師的鼓勵和指導，學生很難順利完成這篇論文，在此，向老師致上最深的謝意。

接下來，我要感謝的是程式言與系統實驗室的所有同學，大家的陪伴讓我在這兩年來的生活中感到許多溫暖。明彥、承穎學長及雅芬學姐，在我功課繁忙時，常常關心我的生活和課業，並提供意見，這讓我很快地適應研究生的生活，真的很謝謝他們。俊元和恆琳是和我要一起畢業的同學，我們同舟共濟，互相鼓勵，我在寫論文的過程中，著實受到他們許多幫助，我非常感謝他們。

最後，感謝我的家人，我在研究所的這段日子，使我無後顧之憂，能夠順利完成碩士學位，最後僅以此論文獻給我最親的家人和朋友，由衷地謝謝他們

Content

Figure

Chapter 1

Introduction

Java is a popular object-orient language. Java may be used as a conventional programming language for writing applications, and it may also be used to write applets on the WWW. The following characteristics are the why Java is popular. [5]

- Architecture Neutral and Portable

  The Java compiler compiles Java programs into bytecode, and bytecode helps to transport code to different software and hardware platforms easily. Java gives detailed definitions to the value range and storage format of its primitive data types, the behavior of its arithmetic operators, etc. The programs are the same on every platform. There are no more data type incompatibility problems.
- Simple and Object-Oriented

  Like C++ but it removes many unnecessary features. No typedef, define, etc., preprocessor commands、No structures or unions、No external functions、No multiple inheritance、No goto statement、No operator overloading、No automatic coercion、No pointers.
- Robust and Secure

  During the compiling phase, it provides compile-time type checking. But due to the unusual bytecode representation mechanism, Java provides runtime checking in the Java Virtual Machine.
- Multi-thread

The Java library provides a class java.lang.Thread that contains a collection of methods modeling a thread life cycle.

1.1 Motivation

Interpreting bytecodes makes Java program many times slower than comparable C or C++ programs. One approach to improving in this situation is “Just-In-Time” (JIT) compilers. Dynamically translate bytecodes to machine code just before a method first executes. This can provide substantial speed up, but it is still slower than C or C++. While JIT compilers have an important place in a Java system, for frequently used applications it is better to use a more traditional “ahead-of-time”. While Java has been primarily touted as an internet/web language, many people are interested in using Java as an alternative to traditional languages, if the performance can be made adequate. For embedded system

applications it makes much more sense to pre-compile the Java program. So far, there are some tools that can offline translate Java source/Java bytecode to native code for performance enhancement. But, lack for translating to original assembly language code. Therefore, we propose a method that can translate java bytecode to X86 assembly code.

1.2 Goal

The goals of our work are to develop a tool that models the flow of translation without generating intermediate representation, and simplifies the task of translating Java bytecode, and demonstrates significant optimizations which touch bytecodes directly and can improve the performance. An optimizing translator from Java bytecode to native machine code which generates intermediary representations is too complex and understands difficultly, and today the performance by JIT compiler has acceptable state. Therefore, we hope to build a simple translator that not only shorten the steps of translation and simplify the method of translation, but also running time of translated assembly codes by translator can be close to the running time of translated native code by JIT compilers or other native compilers. So, our translator simplifies the translation flow, and still has good performance. The exception handle in java is very important. Even some Java development environment asks the user to write the exception handle in program, we also implement the exception handle in Java. We can throw an exception in java and use try catch, so in our implement the exception handle is needed.

Chapter 2

System implementation

2.1 System Architecture

2.1.1 How to call JVM

First, we must understand how to call JVM. We use starvm.c to help us to create JVM. In startvm.c, it will return the JVM reference. We can use the JVM reference to call the JNI function. Through calling the JNI function, we can call system class when program bumping into calling some class function defined in JVM. The flowing chart tells us how to call a system class in assembly. The startvm.c program is in the index A.

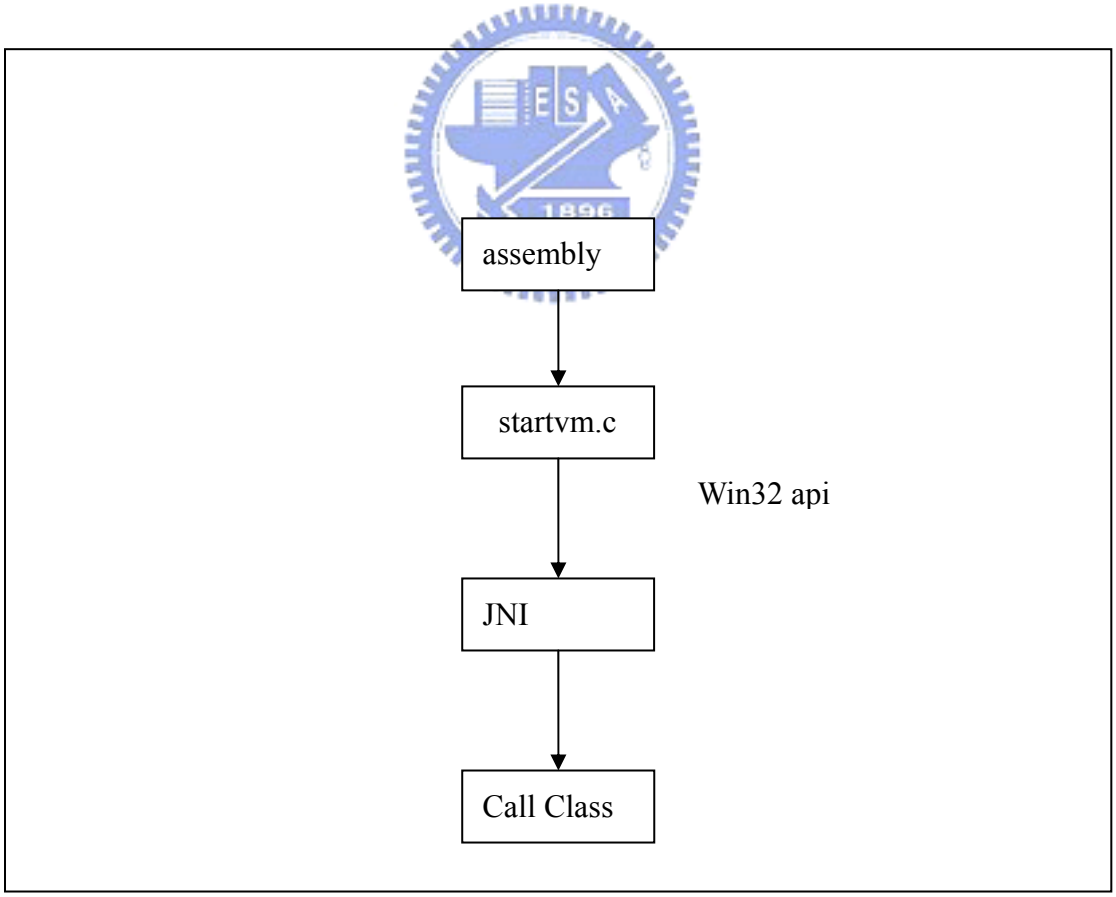

Figure 2.1 Call the JNI function from assembly

```
call    _startvm
mov     jnienv,eax
mov     ebx,[eax]
mov     fntblptr,ebx
```

Figure 2.2 In the beginning of the assembly

In the beginning of our translating into assembly, we call startvm function which defined in startvm.c and store the JVM reference in a word file jnienv[10]. The return JVM reference is the pointer which point to a location that contains a pointer to a function table. In briefly, the jnienv is a double pointer. We also store the pointer which is pointed by jnienv pointer in a word fntblptr for convenience of calling the JNI function.

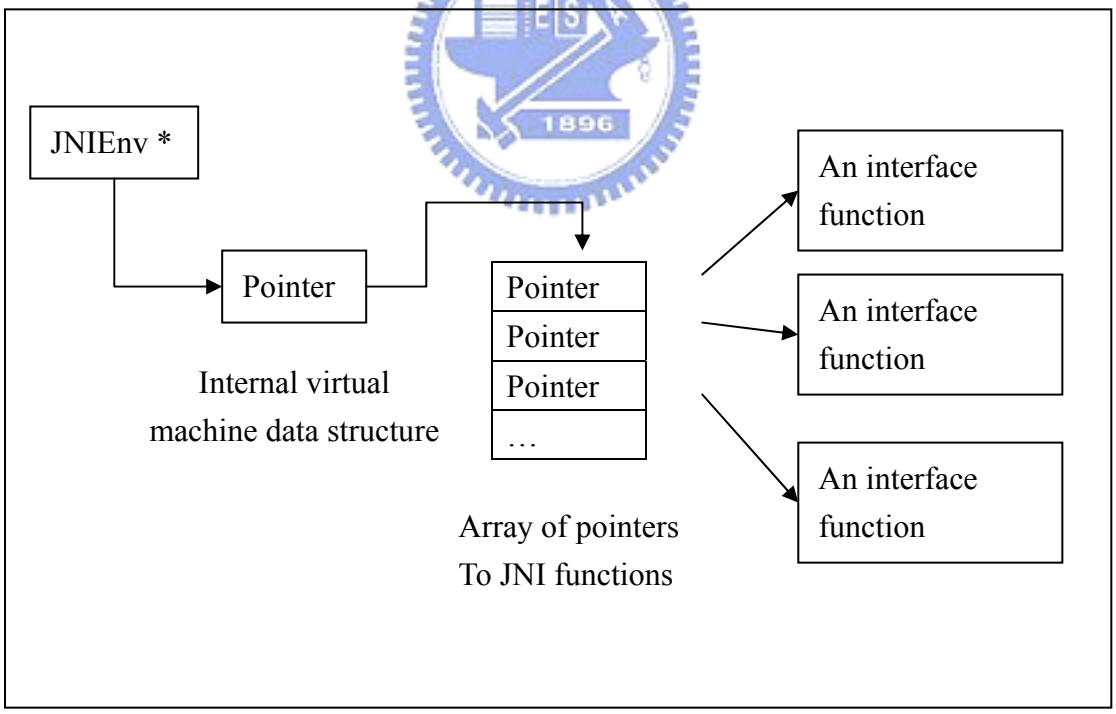

Figure 2.3 JNI function pointer

Before calling class function, we must know how to call JNI function at first. When calling the JNI function in assembly, we must know the index of the JNI function. The index is added to fnblptr for searching the address of mapping JNI function. In our implementation, we write an assembly macro subroutine GetFnptr to help us add the index

to the fntblptr and store the result into the word fnptr which point to the address of mapping JNI function. Note that in assembly the address is word base, so the index must be multiplied by 4 and added to fnblptr for the correct address.

```
GetFnPtr MACRO fntblptr,index,fnptr
        mov     eax,index
        mov     ebx,4
        mul      ebx
        mov     ebx,fntblptr
        add      ebx,eax
        mov     eax,[ebx]
        mov     fnptr,eax
ENDM
```

Figure 2.4 Get function pointer macro

After understanding how to call JNI function, we must know how to call a class function through JNI function. When calling a class function in Java, the JVM will create an object and call the method by it' s name through the object. In translated assembly, firstly we must get the class id through the JNI function FindClass to create an object. Before creating a new object, we must get the constructor id for operating the constructor. So we use the JNI function GetMethodID which passes the method name "<init>" and method signature "( )V" [10]. Now we can create the object through the JNI function NewObject by passing the class id and constructor id. When we get the object id, we call the method by the JNI function Call<Type>Method and CallStatic<Type>Method according to call what static or nonstatic method. The Type is defined by the signature which can be captured in class file.

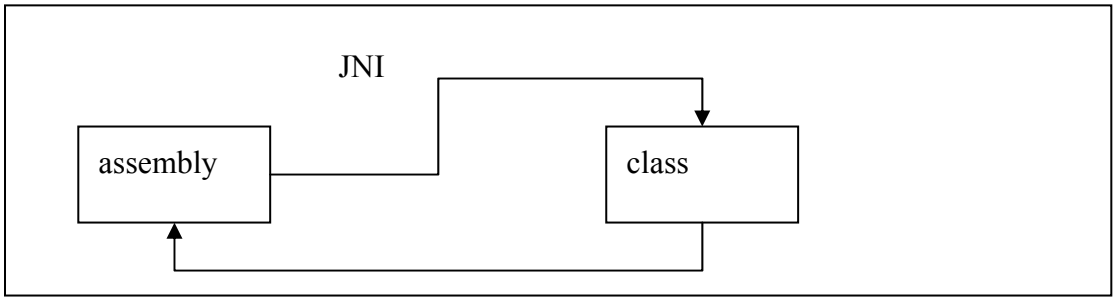

Figure 2.5 Assembly call Java API

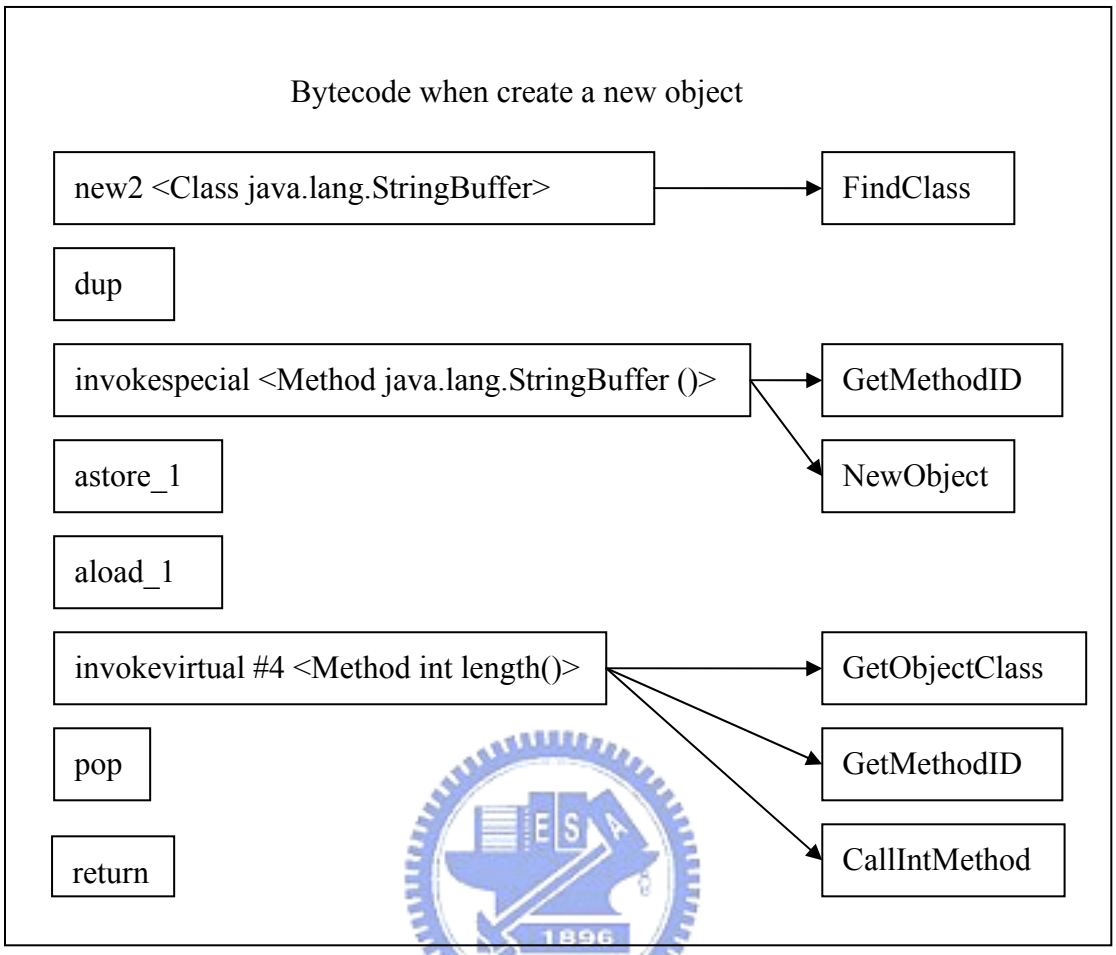

Figure 2.6 Btyecode mapping to JNI function

Now we understand how to call a java library class from assembly in our system. The following section will introduce how to call a class which we write in Java and is translated into assembly. In our implementation, the constructor is not translated into assembly, because we must need the class object id which can be used to field instructions、array instructions and etc. So the new class bytecode instructions which are translated into assembly instructions are the same as the above call library class instruction. The only difference is invoking instruction and getfield instruction. In those instructions, we must need to judge the calling instruction difference. How do we know the difference between calling library and class which we write? If the object class has the name which begins with java, the invoking instruction is translated in call instruction. The following figure shows how to call the program which is wrotten by Java and is translated into assembly.

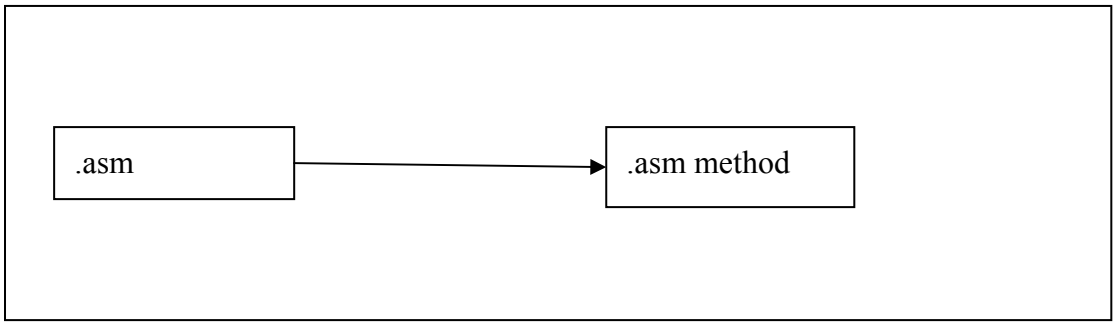

Figure 2.7 Mapping to Assembly

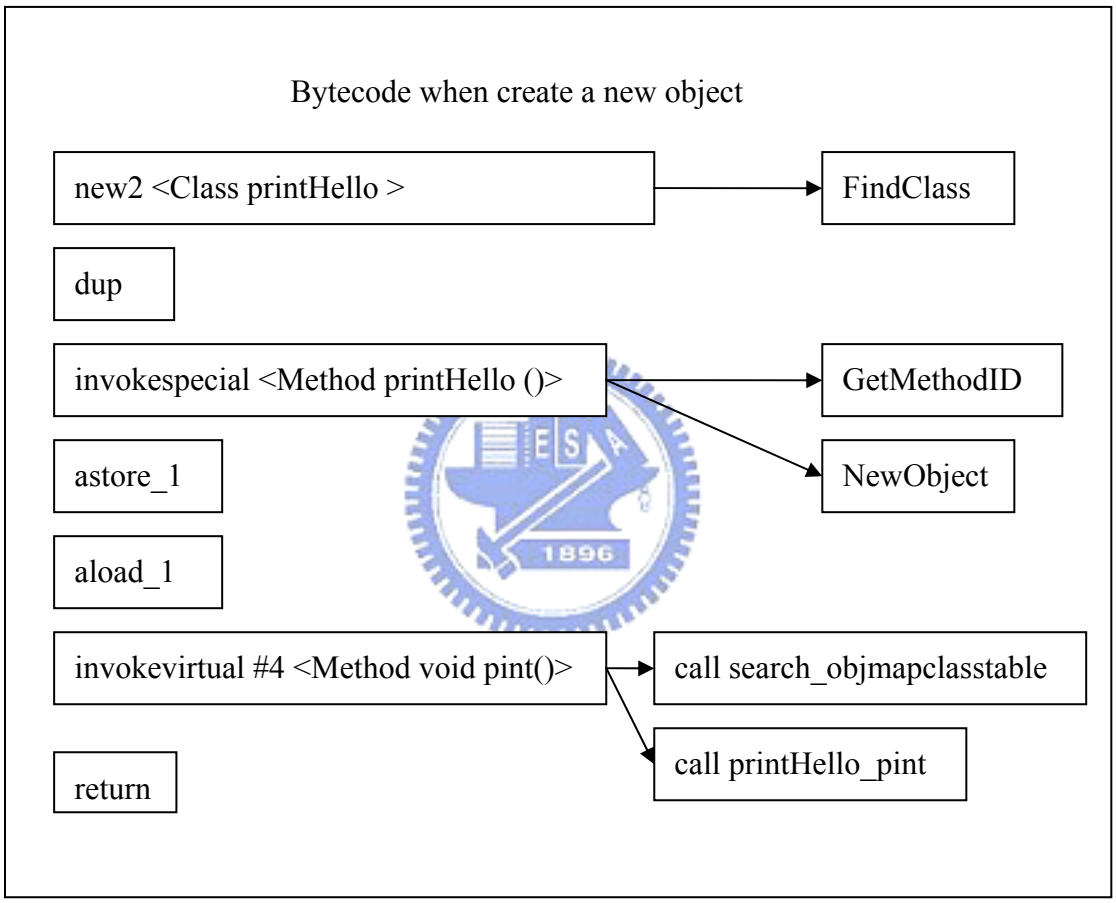

Figure 2.8 Calling the user class

2.1.2 System Implementation Architecture

The following figure explains the environment that is from Java applications (.class) to X86 assembly codes (.asm), and then is from assembly codes (.asm) to execution file (.exe). If there is a Java application which consists of multiple class files, we translate every class file separately to assembly code file. Then, we combine those translated assembly files by using bcc32 compiler to assembler and link those translated assembly files. Because our translater makes the interaction from assembly codes to Java Virtual Machine, we prepare

a C program called startvm.c, which will starts Java Virtual Machine. We use Microsoft Win32 API to start the Java Virtual Machine, and use Java Native Interface (JNI) to let assembly codes have the capacity which communicates assembly codes with Java Virtual Machine.

The following figure explains how to execute those translated assembly files. We use the bcc32 compiler to assembler and to link those assembly files with Java Virtual Machine. The c:\j2sdk1.4.1_01 is the directory where the Java 2 sdk is installed. You need to supply correct include and library directories that correspond to the JDK installation on your machine. The –Ic:\j2sdk1.4.1_01\include\win32 option ensures that your native application is linked with the Win32 multithreaded C library. The actual Java Virtual Machine implementation used at run time is contained in a separate dynamic library file called jvm.dll. The linkage information about invocation interface functions are contained in a file called jvm.lib. Frmat of the jvm.lib belongs to COFF format, but the bcc32 compiler only processes OMF format. So, you have to use the command called coff2omf to change COFF format to OMF format. The jvm1.lib is a canged OMF format.

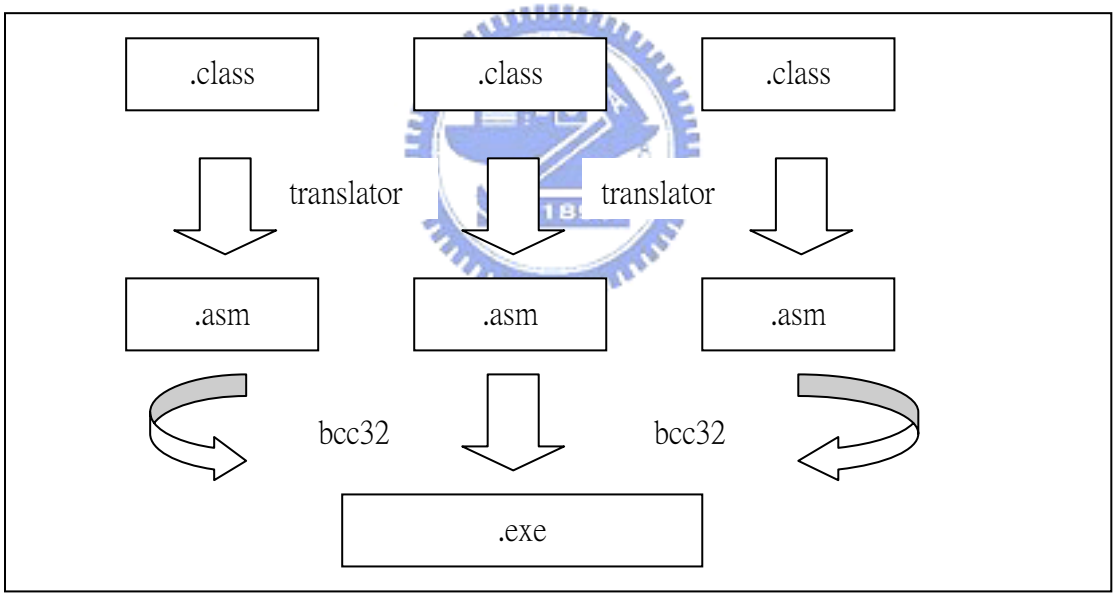

Figure 2.9 Multiplified classes translation

The following figure explains how to execute those translated assembly files. We use the bcc32 compiler to assembler and to link those assembly files with Java Virtual Machine. The c:\j2sdk1.4.1_01 is the directory where the Java 2 sdk is installed. You need to supply the correct include and library directories that correspond to the JDK installation on your machine. The –Ic:\j2sdk1.4.1_01\include\win32 option ensures that your native application is linked with the Win32 multithreaded C library. The actual Java Virtual Machine implementation used at run time is contained in a separate dynamic library file

called jvm.dll. The linkage information about invocation interface functions are contained in a file called jvm.lib. However, the format of the jvm.lib belongs to COFF format, but the bcc32 compiler only processes OMF format. So, you have to use the command called *coff2omf* to change COFF format to OMF format. The jvm1.lib is changed OMF format.

```
bcc32  -Ic:\j2sdk1.4.1_01\include  -Ic:\j2sdk1.4.1_01\include\win3
a.asm  b.asm  c.asm  startvm.c  c:\j2sdk1.4.1_01\lib\jvm1.lib
```

Figure 2.10 Command line in translating

According to above figure example, there is Java application which consists of a.class、b.class and c.class. These are three class files may come from different Java programs, but they are cooperation. These three class files are translated alone by our translator, and generate a.asm, b.asm, and c.asm. The a.class contains the main procedure which is starting point, so put the a.asm on the first place. The startvm.c is used to startup the Java Virtual Machine. After executing the compilation action, it generates a execution file called a.exe. The a.exe is the truly executable file. Assembly codes can bind the Java Virtual Machine through this compilation method, and then call Java api from assembly codes. The tool which assembles and links X86 assembly files in bcc32 compiler, is through TASM.

Presently, our translator is developed in C++, and translation environment based on the Microsoft Windows operating system. For generating accurate X86 assembly codes, we adopt the two-pass operation. The first pass collects informations from the input class file and from the result of *javap* the classfile. Why use the *javap* command which the J2SDK provides? Because the result of *javap* the class file shows Java assembly codes of methods. Among these informations from the first pass, there is information about the bytecode instructions within methods. So, we merely use the command *javap* to get bytecode instructions within methods and other much important information are from the class file. The *javap* function is hidden in translator.

Following figure shows the simple translation environment. The input class file is translated to X86 assembly code and the hidden *javap* action is performed by translator.

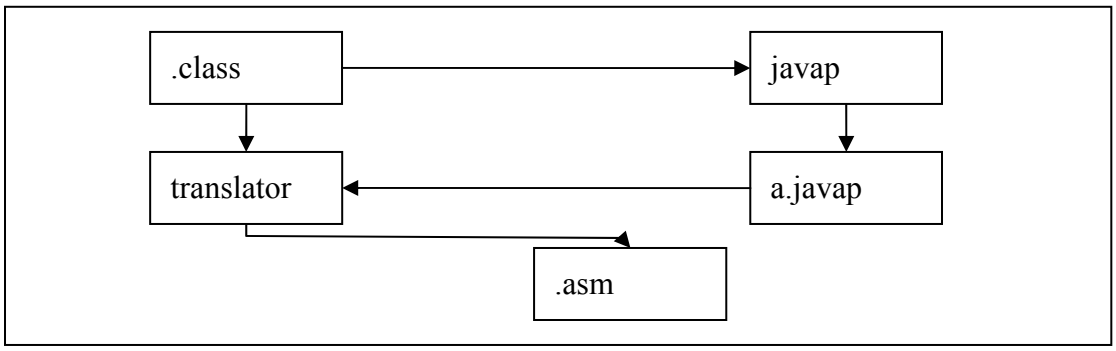

Figure 2.11 Single class translation

The total translation flow is shown in the following figure. We have modularized the code into 6 distinct stages. In stage 1, each method´s bytecodes and associate information is extracted from the input classfile and the result of javap. Decomposer extracts (1) the class´s bytecodes, (2) all method invocation signatures, (3) constant pool contents, (4) the total number of local variables used by the method, (5) the maximum number of operand stack used by the method, (6) static flag, (7) all method exception tables

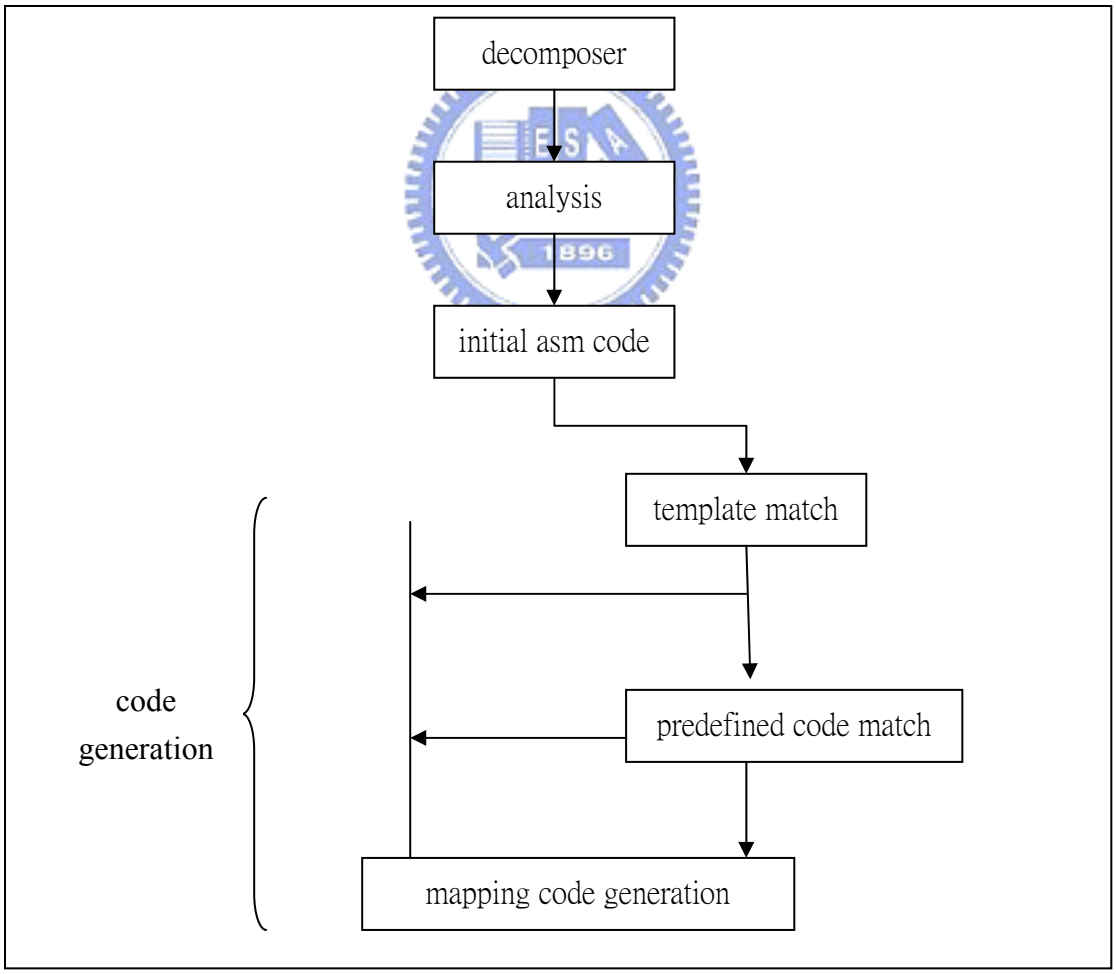

Figure 2.12 System environment

When template matching fails and predefined code matching fails during translating, the type of the bytecode instruction is differentiated, and the translator performs associated mapping code generation. The type of the bytecode instruction is classified into six categories which are stack and local variables、array、arithmetic and logic and type conversion、flow control、object、method invocation. The following figure is the system flow chart.

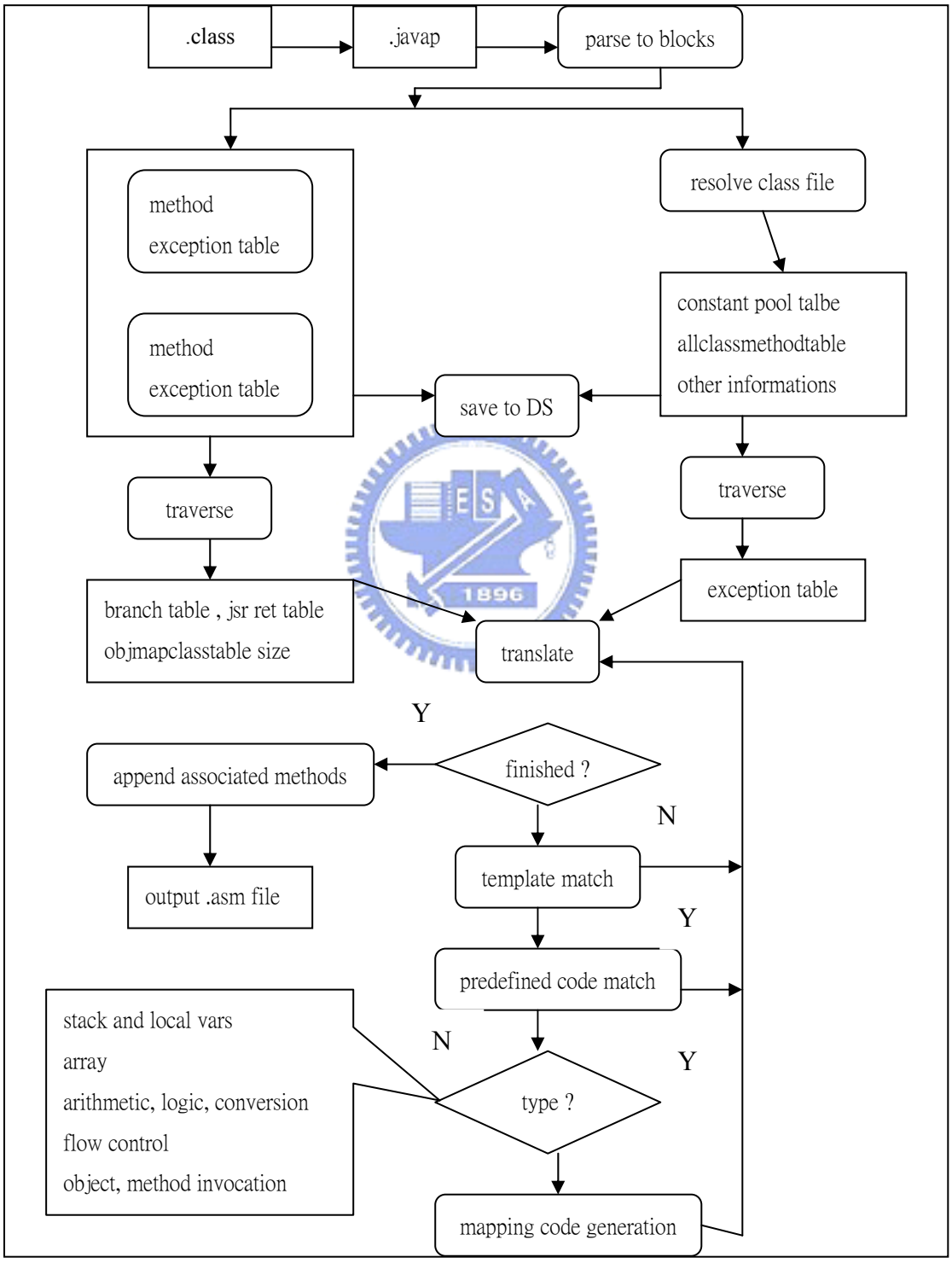

Figure 2.13 System overview

2.1.3 Pass 1

We traverse the bytecode instructions within a method to build branch table and Jsr Ret table. The branch table has all the offsets which occur among those instructions－if_…, lookupswitch, tableswitch, jsr, jsr_w, exception´s target. The jsr ret table has the offsets which are the offsets of next instruction of jsr or jsr_w. The branch table is useful in flow control, and the jsr ret table is useful in exception handling. The instruction map table is the predefined assembly code.

index	type	classname	method/field name	signature	resolved

Table 2.1 Constant Pool Table

```
.classname   idofclass   superclass
    methodname   signature   clsname   callasmname
       …
    methodname   signature   clsname   callasmname
.end
```

Table 2.2 Allclassmethodtable

index	label	mode

Table 2.3 Branch Table

index	label

Table 2.4 Jsr Ret Table

exception type	from	to	target

Table 2.5 Exception Table

bytecode	assembly

Table 2.6 Instruction Map Table

2.1.4 Pass 2

In this section, we focus on pass 2 and describe how the translator translates bytecode instructions. Pass 2 begins the bytecode translation. When bumpting into bytecode instruction, we first search the basic instruction to match the corresponding instruction. If we do not find the corresponding instruction, we implement it dynamically. In this paper, we name this instruction special instruction. The following figure shows the flow of the translator translating.

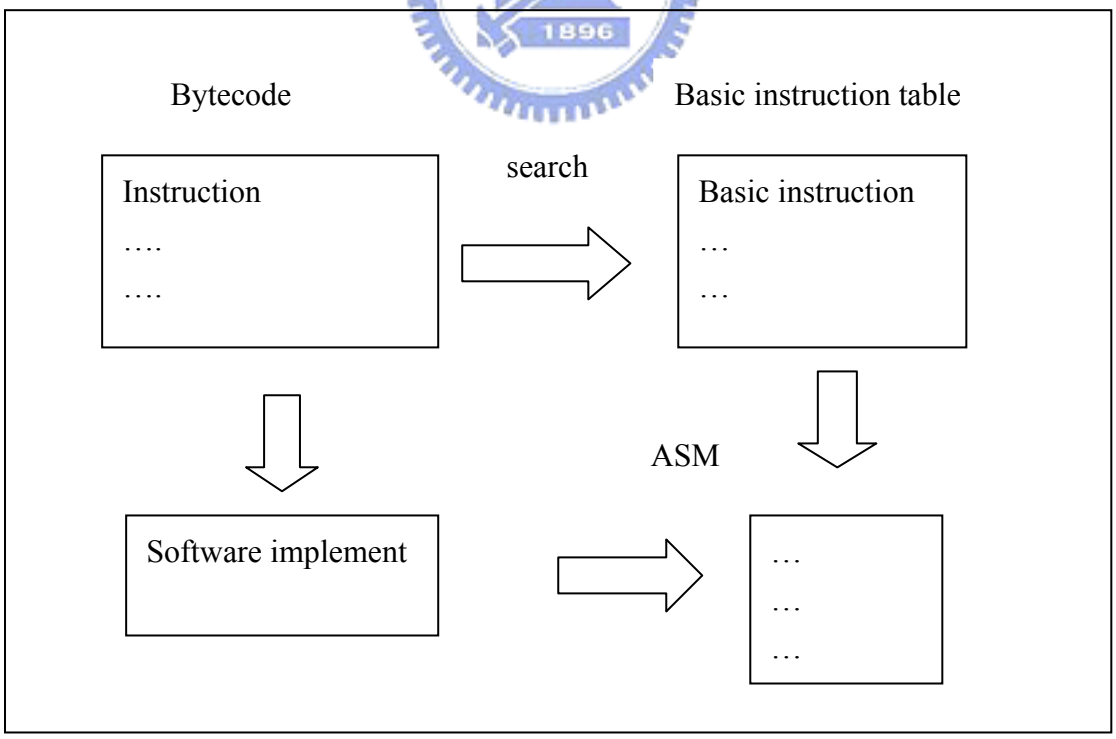

Figure 2.14 Searching the mapping in pass2

2.2 Basic Instruction Translation

Because Java bytecode instructions are stack base, we must adjust between the stack base instructions and register base instructions. The value which an instruction need is in the stack. When an instructions need value to operate, it must take the value from the stack. When operating an instruction produces a result, it must put the result into the stack. The following figure shows the instruction operating.

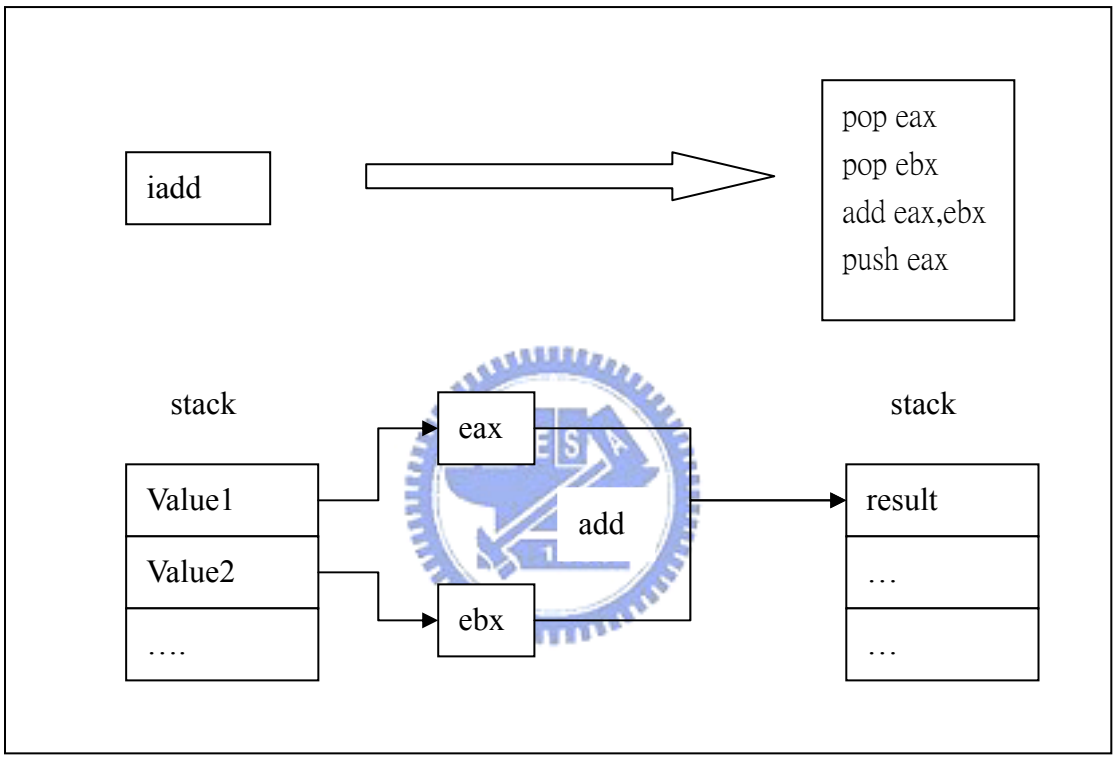

Figure 2.15 Simple bytecode translation

The table is in the Appendix B. The table is used to mapping the simple instruction which do not need to judge dynamically. In our system, when bumping into bytecode instruction, we first search the table to match the bytecode instruction. If it matches, we return the corresponding assembly instructions. If it does not match, we will translate the bytecode instruction dynamically.

2.3 Special Instruction Translation

Because some instruction must need to be translated dynamically, in this section, we will introduce the special instruction.

2.3.1 Long Operation

Because the long instruction in assembly is not supported, we must implement long instruction by ourselves. The long instructions are 64 bit based, so we must separate them into two 32 bit in our implementation. In ladd and lsub instruction, we must pay attention to the carry bit. We first add or sub the low bits and then use the adjust instruction adc and sbb which will help add the carry bit to add or sub high bits. The long addition and substraction are simple so we put these instructions in the basic instruction table.

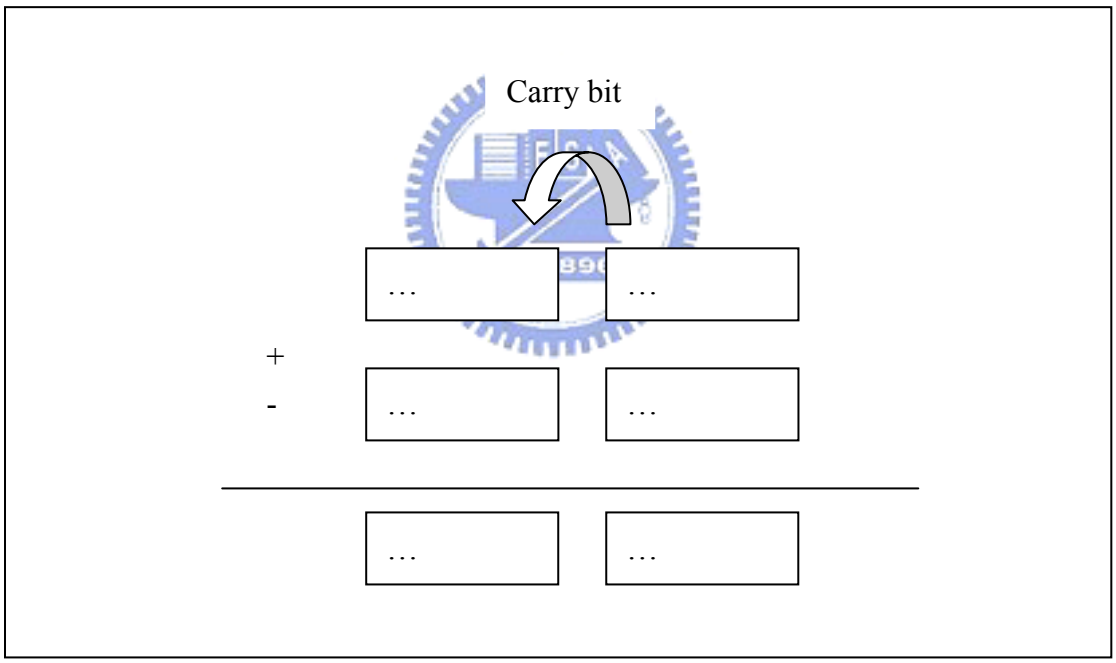

Figure 2.16 Long addition and substraction instruction

The long multiplication and division do not have any assembly instruction to help implementation. We use the multiplication algorithm[6] to implement. P、A and B are 64bit operands. The result is in P:A and before this algorithm, we must translate the long value into unsigned long value. After this algorithm, we translate the result value into signed value. Because this algorithm is only suitable for unsigned value, we must check the long value if the multiplier and multiplicand is negative to translate into unsigned value and judge the result value if it is positive or negative.

```
P=0
A=multiplier
B=multiplicand
Count=64
    While(count>0)
        If(LSB of A=1)
        Then
            P=P+B
            CF=carry generated by P+B
        Else
            CF=0
        End if
        Shift right CF:P:A by one bit position
    Count=count-1
    End while
```

Figure 2.17 Long multiplication algorithm

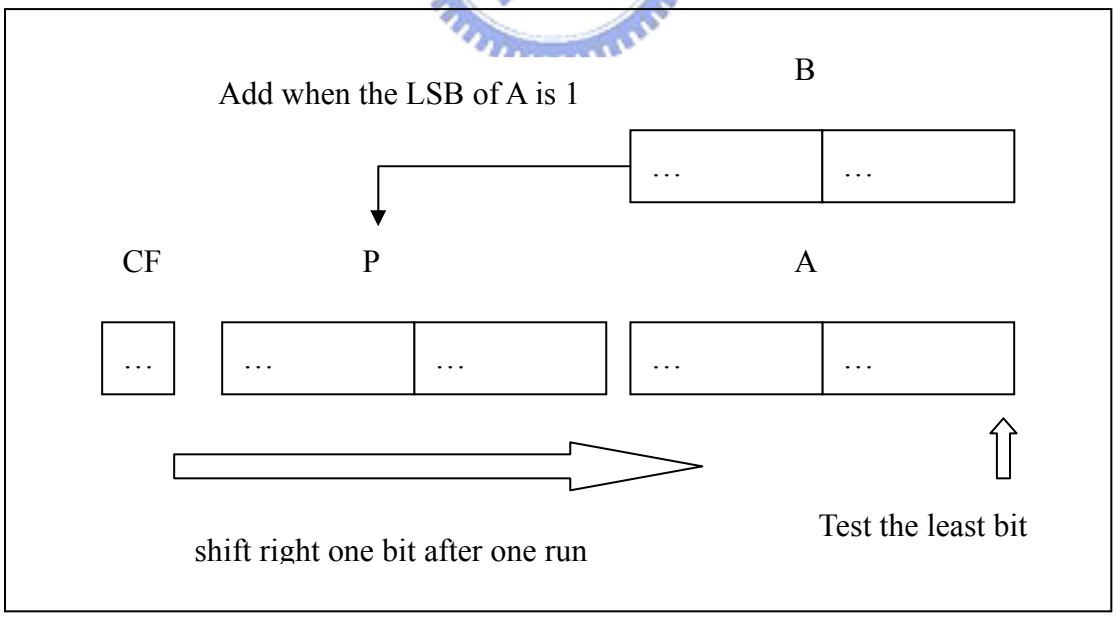

Figure 2.18 Long multiplication flow chart

There are several division algorithms to perform 64-bit unsigned long division. Here we describe and implement what is called the "non-restoring" division algorithm[6]. The division operation, unlike the multiplication operation, produces two results: a quotient and

a remainder. The algorithm consists of testing the sign of P and then, depending on the sign of P, either adds or subtracts B from P. Then P:A is shifted left while manipulating the rightmost bit of A. After repeating these steps 64 times, the quotient is in A and the remainder is in P.

```
P=0
A=dividend
B=divisor
Count=64
While(count>0)
    If(P is negative)
    Then
        Shift left P:A by one bit position
        P=P+B
    Else
        Shift left P:A by one bit position
        P=P-B
    End if
    If(P is negative)
    Then
        Set low-order bit of A to 0
    Else
        Set low-order bit of A to 1
    End if
Count=count-1
End while
If(P is negative)
    P=P+B
End if
```

Figure 2.19 Long division algorithm

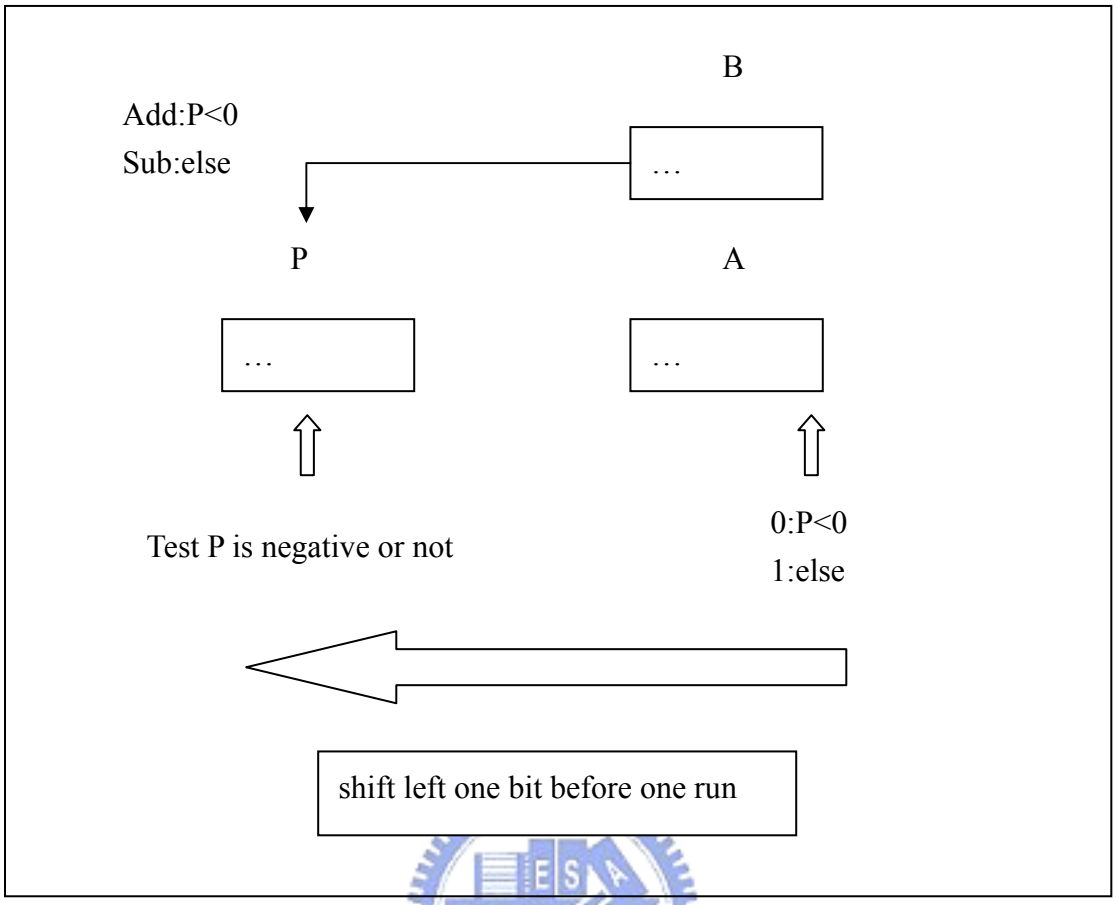

Figure 2.20 Long division flow chart

After addition, subtraction, multiplication and division, we discuss the remaining long instruction. The lneg instruction uses 2' complement to implement. The lshl and lshr use the shift instruction. The lor and lxor use or and xor instruction to implement.

2.3.2 Float Operation

In floating point instruction, there are two different types which are 32bit float and 64 bit double. In floating point arithmetic instruction, there are some instructions which are supported by assembly. In assembly, the floating point instructions use another special stack to operation. The special stack is in the FPU. The special stack in FPU is not a pure stack because the special stack is consisted of eight data registers, each 80 bits long. The special stack can convert any nonfloating formats including 64-bits data type to floating point, so the double instructions have been supported in assembly. These are only the 64-bit instruction supported by assembly. The addition,subtraction,multiplication and division instructions in executing float and double are very similar. The only difference is input data type. The float instructions use the 32 bit real4 data type supported by assembly. The double instructions use the 64 bit real 8 data type. By the way, we must pay attention

to real8 data type and Java stack data type, because the bit order of real8 data type is different from the Java stack. In the long 64 bit instruction, The top data of stack is high 32 bit and the second data of stack is low 32 bit, but in the real8 data type the low 32 bit puts the top data of stack and the high 32 bit puts the second data type.

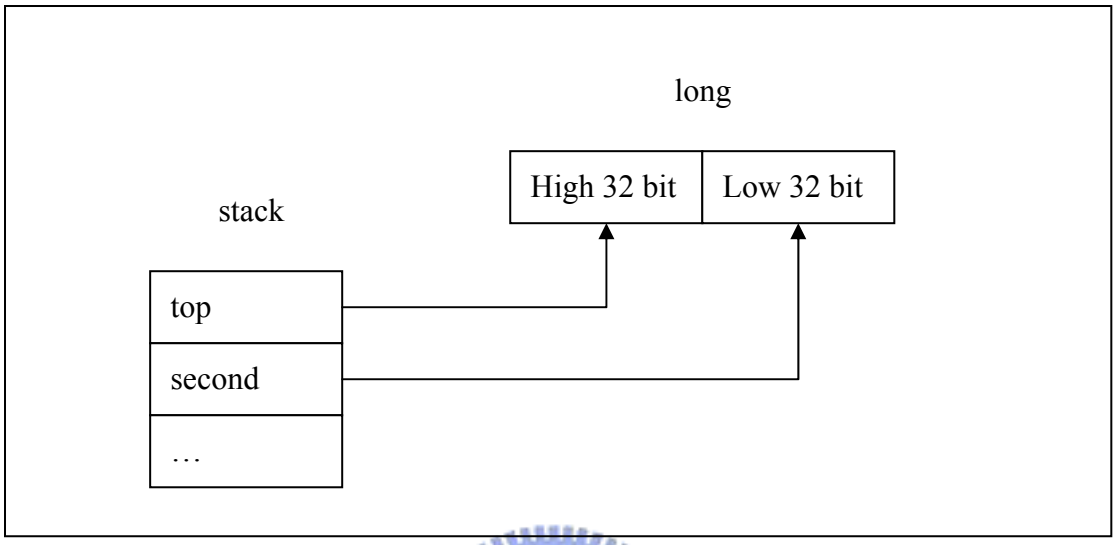

Figure 2.21 Long data type

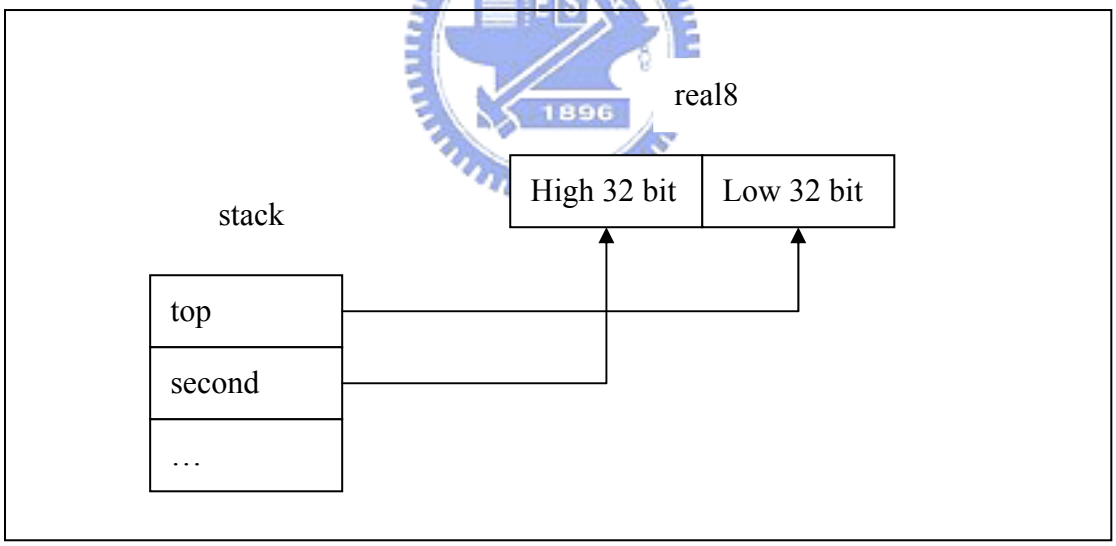

Figure 2.22 Real8 data type

The fadd in Java bytecode uses the fadd in assembly to implement. The fsub in Java bytecode uses the fsub in assembly to implement. The fmul in Java bytecode uses the fmul in assembly to implement. The fdiv in Java bytecode uses the fdiv in assembly to implement. The double arithmetic instructions in bytecode are translated like the float instructions, because the floating point is executed in the special stack we discuss above. We just change the real4 in float instruction into real8 for double instruction.

The frem and drem bytecode instructions are not the same as those of the so called remainder operation defined by IEEE 754. The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is not analogous to that of the usual integer remainder operatior. Instead, the Java virtual machine defines frem and drem to behave in a manner analogous to that of the Java virtual machine integer remainder instructions. [5]

The result of frem and drem instructions are governed by these rules：

(1) If either dividend or divisor is NaN, the result is NaN.

(2) If neither dividend nor divisor is NaN, the sign of the result equals the sign of the dividend.

(3) If the dividend is infinity or divisor is a zero or both, the result is NaN.

(4) If the dividend is finite and the divisor is infinity, the result equals the dividend.

(6) If the dividend is a zero and the divisor is finite, the result equals the dividend.

(7) In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder result form a dividend and divisor is defined by the mathmetical result=dividend-(divisor*q), where q is an intrger that is negative only if dividend/divisor is negative, and positive only if dividend/divisor is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathmetical quotient of dividend and divisor.

Despite the fact that division by zero may occur, evaluation of ferm and drem instructions never throw a runtime exception.

By the way, we must also pay attension to convert float to integer instruction. This instruction is defined in Java specification. The converting floating-point to integer instruction is generated by the following the rules in Java specification. [5]

(1) If the value is NaN, the result of the conversion is an int0.

(2) Otherwise, if the value is not infinity, it is rounded to an integer value V, rounding towards zero using IEEE 754 round towards zero mode. If this integer value V can be represented as an int, then the result is the int value V.

(3) Otherwise, either the value must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type int, or the value must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type int.

In the rule 2, we can use the special instruction which is supported by assembly. We put the floating-point value in the FPU and then use the fist to convert the floating-point to

integer. The FPU will help us to convert the floating-point to integer.

By the way, the following table is the format of NaN,negative infinity,positive infinity,negative value of large magnitude and positive value of large magnitude.

Value	**Float bits(sign exponent mantissa)**
+Infinity	0 11111111 00000000000000000000000
-Infinity	1 11111111 00000000000000000000000
NaN	0 11111111 10000000000000000000000
Largest positive (finite) float	0 11111110 11111111111111111111111
Largest negative (finite) float	1 11111110 11111111111111111111111

Value	**Int**
smallest representable value of type int	080000000h
the largest representable value of type int	07fffffffh

Valeu	Int
cwChop	01f72h

Figure 2.23 Special value in Java

Remain special instructions are discussed by my partner Jun Yuan Chen in Dynamic Dispatch Implementation based on Translating Java Bytecode to X86 Assembly.

Chapter 3

Handle Exception

Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred. When a method encounters an abnormal condition that it can't handle itself, it may throw an exception. Throwing an exception is like throwing a beeping, flashing red ball to indicate there is a problem that can't be handled where it occurred. Exceptions are caught by handlers positioned along the thread's method invocation stack. If the calling method isn't prepared to catch the exception, it throws the exception up to its calling method, and so on. If one of the threads of your program throws an exception that isn't caught by any method along the method invocation stack, that thread will expire.

3.1 Throw Exception

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw any object as an exception. Hwever, only those objects whose classes descend from Throwable can be thrown. Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw. A small part of this family is shown in flowing figure.

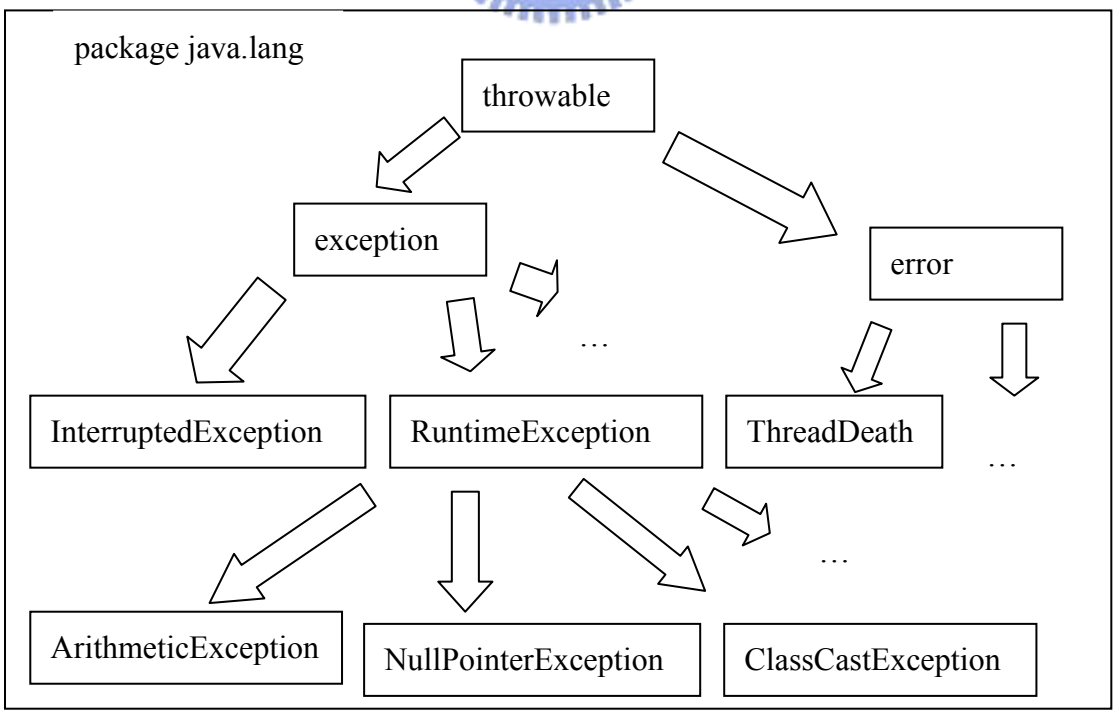

Figure 3.1 Exception class family

How do we throw Exception from our system to JVM? We use the ThrowNew function JNI supported to imply JVM that we want to throw an exception.

3.1.1 Throw Exception Java Support

How do we throw an exception by using ThrowNew function? The ThrowNew function needs class id and exception message, so we must call FindClass function to get class id and wirte message which you want to tell JVM before we call ThrowNew function.

In our system, we first need to find the class id when throwing an exception. Second, we get the initial method and create a new object of this exception. Then, we use initial method and object to call constructor. Third, we call GetObjectClass to get the class id from the object and then call ThrowNew to throw the exception with passing object id and null message. The flowing figures show how to translate Java code about throwing exceptions to Assembly exception simulation.

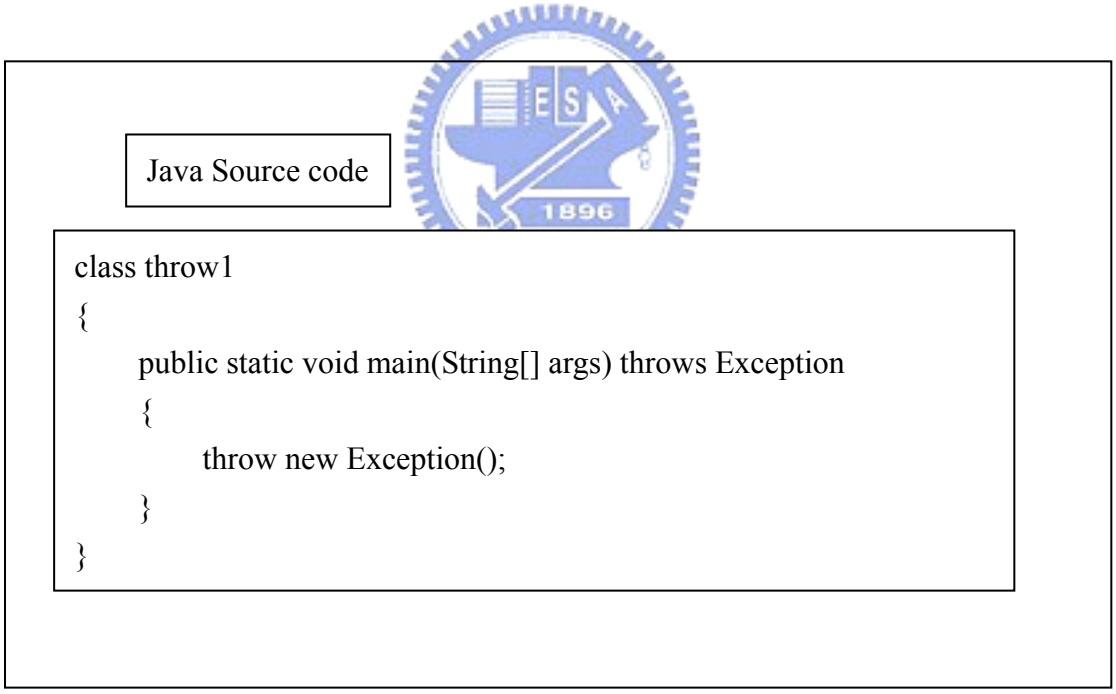

Figure 3.2 Throw exception example

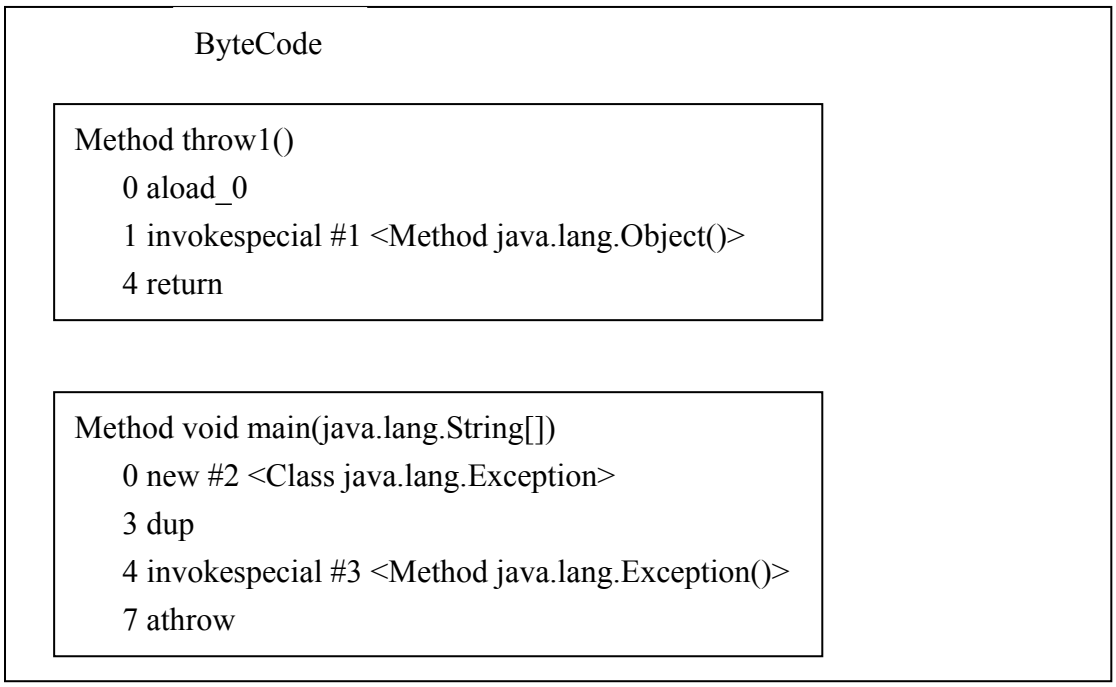

Figure 3.3 Throw exception example

When throwing an exception, the bytecode will have new, dup, invokespecial and athrow instructions. New instruction creates an object in stack. Dup instruction copies the top of stack and put them in the first position and second position of stack. Invokespecial instruction performs the constructor. Athrow throws exception to JVM.

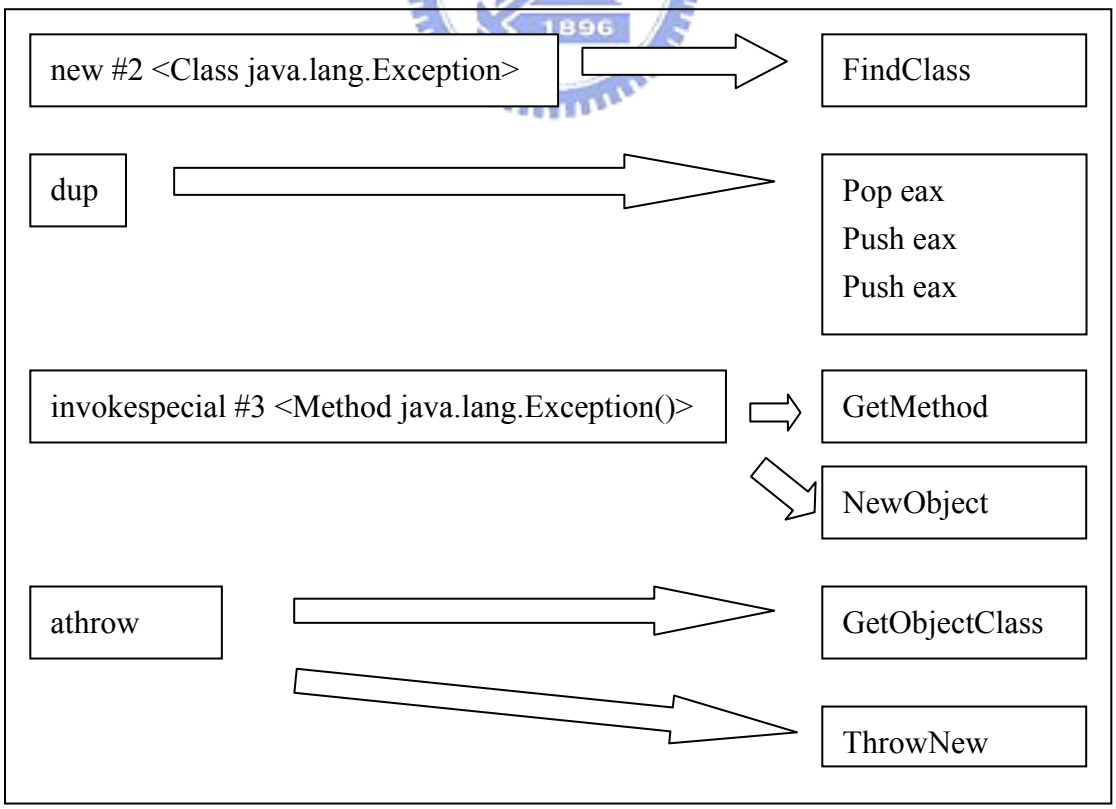

Figure 3.4 Throw exception mapping

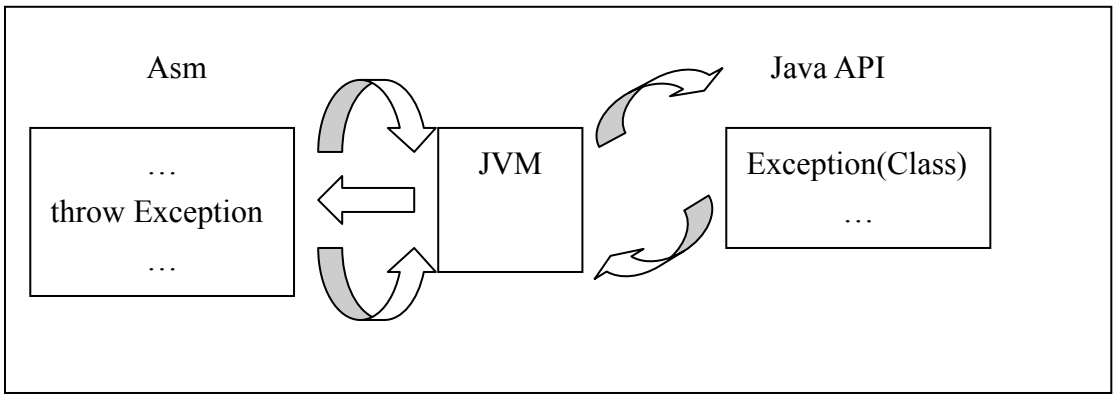

Figure 3.5 Throw exception flow chart

We translate new instruction into the jni function FindClass and then push the class id in stack. Invokespecial instruction is translated into GetMethod、NewObject and GetObjectclass, because invokespecial is used to call constructor and the jni function NewObject will operate the constructor. GetMethod is used to get initial method. When calling NewObject, we must pass an initial method id for parameter. The NewObject will create a newobject and use the initial method id to find constructor. The athrow instruction will be translated into GetObjectClass and ThrowNew because the top of stack is an object id. We need class id to call ThrowNew.

3.1.2 Throw Exception User Written

In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you only need declare it as a subclass of some member of the Throwable family. In general, however, the throwable classes that you define should extend class Exception.

Java Source code

```
class UserdefineException extends Exception
{
        …
}
class throw2
{
        public static void main(String[] args) throws Exception
        {
                throw new UserdefineException();
        }
}
```

Figure 3.6 Throw user exception example

ByteCode of throw2

```
Method throw2()
    0 aload_0
    1 invokespecial #1 <Method java.lang.Object()>
    4 return

Method void main(java.lang.String[])
    0 new #2 <Class UserdefineException>
    3 dup
    4 invokespecial #3 <Method UserdefineException()>
    7 athrow
```

Figure 3.7 Throw user exception main bytecode

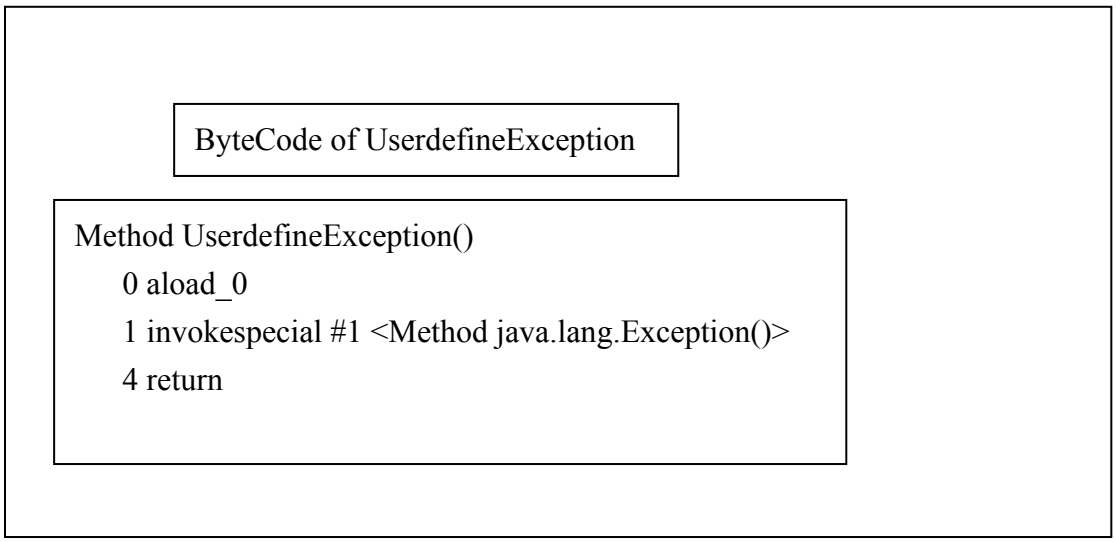

Figure 3.8 Throw user exception class bytecode

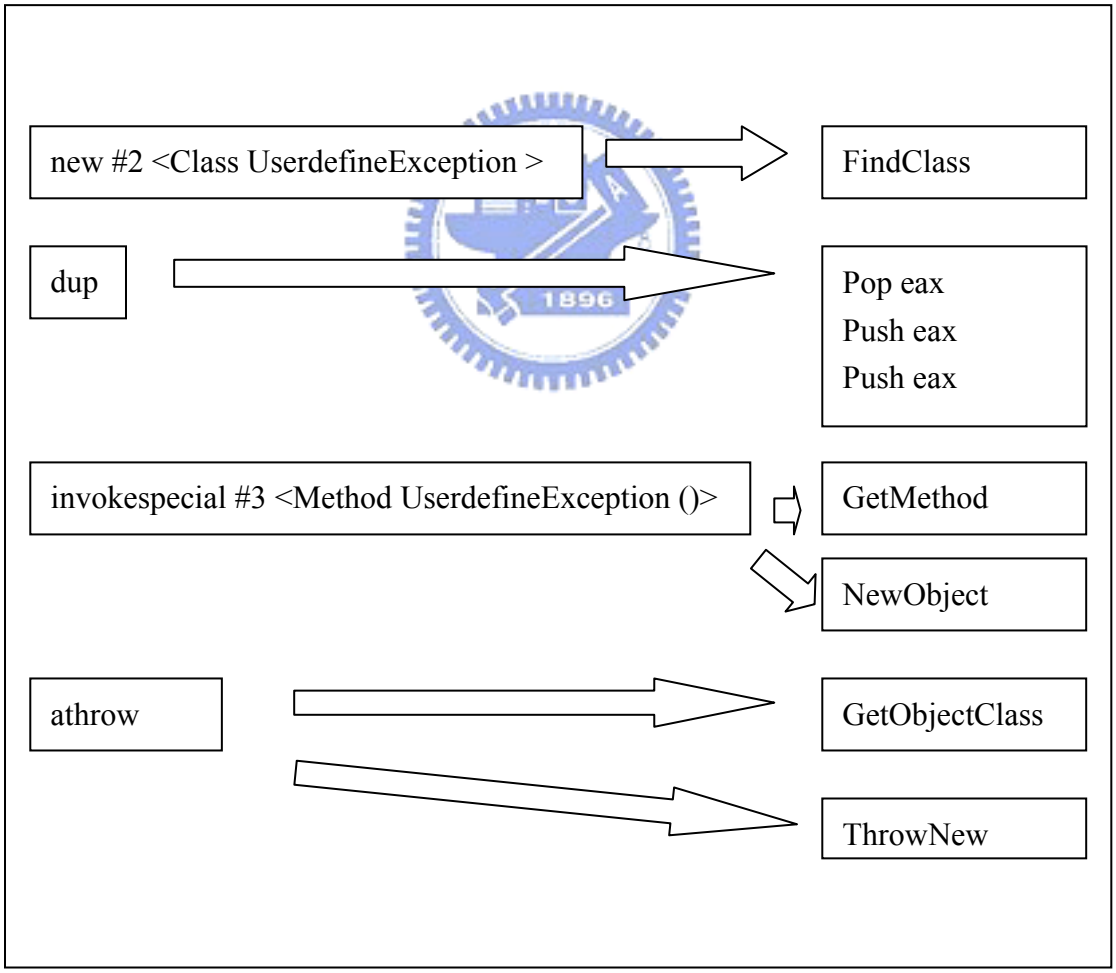

Figure 3.9 Throw user exception mapping

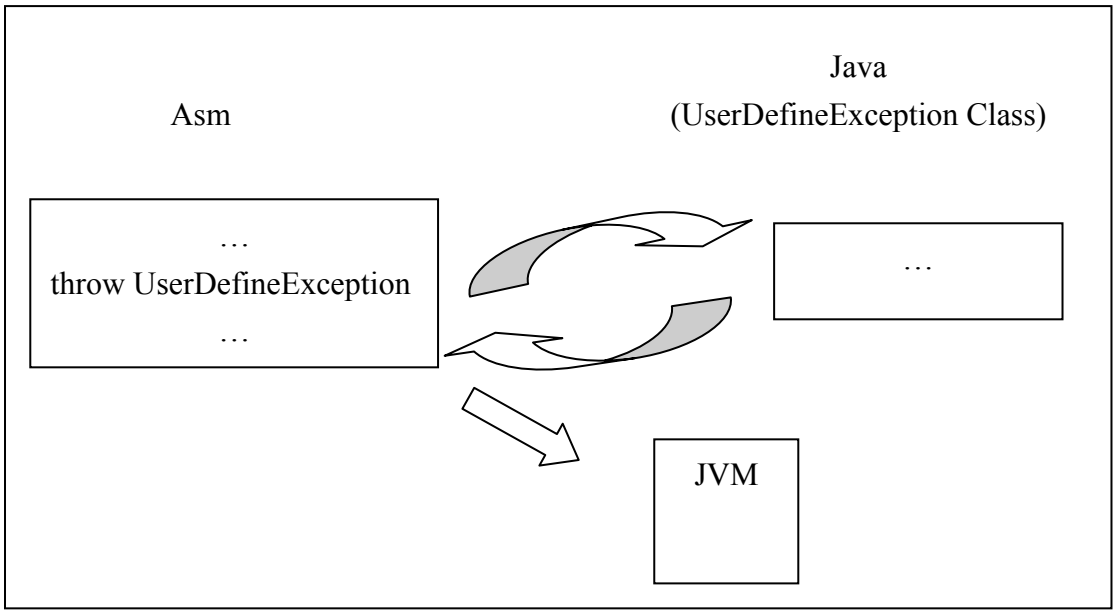

Figure 3.10 Throw user exception flow chart

Throwing exception written by user is mostly like throwing exception Java support. The reason is that user writes Java code when throwing exception defined by user. Usually, the exception class that user wants to throw only has constructor. So we do not translate exception class into assembly. We just use FindClass, GetMethod and NewObject to operate the constructor.

3.1.3 Describe Exception

The above section tells us how to throw an exception to JVM, but when JVM catches the exception, the JVM will do what they must to do. In Java code, when we operate throw instruction without try catch case, JVM only prints the exception message on the screen. So we use ExceptionDescribe and ExceptionClear to print exception message and then clear exception from JVM. ExceptionDescribe and ExceptionClear are supported by JNI.

In our system implementation, we add a label Exhandle, ExceptionDescribe and ExceptionClear in main assembly code. Except main assembly code, we only add the label Exhandle. When called class has thrown an exception, the called class need not print exception on the screen and clear the exception in JVM. Because the calling class may handle the called class exception, ExceptionDescribe and ExceptionClear will now be called repeatedly. The label Exhandle is for jumping when we check exception which has been thrown in JVM.

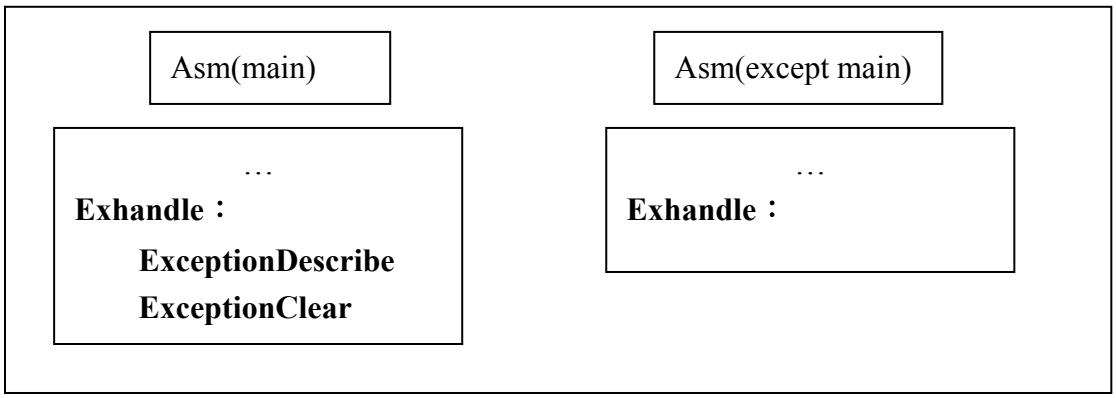

Figure 3.11 Check exception

3.2 Check Exception

The exception check is very important that can tell users where the exception is. In JNI, we use ExceptionCheck function to determine if JVM has exception.

In Java implementation, we must add instruction checking exception after every instruction. Now we can reduce some checking exception instructions because some checking instructions are not needed. In our implementation, we add exception check after the call instruction. When calling Java system class or userdefine class, the called class may throw exception. Because called class usually throw exception in try catch case.

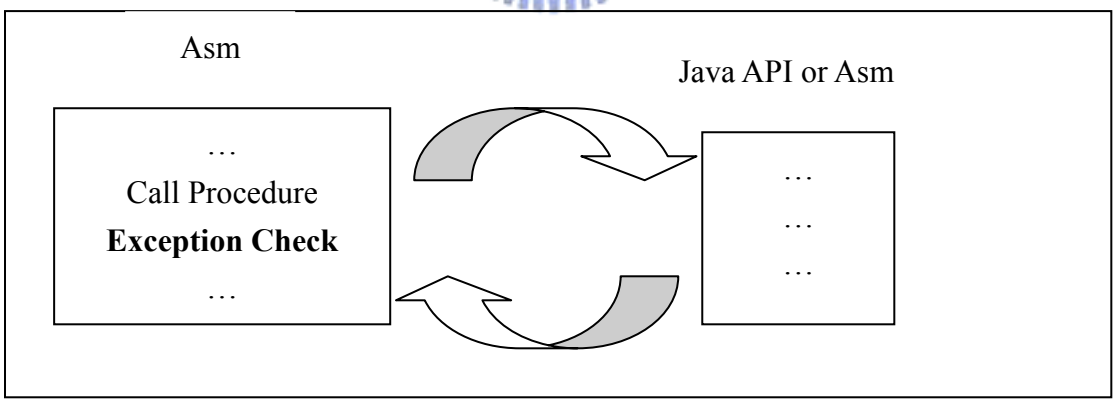

Figure 3.12 Check exception after call

Sometimes, exception check must be used recursively and factorially.

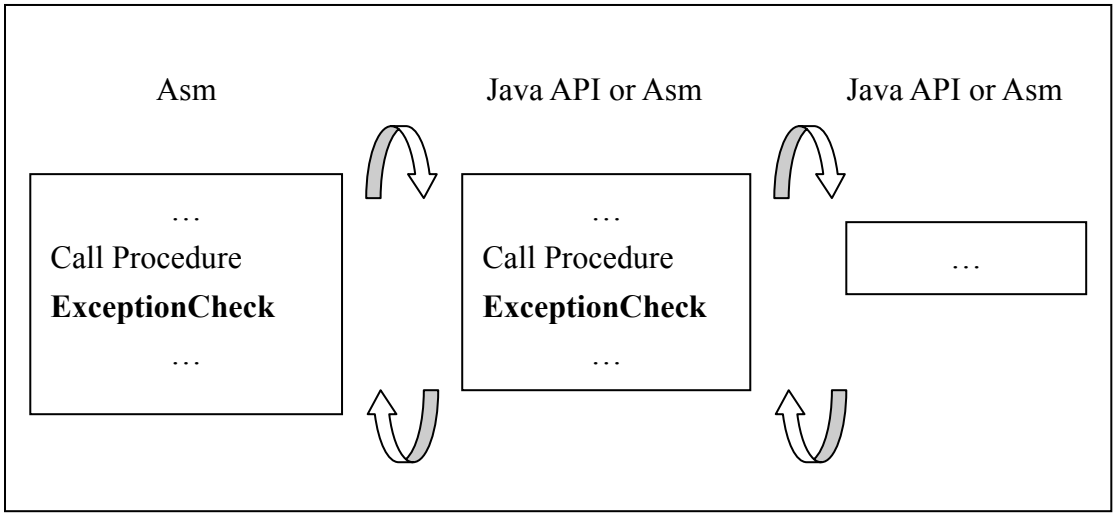

Figure 3.13 Check exception example

Also, we add exception check after throw exception instruction. When operating Java code which has exception, JVM will break the operation. Because the instructions after throw exception instruction are not operated by JVM, we check exception and if find an exception, our system operation will jump to Exhandle.

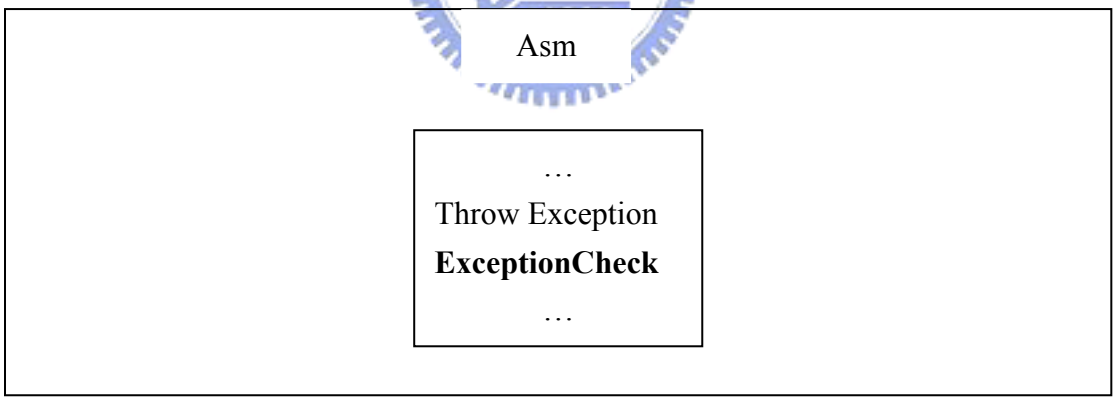

Figure 3.14 Check exception after throw

In Java, the JVM only handle divide by zero exception of arithmetic instruction. Java does not check overflow and underflow automatically. So the exception check of divide by zero exception is not the same as above exception check. Above exceptioncheck is using ExceptionCheck function of JNI, but we must add some assembly code by myself in this case before ExceptionCheck. In 32 bits operands, we can easily check divide by zero. In 64 bits operands, we must separate into two 32 bit parts and check the parts. The divide by zero exception is checked except float-point division.

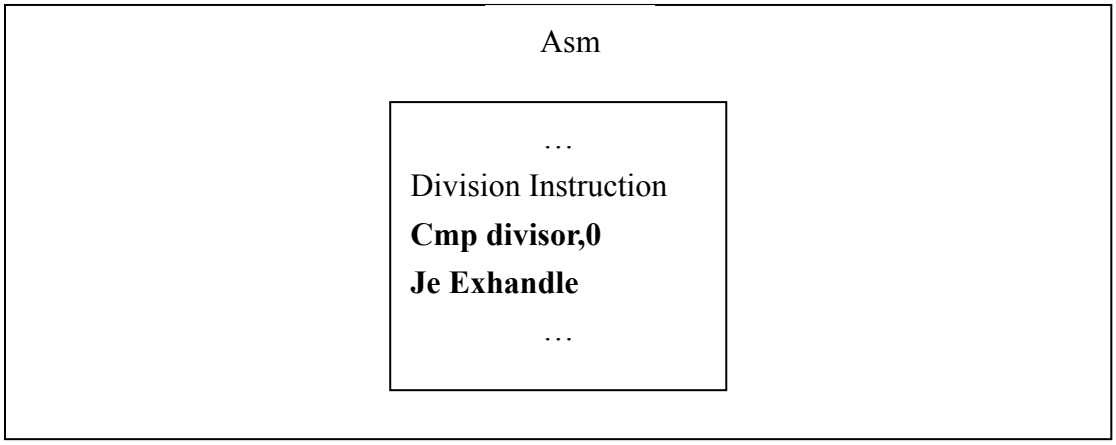

Figure 3.15 Check exception before division

3.3 Handle Try, Catch and Finally

Java support try, catch and finally to handle exception. It is very important to support user to handle exception by itself. When bumping into occurring exception, user has control to change the operating path to another operating path.

3.3.1 Under Try and Catch Situation

How does our system simulate try and catch? First, we must understand how try block operate. In try block, when some instruction has exception, the JVM must search the exception table to find the matching exception and get the catch block address. Then, the operating control will jump to the catch block of the exception which is thrown in try catch. So the JVM must have an exception table which has all exceptions defined in Java source code. The exception table is built in compiling time. Using javap instruction can show the exception table. The exception table has all exceptions defined in Java source code and the mapping catch block address.

Our system must insert another instruction in exception check instructions. We insert the jump instruction which check if matching the thrown exception. If matching the exception in exception table, the operating instruction jump the mapping catch block. So we must build an exception table in assembly which is translated by our system and when exception occur, we must search the exception table which is translated in assembly to find the mapping catch block address. In try and catch situation, the exception check instructions is different from above simple exception check. First, we check exception whenever it occurs. Second, if exception occurs, we search the matching exception in exception table. Third, if

we match the exception in exception table, we jump to the mapping catch block address. Else if we do not match, we jump to the label exhandle to print the exception on the screen.

In the beginning of the assembly which our system translates, we build exception table. We use structure to save the exception information in assembly. The information in exception include from、to、target and type. The label from and to means from some instruction to some instruction is protected by try block. The label target means jump to the beginning instruction of the catch block. The label type save the exception class id which can be matched when search the exception table. Why do we save the exception class id in the label type? This is because we can use the JNI function IsInstanceOf when searching table. The JNI function IsInstanceOf test whether the object is an instance of a class or an interface. The IsInstanceOF will be discussed when searching exception table.

Java Source Code

```
class trycatch
{
    public static void main(String[] args) throws Exception{
            int i=0;
        try
        {
            throw new Exception();
        }
        catch(Exception x)
        {
            i=1;
        }
        System.out.println(i);
    }
}
```

Figure 3.16 Try catch example

Bytecode of trycatch

```
Method trycatch()
    0 aload_0
    1 invokespecial #1 <Method java.lang.Object()>
    4 return
```

```
Method void main(java.lang.String[])
    0 iconst_0
    1 istore_1
    2 new #2 <Class java.lang.Exception>
    5 dup
    6 invokespecial #3 <Method java.lang.Exception()>
    9 athrow
   10 astore_2
   11 iconst_1
   12 istore_1
   13 goto 16
   16 getstatic #4 <Field java.io.PrintStream out>
   19 iload_1
   20 invokevirtual #5 <Method void println(int)>
   23 return
```

```
Exception table:
    from    to  target type
      2     10     10   <Class java.lang.Exception>
```

Figure 3.17 Try catch example bytecode

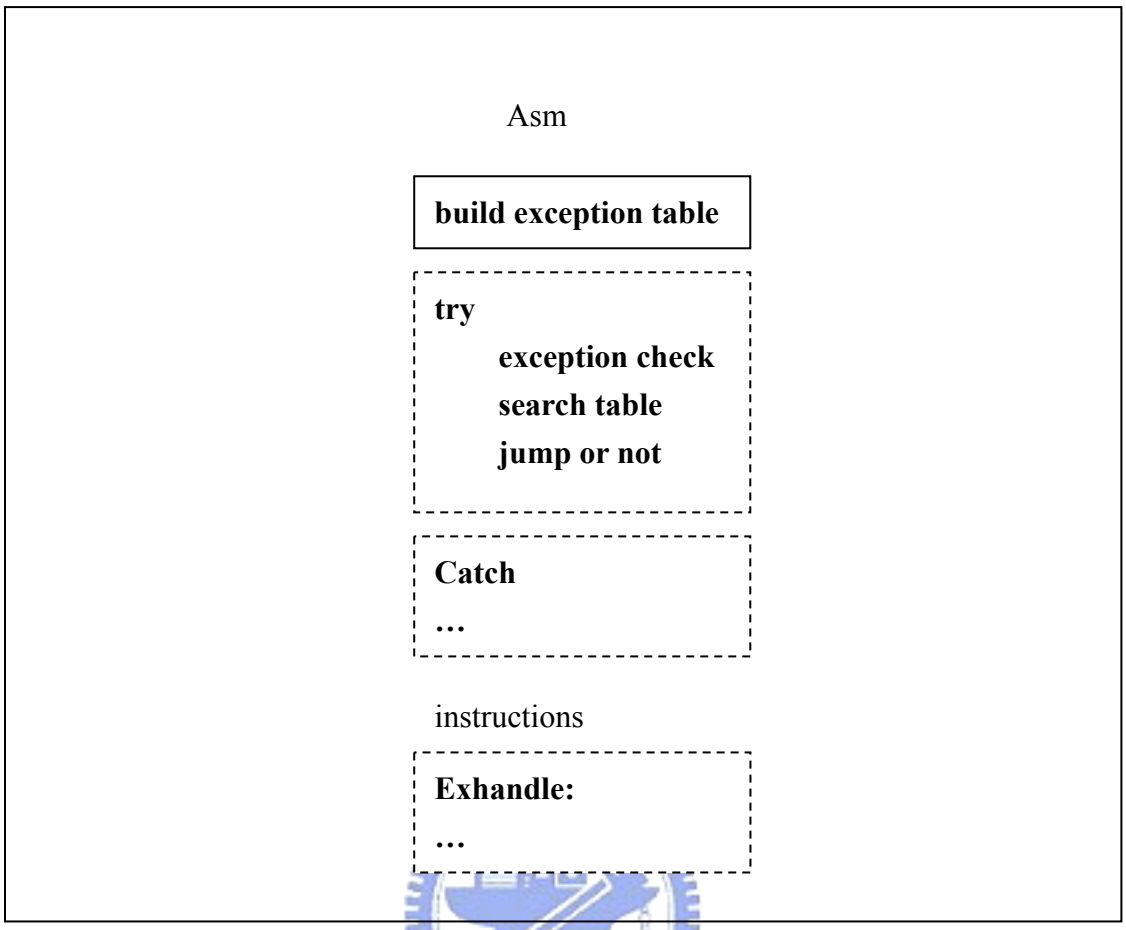

Figure 3.18 Try catch in Assembly

We use directive struc to define an exception table in assembly and dynamically build the exception table through the pass 1 information in run time. The search table function will get exception object from JVM through JNI function ExceptionOccurred. Then, we first check the instruction address which throw exception if match the scope of from and to in every index of exception table. If matching the scope, we will call the IsInstanceOf function to check if the exception object is the inheritance of the exception class which we get through matching the scope of from and to. If the exception object matches the scope of address and inheritance of exception class, the search table function will return the target address to imply where the catch block is.

Directive of Exception Table

```
exceptiontable      struc
      from      dword     ?
      to        dword     ?
      target    dword     ?
      type      dword     ?
exceptiontable      ends
```

Body of Buile Exception Table

```
      mov     esi,offset allexception
      mov     (exceptiontable ptr [esi]).from,2
      mov     (exceptiontable ptr [esi]).to,10
      mov     (exceptiontable ptr [esi]).target,10
      push    offset exception_0
      push    jnienv
      GetFnPtr    fntblptr,6,fnptr
      call    [fnptr]
      mov     (exceptiontable ptr [esi]).type,eax
```

Figure 3.19 Build exception table in Assembly

Beginning of Search Table Function

```
push      jnienv
GetFnPtr      fntblptr,15,fnptr
call      [fnptr]
mov       exobj,eax
```

Body of Search Table Function

```
.while      edi<exnum
   mov      ebx,(exceptiontable ptr [esi]).from
   mov      ecx,(exceptiontable ptr [esi]).to
   .if      (ebx<=index)&&(index<ecx)
        mov      edx,(exceptiontable ptr [esi]).type
        push     edx
        push     exobj
        push     jnienv
        GetFnPtr      fntblptr,32,fnptr
        call      [fnptr]
        .if      eax==1
             mov      edx,(exceptiontable ptr [esi]).target
             push     edx
             GetFnPtr      fntblptr,17,fnptr
             push     jnienv
             call     [fnptr]
             .break
        .endif
   .endif
   add      esi,type exceptiontable
   inc      edi
.endw
```

Figure 3.20 Search exception table in Assembly

3.3.2 Under Try and Finally Situation

The try and finally situation in Java is almost like calling a subroutine. So finally block has ret instruction and try block has jsr instruction like the call instruction. The jsr instruction will indicate the address of finally block. When bumping to jsr instruction, it will store the next instruction address to let the finally jump back and then will jump to the finally block address. In the end of finally block, there is an ret instruction which help return to the try block address stored in local variable. In the try and finally situation, the Java compiler will insert added catch block to protect the try block. Because the try block may throw exception and the finally instruction must be operated after operating try instruction, the catch block do the athrow and jsr instruction to insure that throw exception and call the finally block. So the exception table has any type that throws any exception.
When try block throws exception to JVM, it will jump to the added catch block. The catch block will use jsr to call finally block and then use athow instruction to throw exception that thrown in try block to JVM. So we must change the athrow instruction into ExceptionOccurred and ThrowNew to handle this catch block situation.

In our implementation, jsr is translated into only jump instruction. Why do we not store the address of next instruction ? Because in pass 1 we store the BRANCH_TABLE, it stores the address that some instruction will jump to. We use it to check where the instruction jumps to. The ret instruction will be translated into compare the local variable value with the index of BRANCH_TABLE in the end of finally block.

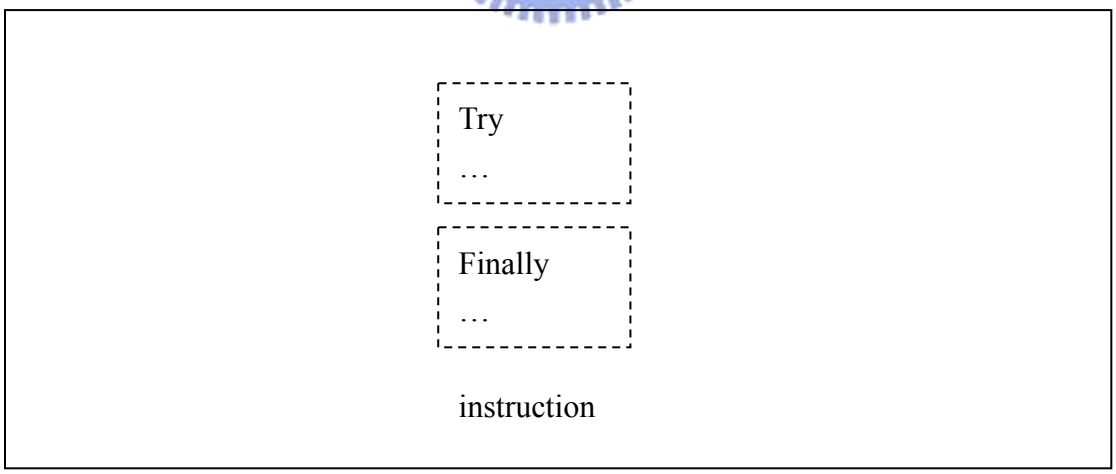

Figure 3.21 Try finally flow chart

Java code

```
class tryfin
{
	public static void main(String[] args) {
		int i=0;
		try {
			i++;
		}
		finally {
			i++;
		}
		System.out.println(i);
	}
}
```

Figure 3.22 Try finally example

```
Method tryfin()
    0 aload_0
    1 invokespecial #1 <Method java.lang.Object()>
    4 return
```

```
Method void main(java.lang.String[])
    0 iconst_0
    1 istore_1
    2 iinc 1 1
    5 jsr 17
    8 goto 23
   11 astore_2
   12 jsr 17
   15 aload_2
   16 athrow
   17 astore_3
   18 iinc 1 1
   21 ret 3
   23 getstatic #2 <Field java.io.PrintStream out>
   26 iload_1
   27 invokevirtual #3 <Method void println(int)>
   30 return
```

Exception table of tryfin

```
Exception table:
    from    to  target type
       2     8      11   any
      11    15      11   any
```

Figure 3.23 Try finally examble bytecode

jsr instruction

```
mov     eax,8
push    eax
jmp     aa_17
```

ret instruction

```
mov     eax,[aa_local_vars+3*4]
cmp     eax,8
je      aa_8
```

Build exception table

```
mov     esi,offset allexception
mov     (exceptiontable ptr [esi]).from,2
mov     (exceptiontable ptr [esi]).to,8
mov     (exceptiontable ptr [esi]).target,11
mov     eax,0
mov     (exceptiontable ptr [esi]).type,eax
add     esi,type exceptiontable
mov     (exceptiontable ptr [esi]).from,11
mov     (exceptiontable ptr [esi]).to,15
mov     (exceptiontable ptr [esi]).target,11
mov     eax,0
mov     (exceptiontable ptr [esi]).type,eax
```

Search exception table

```
mov     edx,(exceptiontable ptr [esi]).type
.if     edx==0
    push    edx
    .break
.endif
```

Figure 3.24 Try finally mapping Assembly code

3.3.3 Under Try, Catch and Finally Situation

The following is the try、catch and finally example. In this situation, we must pay attention to the exception table. Because the exception will appear the exception types and any type, we must combine the two types. When bumping into exception, we will first check the exception. If we do not find the correct the exception type, we will take it as the any type.

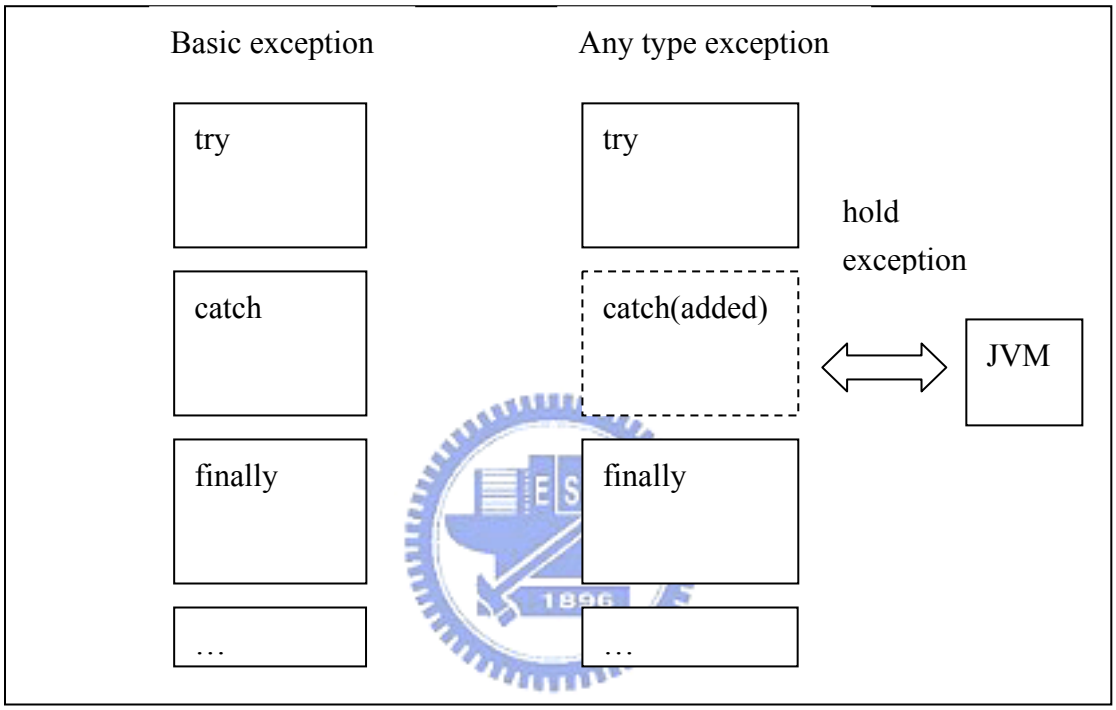

Figure 3.25 Two types of catch

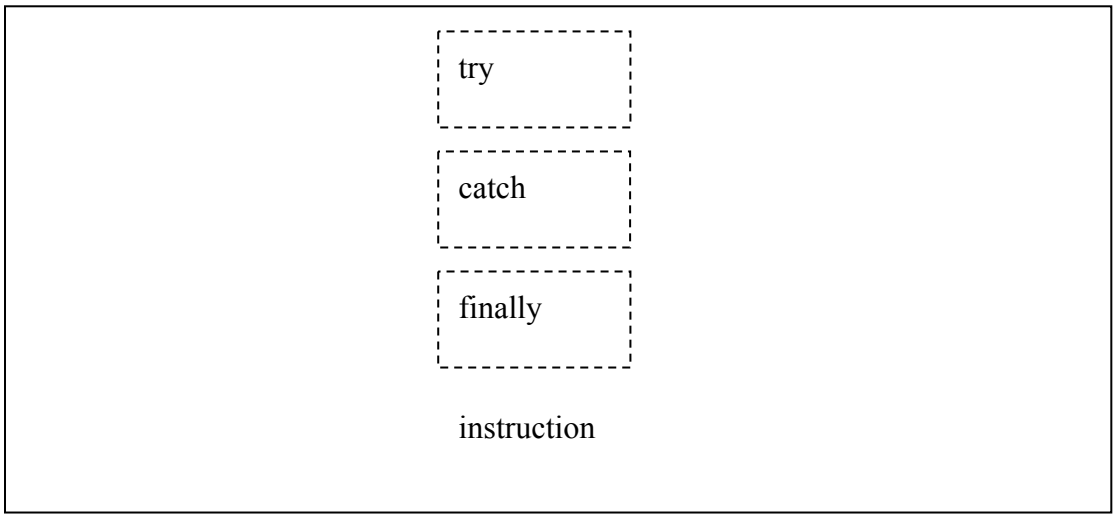

Figure 3.26 Try catch finally flow chart

```
class trycatchfin
{
	public static void main(String[] args)throws Exception
	{
		int i=0;
		try
		{
			throw new Exception();
		}
		catch(Exception e)
		{
			i++;
		}
		finally
		{
			i++;
		}
		System.out.println(i);
	}
}
```

Figure 3.27 Try catch finally example

```
Method trycatchfin()
    0 aload_0
    1 invokespecial #1 <Method java.lang.Object()>
    4 return

Method void main(java.lang.String[])
    0 iconst_0
    1 istore_1
    2 new #2 <Class java.lang.Exception>
    5 dup
    6 invokespecial #3 <Method java.lang.Exception()>
    9 athrow
   10 astore_2
   11 iinc 1 1
   14 jsr 26
   17 goto 33
   20 astore_3
   21 jsr 26
   24 aload_3
   25 athrow
   26 astore 4
   28 iinc 1 1
   31 ret 4
   33 getstatic #4 <Field java.io.PrintStream out>
   36 iload_1
   37 invokevirtual #5 <Method void println(int)>
   40 return
```

```
Exception table:
  from    to  target type
     2    10      10   <Class java.lang.Exception>
     2    17      20   any
    20    24      20   any
```

Figure 3.28 Try catch finally bytecode

Chapter 4

Performance Analysis

We check the speed of throwing exception. In our example, we test the 1000000 loop to throw 1000000 exceptions. Following figure is our test example.

```
for(h=0;h<1000000;h++)
        {
        try
        {
                throw new Exception();
        }
        catch(Exception e)
        {
        }
}
```

Figure 4.1 Exception test

The speed is followed.

Java interpreter	JIT compiler	Our system
4.41	4.31	5.82
4.47	4.58	5.62
4.42	4.02	5.90

Table 4.1 Exception rate

In the above table, we can find the Java interpreter and JIT compiler hava almost the same time. We can suppose that the JIT compiler do not do any optimization when throwing exception. In our system, the time is also the same as the above two compilers. Because our system uses the jni function, our performance is slow down by calling JVM. However, the example is not general. The slow time does not represent our system is much slower than Java compiler, because the loop is too big. Under the 100 loop, our system will much faster than another one because in our system, the searching exception table is faster.

The following table is showing the arithmetic instruction operating time. Each item is executed under 100000000 times

Benchmark	Interpreter	JIT	Asm code
Addition and subtraction on integer	13.47	0.84	2.05
Multiplication and division on integer	16.5	3.7	5.08
Addition and subtraction on long	14.41	0.72	5.52
Multiplication and division on long	22.09	9.88	105.99
Addition and subtraction on float	11.67	1.81	3.69
Multiplication and division on float	12.39	2.44	4.86
Addition and subtraction on double	14.66	1.69	11.66
Multiplication and division on double	15.58	3.19	12.73

Table 4.2 Arithmetic instruction rate

The time of assembly code translated by our system is between the interpreter and JIT compiler. That is because the JIT do some optimization dynamically at the loop. In the table, we must pay attention to the long multiplication and division. The time in long multiplication and division is even slower than interpreter. In our search, we can approximately know the long multiplication and division instructions in JVM do another operation and not the software implementation. We doubt that the JVM can translate the long data into double data and use the double value to do arithmetic operation. After arithmetic operation, the double result value will be translated back into long.

Chapter 5

Conclusion and Future work

5.1 Conclusion

Performance of Java is always an interesting issue. In the thesis, we implement a system to translate the Java bytecode to assembly code to speed Java program operation. Even in some case, our performance is faster than JIT compiler. Because the JIT can translate the Java Library to native code and do optimization at Java library, our system will be slower than JIT compiler in calling many library functions.

Our system can simulate the Java operation. This issue can help us to know how the Java to operation. Because Java protect does not support translating Java bytecode to assembly code, we implement a system to translate Java bytecode to assembly code to help tracing the java code.

5.2 Future Work

Due to the limitation of time, only a part of the idea is implemented. Our system does not support the recursive program because in translating recursive function we must do extra local variables to store the temp variables. In recursive program, the operation time will be much slower and the memory will be much bigger. So the recursive program is the future work.

In optimization, we only do the temporal match. This is the local optimization. In future work we can do the global optimization. We can translate the Java bytecode to intermediary instruction to do global optimization and draw the flow chart to find the data block to judge when to keep or erase.

Reference

[1] **Platform independence issues in compiling Java bytecode to native code**

Ye Hua; Tong WeiQin; Yao WenSheng;

High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on ,Volume: 1 , 14-17 May 2000

Pages:530 - 532 vol.1

[2] **Java bytecode to native code translation: the Caffeine prototype and preliminary results**

Hsieh, C.-H.A.; Gyllenhaal, J.C.; Hwu, W.W.;

Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on , 2-4 Dec. 1996

Pages:90 – 97

[3] **Using data flow analysis to infer type information in Java bytecode**

Maggi, P.; Sisto, R.;

Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on , 10 Nov. 2001

Pages:211 – 222

[4] **Bytecode optimization**

Babic, D.; Rakamaric, Z.;

Information Technology Interfaces, 2002. ITI 2002. Proceedings of the 24th International Conference on , 24-27 June 2002

Pages:377 - 382 vol.1

[5] **The Java Virtual Machine Specification (Second Edition)**

Tim Linholm; Frank Yellin

First printing, April 1999

[6] **Computer organization and design:the hardware / software interface**

David A; Patterson;John L. Hennessy;

Morgan Kaufmann Publishers,1994

[7] **80x86 Assembly Language and Computer Architecture**

Richard C. Detmer

Jones and Bartlett Computer Sciene,2001

[8] **ASSEMBLY LANGUAGE FOR INTEL-BASED COMPUTERS(3rd EDITION)**

Kip R. Irvine

Prentice-Hall, Upper Saddle River, New Jersey 07458

[9] **INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING**
Sivaram P. Dandamudi
Springer,1998

[10] **The Java Native Interface Programmer's Guide and Specification**
Sheng Liang,
Addison-Wesley, 3rd Printing December 1999

[11] **JAVA 虛擬機器**
Jon Meyer & Troy Downing 著
蔡寶進譯,美商歐萊禮有限公司,2000 年 7 月初版

[12] **Dependence analysis of Java bytecode**
Jianjun Zhao;
Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International , 25-27 Oct. 2000
Pages:486 - 491

[13] **Java runtime systems: characterization and architectural implications**
Radhakrishnan, R.; Vijaykrishnan, N.; John, L.K.; Sivasubramaniam, A.; Rubio, J.; Sabarinathan, J.;
Computers, IEEE Transactions on , Volume: 50 , Issue: 2 , Feb. 2001
Pages:131 - 146

[14] **Design, and implementation of a Java execution environment**
Chen, F.G.; Ting-Wei Hou;
Parallel and Distributed Systems, 1998. Proceedings., 1998 International Conference on , 14-16 Dec. 1998
Pages:686 - 692

[15] **Parameter passing for the Java virtual machine**
Gough, K.J.;
Computer Science Conference, 2000. ACSC 2000. 23rd Australasian , 31 Jan.-3 Feb. 2000
Pages:81 - 87

[16] **Java Virtual Machine performance analysis with Java instruction level parallelism and advanced folding scheme**
Kim, A.; Chang, M.;
Performance, Computing, and Communications Conference, 2002. 21st IEEE International , 3-5 April 2002
Pages:9 – 15

APPENDIX A

```
#include <jni.h>
#include <stdio.h>
#include <windows.h>

typedef jint (JNICALL CreateJavaVM_t)(JavaVM **pvm, void **env, void *args);

#define PATH_SEPARATOR ';' /* define it to be ':' on Solaris */
#define USER_CLASSPATH "." /* where Prog.class is */

JNIEnv*    startvm();

JNIEnv*    startvm() {

        CreateJavaVM_t *CreateJavaVM;
        HINSTANCE hvm;

        JNIEnv *env;
        JavaVM *jvm;
        jint res;
        jclass cls;
        jmethodID mid;
        jstring jstr;
        jclass stringClass;
        jobjectArray args;
#ifdef JNI_VERSION_1_2
        JavaVMInitArgs vm_args;
        JavaVMOption options[1];
        options[0].optionString = "-Djava.compiler=NONE";
        //options[1].optionString = "-Djava.class.path=c:\\j2sdk1.4.1_01";
        options[1].optionString = "-Djava.class.path=.";
        //options[2].optionString = "-Djava.library.path=c:\\j2sdk1.4.1_01\\lib";
        //options[3].optionString = "-verbose:jni";
        //vm_args.version = 0x00010002;
        vm_args.version = JNI_VERSION_1_4;
        vm_args.options = options;
        vm_args.nOptions = 2;
```

```
        vm_args.ignoreUnrecognized = JNI_TRUE;
/* Create the Java VM */

      hvm=LoadLibrary("c:\\j2sdk1.4.1_02\\jre\\bin\\client\\jvm.dll");
      if(hvm == NULL)
      {
       return NULL;
      }
      CreateJavaVM = (CreateJavaVM_t *)GetProcAddress(hvm,"JNI_CreateJavaVM");

        //res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

        res = CreateJavaVM(&jvm, (void **)&env, &vm_args);

#else
        JDK1_1InitArgs vm_args;
        char classpath[1024];
        vm_args.version = 0x00010001;
        JNI_GetDefaultJavaVMInitArgs(&vm_args);
        /* Append USER_CLASSPATH to the default system class path */
        sprintf(classpath, "%s%c%s",
                        vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH);
        vm_args.classpath = classpath;
/* Create the Java VM */
        res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
#endif /* JNI_VERSION_1_2 */

        if (res < 0) {
                fprintf(stderr, "Can't create Java VM\n");
                exit(0);
        }

        return env;

}
```

APPENDIX B

iconst_m	mov eax,-1 push eax
iconst_0	mov eax,0 push eax
iconst_1	mov eax,1 push eax
iconst_2	mov eax,2 push eax
iconst_3	mov eax,3 push eax
iconst_4	mov eax,4 push eax
iconst_5	mov eax,5 push eax
lconst_0	mov eax,0 push eax push eax

lconst_1	mov eax,1
	push eax
	mov eax,0
	push eax

aconst_null	mov eax,0 push eax
nop	nop
pop	pop eax
pop2	pop eax pop ebx
dup	pop eax push eax push eax
dup2	pop ebx pop eax push eax push ebx push eax push ebx

dup_x1	pop eax pop ebx push eax push ebx push eax
dup2_x1	pop eax pop ebx pop ecx push ebx push eax push ecx push ebx push eax
dup_x2	pop eax pop ebx pop ecx push eax push ecx push ebx push eax
dup2_x2	pop eax pop ebx pop ecx pop edx push ebx push eax push edx push ecx push ebx;push eax
swap	pop eax pop ebx push eax push ebx
iadd	pop eax pop ebx add eax,ebx push eax
isub	pop ebx

	pop eax sub eax,ebx push eax
imul	pop eax pop edx imul edx push eax
idiv	pop ecx mov edx,0 pop eax idiv ecx push eax
irem	pop ecx mov edx,0 pop eax idiv ecx push edx
ineg	pop eax neg eax push eax
ishl	pop ecx pop eax shl eax,CL push eax
ishr	pop ecx pop eax sar eax,CL push eax
iushr	pop ecx pop eax shr eax,CL push eax
iand	pop eax pop ebx and eax,ebx push eax
ior	pop eax pop ebx

	or eax,ebx push eax
ixor	pop eax pop ebx xor eax,ebx push eax
fadd	pop real4buf fld real4buf pop real4buf fld real4buf fadd fstp real4buf push real4buf finit
fsub	pop real4buf fld real4buf pop real4buf fld real4buf;fsubr fstp real4buf push real4buf finit
fmul	pop real4buf fld real4buf pop real4buf fld real4buf fmul fstp real4buf push real4buf finit
fdiv	pop real4buf fld real4buf pop real4buf fld real4buf fdivr fstp real4buf push real4buf finit
fneg	pop real4buf

	fld real4buf fchs fstp real4buf push real4buf finit
dadd	pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fadd;fstp real8buf push dword ptr real8buf push dword ptr real8buf+4 finit
dsub	pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fsubr fstp real8buf push dword ptr real8buf push dword ptr real8buf+4 finit
dmul	pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fmul fstp real8buf push dword ptr real8buf push dword ptr real8buf+4 finit
ddiv	pop dword ptr real8buf+4

	pop dword ptr real8buf fld real8buf pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fdivr fstp real8buf push dword ptr real8buf push dword ptr real8buf+4 finit
dneg	pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fchs fstp real8buf push dword ptr real8buf push dword ptr real8buf+4 finit
i2b	pop eax and eax,0000000ffh cbw cwd shl edx,16 or eax,edx push eax
i2s	pop eax and eax,00000ffffh cwd shl edx,16 or eax,edx push eax
i2c	pop eax and eax,00000ffffh push eax
i2l	pop eax;cdq push eax push edx
i2f	pop intbuf

	fild intbuf fstp real4buf push real4buf
i2d	pop intbuf fild intbuf fstp real8buf push dword ptr real8buf push dword ptr real8buf+4
l2i	pop dword ptr longbuf+4 pop dword ptr longbuf push dword ptr longbuf
l2f	pop dword ptr longbuf+4 pop dword ptr longbuf fild longbuf fstp real4buf push real4buf
l2d	pop dword ptr longbuf+4 pop dword ptr longbuf fild longbuf fstp real8buf push dword ptr real8buf push dword ptr real8buf+4
f2d	pop real4buf fld real4buf fstp real8buf push dword ptr real8buf push dword ptr real8buf+4
d2f	pop dword ptr real8buf+4 pop dword ptr real8buf fld real8buf fstp real4buf push real4buf

APPENDIX C

jclass DefineClass(const char *name, jobject loader, const jbyte *buf,jsize len)	5
jclass FindClass(const char *name)	6
jmethodID FromReflectedMethod(jobject method)	7
jfieldID FromReflectedField(jobject field)	8
jobject ToReflectedMethod(jclass cls, jmethodID methodID, jboolean isStatic)	9
jclass GetSuperclass(jclass sub)	10
jboolean IsAssignableFrom(jclass sub, jclass sup)	11
jobject ToReflectedField(jclass cls, jfieldID fieldID, jboolean isStatic)	12
jint Throw(jthrowable obj)	13
jint ThrowNew(jclass clazz, const char *msg)	14
jthrowable ExceptionOccurred()	15
void ExceptionDescribe()	16
void ExceptionClear()	17
void FatalError(const char *msg)	18
jint PushLocalFrame(jint capacity)	19
jobject PopLocalFrame(jobject result)	20
jobject NewGlobalRef(jobject lobj)	21
void DeleteGlobalRef(jobject gref)	22
void DeleteLocalRef(jobject obj)	23
jboolean IsSameObject(jobject obj1, jobject obj2)	24
jobject NewLocalRef(jobject ref)	25
jint EnsureLocalCapacity(jint capacity)	26
jobject AllocObject(jclass clazz)	27
jobject NewObject(jclass clazz, jmethodID methodID, ...)	28
jobject NewObjectV(jclass clazz, jmethodID methodID,va_list args)	29
jobject NewObjectA(jclass clazz, jmethodID methodID,jvalue *args)	30
jclass GetObjectClass(jobject obj)	31
jboolean IsInstanceOf(jobject obj, jclass clazz)	32
jmethodID GetMethodID(jclass clazz, const char *name,const char *sig)	33
jobject CallObjectMethod(jobject obj, jmethodID methodID, ...)	34
jobject CallObjectMethodV(jobject obj, jmethodID methodID,va_list args)	35
jobject CallObjectMethodA(jobject obj, jmethodID methodID,jvalue * args)	36
jboolean CallBooleanMethod(jobject obj,jmethodID methodID, ...)	37
jboolean CallBooleanMethodV(jobject obj, jmethodID methodID,va_list args)	38
jboolean CallBooleanMethodA(jobject obj, jmethodID methodID,jvalue * args)	39

jbyte CallByteMethod(jobject obj, jmethodID methodID, ...)	40
jbyte CallByteMethodV(jobject obj, jmethodID methodID,va_list args)	41
jbyte CallByteMethodA(jobject obj, jmethodID methodID,jvalue * args)	42
jchar CallCharMethod(jobject obj, jmethodID methodID, ...)	43
jchar CallCharMethodV(jobject obj, jmethodID methodID,va_list args)	44
jchar CallCharMethodA(jobject obj, jmethodID methodID,jvalue * args)	45
jshort CallShortMethod(jobject obj, jmethodID methodID, ...)	46
jshort CallShortMethodV(jobject obj, jmethodID methodID,va_list args)	47
jshort CallShortMethodA(jobject obj, jmethodID methodID,jvalue * args)	48
jint CallIntMethod(jobject obj, jmethodID methodID, ...)	49
jint CallIntMethodV(jobject obj, jmethodID methodID,va_list args)	50
jint CallIntMethodA(jobject obj, jmethodID methodID,jvalue * args)	51
jlong CallLongMethod(jobject obj, jmethodID methodID, ...)	52
jlong CallLongMethodV(jobject obj, jmethodID methodID,va_list args)	53
jfloat CallFloatMethod(jobject obj, jmethodID methodID, ...)	54
jfloat CallFloatMethodV(jobject obj, jmethodID methodID,va_list args)	55
jfloat CallFloatMethodA(jobject obj, jmethodID methodID,jvalue * args)	56
jdouble CallDoubleMethod(jobject obj, jmethodID methodID, ...)	57
jdouble CallDoubleMethodV(jobject obj, jmethodID methodID,va_list args)	58
jdouble CallDoubleMethodA(jobject obj, jmethodID methodID,jvalue * args)	59
void CallVoidMethod(jobject obj, jmethodID methodID, ...)	60
void CallVoidMethodV(jobject obj, jmethodID methodID,va_list args)	61
void CallVoidMethodA(jobject obj, jmethodID methodID,jvalue * args)	62
jobject CallNonvirtualObjectMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	63
jobject CallNonvirtualObjectMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	64
jobject CallNonvirtualObjectMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	65
jboolean CallNonvirtualBooleanMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	66
jboolean CallNonvirtualBooleanMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	67
jboolean CallNonvirtualBooleanMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	68
jbyte CallNonvirtualByteMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	69
jbyte CallNonvirtualByteMethodV(jobject obj, jclass clazz,jmethodID methodID,	70

va_list args)	
jbyte CallNonvirtualByteMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	71
jchar CallNonvirtualCharMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	72
jchar CallNonvirtualCharMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	73
jchar CallNonvirtualCharMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	74
jshort CallNonvirtualShortMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	75
jshort CallNonvirtualShortMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	76
jshort CallNonvirtualShortMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	77
jint CallNonvirtualIntMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	78
jint CallNonvirtualIntMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	79
jint CallNonvirtualIntMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	80
jlong CallNonvirtualLongMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	81
jlong CallNonvirtualLongMethodV(jobject obj, jclass clazz,jmethodID methodID, va_list args)	82
jlong CallNonvirtualLongMethodA(jobject obj, jclass clazz,jmethodID methodID, jvalue * args)	83
jfloat CallNonvirtualFloatMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	84
jfloat CallNonvirtualFloatMethodV(jobject obj, jclass clazz,jmethodID methodID,va_list args)	85
jfloat CallNonvirtualFloatMethodA(jobject obj, jclass clazz,jmethodID methodID,jvalue * args)	86
jdouble CallNonvirtualDoubleMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	87
jdouble CallNonvirtualDoubleMethodV(jobject obj, jclass clazz,jmethodID methodID,va_list args)	88
jdouble CallNonvirtualDoubleMethodA(jobject obj, jclass clazz,jmethodID methodID,jvalue * args)	89
void CallNonvirtualVoidMethod(jobject obj, jclass clazz,jmethodID methodID, ...)	90

void CallNonvirtualVoidMethodV(jobject obj, jclass clazz,jmethodID methodID,va_list args)	91
void CallNonvirtualVoidMethodA(jobject obj, jclass clazz,jmethodID methodID,jvalue * args)	92
jfieldID GetFieldID(jclass clazz, const char *name,const char *sig)	93
jobject GetObjectField(jobject obj, jfieldID fieldID)	94
jboolean GetBooleanField(jobject obj, jfieldID fieldID)	95
jbyte GetByteField(jobject obj, jfieldID fieldID)	96
jchar GetCharField(jobject obj, jfieldID fieldID)	97
jshort GetShortField(jobject obj, jfieldID fieldID)	98
jint GetIntField(jobject obj, jfieldID fieldID)	99
jlong GetLongField(jobject obj, jfieldID fieldID)	100
jfloat GetFloatField(jobject obj, jfieldID fieldID)	101
jdouble GetDoubleField(jobject obj, jfieldID fieldID)	102
void SetObjectField(jobject obj, jfieldID fieldID, jobject val)	103
void SetBooleanField(jobject obj, jfieldID fieldID,jboolean val)	104
void SetByteField(jobject obj, jfieldID fieldID,jbyte val)	105
void SetCharField(jobject obj, jfieldID fieldID,jchar val)	106
void SetShortField(jobject obj, jfieldID fieldID,jshort val)	107
void SetIntField(jobject obj, jfieldID fieldID,jint val)	108
void SetLongField(jobject obj, jfieldID fieldID,jlong val)	109
void SetFloatField(jobject obj, jfieldID fieldID,jfloat val)	110
void SetDoubleField(jobject obj, jfieldID fieldID,jdouble val)	111
jmethodID GetStaticMethodID(jclass clazz, const char *name,const char *sig)	112
jobject CallStaticObjectMethod(jclass clazz, jmethodID methodID,...)	113
jobject CallStaticObjectMethodV(jclass clazz, jmethodID methodID,va_list args)	114
jobject CallStaticObjectMethodA(jclass clazz, jmethodID methodID,jvalue *args)	115
jboolean CallStaticBooleanMethod(jclass clazz,jmethodID methodID, ...)	116
jboolean CallStaticBooleanMethodV(jclass clazz,jmethodID methodID, va_list args)	117
jboolean CallStaticBooleanMethodA(jclass clazz,jmethodID methodID, jvalue *args)	118
jbyte CallStaticByteMethod(jclass clazz,jmethodID methodID, ...)	119
jbyte CallStaticByteMethodV(jclass clazz,jmethodID methodID, va_list args)	120
jbyte CallStaticByteMethodA(jclass clazz,jmethodID methodID, jvalue *args)	121
jchar CallStaticCharMethod(jclass clazz,jmethodID methodID, ...)	122
jchar CallStaticCharMethodV(jclass clazz,jmethodID methodID, va_list args)	123

jchar CallStaticCharMethodA(jclass clazz,jmethodID methodID, jvalue *args)	124
jshort CallStaticShortMethod(jclass clazz,jmethodID methodID, ...)	125
jshort CallStaticShortMethodV(jclass clazz,jmethodID methodID, va_list args)	126
jshort CallStaticShortMethodA(jclass clazz,jmethodID methodID, jvalue *args)	127
jint CallStaticIntMethod(jclass clazz,jmethodID methodID, ...)	128
jint CallStaticIntMethodV(jclass clazz,jmethodID methodID, va_list args)	129
jint CallStaticIntMethodA(jclass clazz,jmethodID methodID, jvalue *args)	130
jlong CallStaticLongMethod(jclass clazz,jmethodID methodID, ...)	131
jlong CallStaticLongMethodV(jclass clazz,jmethodID methodID, va_list args)	132
jlong CallStaticLongMethodA(jclass clazz,jmethodID methodID, jvalue *args)	133
jfloat CallStaticFloatMethod(jclass clazz,jmethodID methodID, ...)	134
jfloat CallStaticFloatMethodV(jclass clazz,jmethodID methodID, va_list args)	135
jfloat CallStaticFloatMethodA(jclass clazz,jmethodID methodID, jvalue *args)	136
jdouble CallStaticDoubleMethod(jclass clazz,jmethodID methodID, ...)	137
jdouble CallStaticDoubleMethodV(jclass clazz,jmethodID methodID, va_list args)	138
jdouble CallStaticDoubleMethodA(jclass clazz,jmethodID methodID, jvalue *args)	139
void CallStaticVoidMethod(jclass cls, jmethodID methodID, ...)	140
void CallStaticVoidMethodV(jclass cls, jmethodID methodID,va_list args)	141
void CallStaticVoidMethodA(jclass cls, jmethodID methodID,jvalue * args)	142
jfieldID GetStaticFieldID(jclass clazz, const char *name,const char *sig)	143
jobject GetStaticObjectField(jclass clazz, jfieldID fieldID)	144
jboolean GetStaticBooleanField(jclass clazz, jfieldID fieldID)	145
jbyte GetStaticByteField(jclass clazz, jfieldID fieldID)	146
jchar GetStaticCharField(jclass clazz, jfieldID fieldID)	147
jshort GetStaticShortField(jclass clazz, jfieldID fieldID)	148
jint GetStaticIntField(jclass clazz, jfieldID fieldID)	149
jlong GetStaticLongField(jclass clazz, jfieldID fieldID)	150
jfloat GetStaticFloatField(jclass clazz, jfieldID fieldID)	151
jdouble GetStaticDoubleField(jclass clazz, jfieldID fieldID)	152
void SetStaticObjectField(jclass clazz, jfieldID fieldID,jobject value)	153
void SetStaticBooleanField(jclass clazz, jfieldID fieldID,jboolean value)	154
void SetStaticByteField(jclass clazz, jfieldID fieldID,jbyte value)	155
void SetStaticCharField(jclass clazz, jfieldID fieldID,jchar value)	156
void SetStaticShortField(jclass clazz, jfieldID fieldID,jshort value)	157
void SetStaticIntField(jclass clazz, jfieldID fieldID,jint value)	158
void SetStaticLongField(jclass clazz, jfieldID fieldID,jlong value)	159
void SetStaticFloatField(jclass clazz, jfieldID fieldID,jfloat value)	160

void SetStaticDoubleField(jclass clazz, jfieldID fieldID,jdouble value)	161
jstring NewString(const jchar *unicode, jsize len)	162
jsize GetStringLength(jstring str)	163
const jchar *GetStringChars(jstring str, jboolean *isCopy)	164
void ReleaseStringChars(jstring str, const jchar *chars)	165
jstring NewStringUTF(const char *utf)	166
jsize GetStringUTFLength(jstring str)	167
const char* GetStringUTFChars(jstring str, jboolean *isCopy)	168
void ReleaseStringUTFChars(jstring str, const char* chars)	169
jsize GetArrayLength(jarray array)	170
jobjectArray NewObjectArray(jsize len, jclass clazz,jobject init)	171
jobject GetObjectArrayElement(jobjectArray array, jsize index)	172
void SetObjectArrayElement(jobjectArray array, jsize index,jobject val)	173
jbooleanArray NewBooleanArray(jsize len)	174
jbyteArray NewByteArray(jsize len)	175
jcharArray NewCharArray(jsize len)	176
jshortArray NewShortArray(jsize len)	177
jintArray NewIntArray(jsize len)	178
jlongArray NewLongArray(jsize len)	179
jfloatArray NewFloatArray(jsize len)	180
jdoubleArray NewDoubleArray(jsize len)	181
jboolean * GetBooleanArrayElements(jbooleanArray array, jboolean *isCopy)	182
jbyte * GetByteArrayElements(jbyteArray array, jboolean *isCopy)	183
jchar * GetCharArrayElements(jcharArray array, jboolean *isCopy)	184
jshort * GetShortArrayElements(jshortArray array, jboolean *isCopy)	185
jint * GetIntArrayElements(jintArray array, jboolean *isCopy)	186
jlong * GetLongArrayElements(jlongArray array, jboolean *isCopy)	187
jfloat * GetFloatArrayElements(jfloatArray array, jboolean *isCopy)	189
jdouble * GetDoubleArrayElements(jdoubleArray array, jboolean *isCopy)	190
void ReleaseBooleanArrayElements(jbooleanArray array,jboolean *elems,jint mode)	191
void ReleaseByteArrayElements(jbyteArray array,jbyte *elems,jint mode)	192
void ReleaseCharArrayElements(jcharArray array,jchar *elems,jint mode)	193
void ReleaseShortArrayElements(jshortArray array,jshort *elems,jint mode)	194
void ReleaseIntArrayElements(jintArray array,jint *elems,jint mode)	195
void ReleaseLongArrayElements(jlongArray array,jlong *elems,jint mode)	196
void ReleaseFloatArrayElements(jfloatArray array,jfloat *elems,jint mode)	197

void ReleaseDoubleArrayElements(jdoubleArray array,jdouble *elems,jint mode)	198
void GetBooleanArrayRegion(jbooleanArray array,jsize start, jsize len, jboolean *buf)	199
void GetByteArrayRegion(jbyteArray array,jsize start, jsize len, jbyte *buf)	200
void GetCharArrayRegion(jcharArray array,jsize start, jsize len, jchar *buf)	201
void GetShortArrayRegion(jshortArray array,jsize start, jsize len, jshort *buf)	202
void GetIntArrayRegion(jintArray array,jsize start, jsize len, jint *buf)	203
void GetLongArrayRegion(jlongArray array,jsize start, jsize len, jlong *buf)	204
void GetFloatArrayRegion(jfloatArray array,jsize start, jsize len, jfloat *buf)	205
void GetDoubleArrayRegion(jdoubleArray array,jsize start, jsize len, jdouble *buf)	206
void SetBooleanArrayRegion(jbooleanArray array, jsize start, jsize len,jboolean *buf)	207
void SetByteArrayRegion(jbyteArray array, jsize start, jsize len,jbyte *buf)	208
void SetCharArrayRegion(jcharArray array, jsize start, jsize len,jchar *buf)	209
void SetShortArrayRegion(jshortArray array, jsize start, jsize len,jshort *buf)	210
void SetIntArrayRegion(jintArray array, jsize start, jsize len,jint *buf)	211
void SetLongArrayRegion(jlongArray array, jsize start, jsize len,jlong *buf)	212
void SetFloatArrayRegion(jfloatArray array, jsize start, jsize len,jfloat *buf)	213
void SetDoubleArrayRegion(jdoubleArray array, jsize start, jsize len,jdouble *buf)	214
jint RegisterNatives(jclass clazz, const JNINativeMethod *methods,jint nMethods)	215
jint UnregisterNatives(jclass clazz)	216
jint MonitorEnter(jobject obj)	217
jint MonitorExit(jobject obj)	218
jint GetJavaVM(JavaVM **vm)	219
void GetStringRegion(jstring str, jsize start, jsize len, jchar *buf)	220
void GetStringUTFRegion(jstring str, jsize start, jsize len, char *buf)	221
void * GetPrimitiveArrayCritical(jarray array, jboolean *isCopy)	222
void ReleasePrimitiveArrayCritical(jarray array, void *carray, jint mode)	223
const jchar * GetStringCritical(jstring string, jboolean *isCopy)	224
void ReleaseStringCritical(jstring string, const jchar *cstring)	225
jweak NewWeakGlobalRef(jobject obj)	226
void DeleteWeakGlobalRef(jweak ref)	227
jboolean ExceptionCheck()	228
jobject NewDirectByteBuffer(void* address, jlong capacity)	229
void* GetDirectBufferAddress(jobject buf)	230
jlong GetDirectBufferCapacity(jobject buf)	231