Chapter 1 Introduction

1.1 The Java Programming 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. As the massive growth of the Internet and the WWW, Java has become more and more popular since it was first introduced by Sun Microsystems in 1995.

Figure 1.1 shows the simplified flow of compiling and running Java programs. Each program (.java) is translated by a Java compiler into a platform-independent intermediate form — the Java bytecode (.class). The Java bytecode can then be loaded by a classloader from local disks or remote hosts via network connection, and run in the Java Virtual Machine[5,21,22,23,33] (interpreting or just-in-time compiling). The Java Virtual Machine may be implemented by software or hardware.

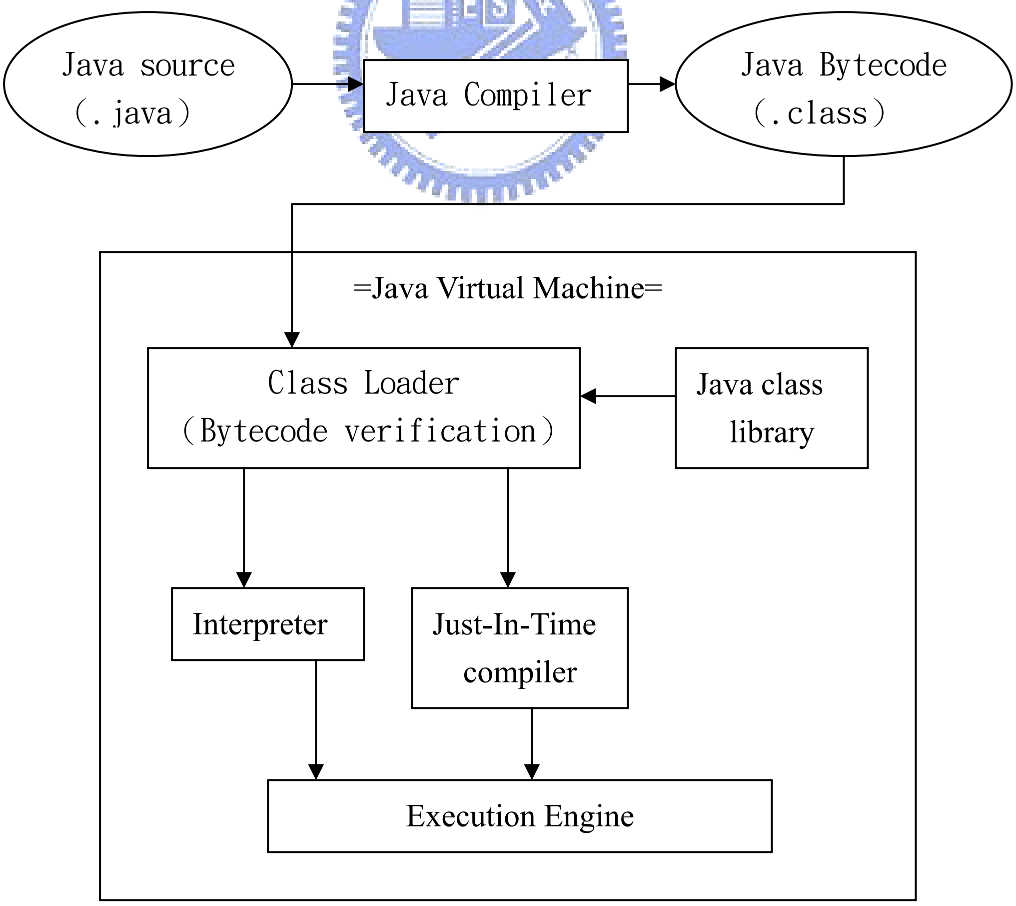


Figure 1.1 Compiling and Running Java programs

Here we summarize some features of the Java Programming Language[3].

- 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.

1.2 Research Motivation

Successful as Java is, the performance of Java is still an issue. It is implemented by compiling to portable bytecodes[5]. However, interpreting bytecodes makes Java program many times slower than comparable C or C++ programs. One approach to improving this situation is "Just-In-Time" (JIT) compilers. Dynamically translating bytecodes to machine codes just before methods are first executed. This can provide substantial speed up, but it is still slower than C or C++. There are two main drawbacks with the JIT approach compared to conventional compilers：

- The compilation is done every time the application is executed, which means start-up times are much worse than pre-compiled code.

- Since the JIT compiler has to run fast , it cannot do any nontrivial optimization. Only simple register allocation and peepoptimizations are practical.

While JIT compilers have an important position 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.

1.3 Goals

There are several benefits in translating Java bytecode to X86 assembly code.

一、 improving the execution time than interpreter.
二、 Providing a chance for combining existent assembly code with the translated assembly code.
三、 Providing a chance for interaction of assembly code and Java Virtual Machine.

The goals of our work are to develop a tool that models the flow of translation without generating intermediate representation[6,8], and simplify the task of translating Java bytecode, and demonstrate significant optimizations which touch bytecodes directly and can improve the performance. An optimizing translator which is from Java bytecodes to native machine codes and generates intermediate representations is too complex and is understood 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[1,16] or other native compilers[15,17,19]. So, our translator simplifies the translation flow, and still has good performance.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 briefly overviews the system architecture of Java Virtual Machine, and introduces every kind of software approach to bytecode execution enhancement. Chapter 3 describes the system architecture of our approach and how the system translates every kind of Java bytecode to X86 assembly code and introduces some associated translating examples. The experimental evaluation and analysis are discussed in Chapter 4. Chapter 5 concludes this thesis and discusses possible future works.

Chapter 2 Variations of Java Bytecode Execution

Interpretation is slow. Since the design of Java never restricts itself to be interpreted, we may try to enhance the performance of Java using different approaches. Several possible ways of implementing a faster performance are enumerated below.

2.1 Just-In-Time Code Generators

A "Just-In-Time" (JIT) Java compiler produces native code from Java bytecode instructions during program execution. The results of the compilation are not kept between runs. A JIT code generator can make use of specific hardware coprocessors running in the current environment. Figure 2.1 is the data structures of Intel VM.

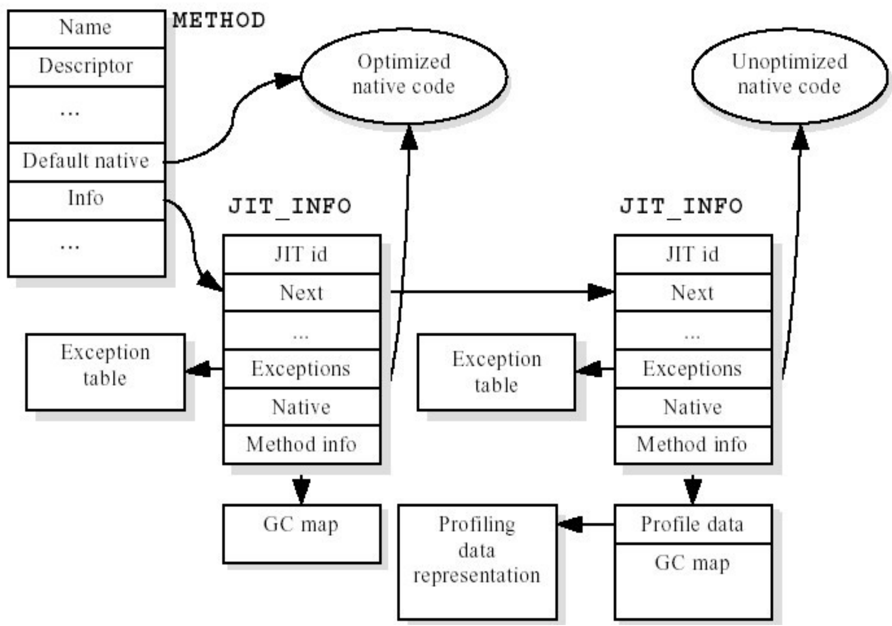


Figure 2.1 Data structures of Intel VM.

Overall program execution time now includes JIT compilation time, in contrast to the traditional methodology of performance measurement in which compilation time is ignored. Compilation speed is more important in a Java JIT compiler than in a traditional compiler, requiring optimization algorithms to be lightweight and effective. It is also important for the Java JIT compiler to interact with other parts of the system.

So far, there are two types of JIT implementation. One is fast, effective code generation[1]. It has no explicit intermediate representation, and it generates native code directly from bytecodes in a pass. The first pass is gathering important information, the second pass is lazy code selection. Figure 2.2 shows it´s five major phases.

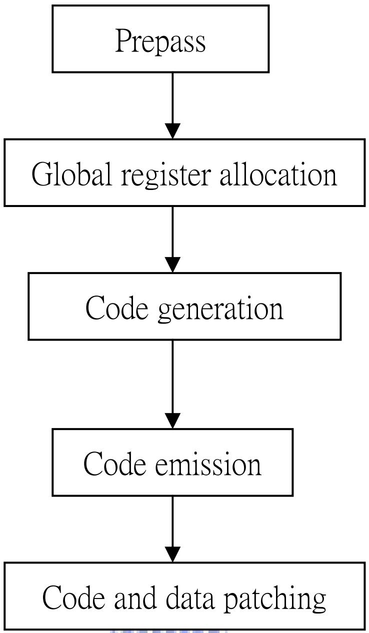


Figure 2.2 Fast, effective code generation in JIT compiler

Another one is optimizing compiler[18]. The optimizing compiler takes a conventional compilation approach that builds an intermediate representation (IR) and performs global optimizations based on the IR. Global optimizations are highly effective in improving the code quality. However, they are expensive in terms of compilation time. Therefore, we apply global optimization to a method only if the method is identified as "hot". Figure 2.3 is the structure of the optimizing compiler.

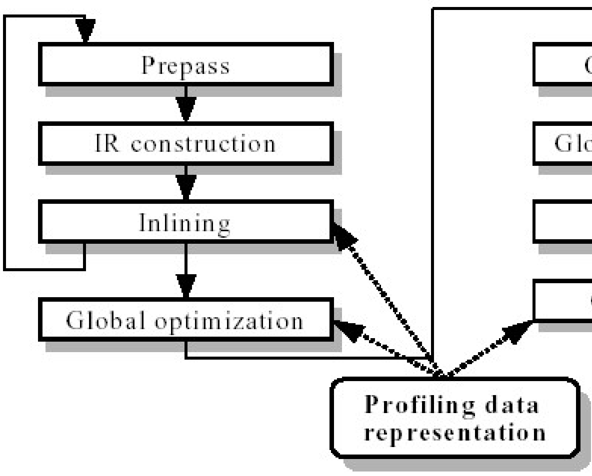


Figure 2.3 Structure of the optimizing compiler

2.2 Native Compilers

Native compilers translate Java source code/bytecode to native code which is platform dependent but is not portable. Since it is assumed that the compilation is performed once and maintained on a disk, additional time may be devoted to optimizations. Most of these native compilers[6,17,15,19] are forced to build 3-address code intermediate representations, and perform significant optimizations. Among them, translated code by some native compilers, which is runned at nearly the full performance of native code directly generated from a source representation such as the C programming language. The following shows the translation path of native compilers which belong to this category[8].

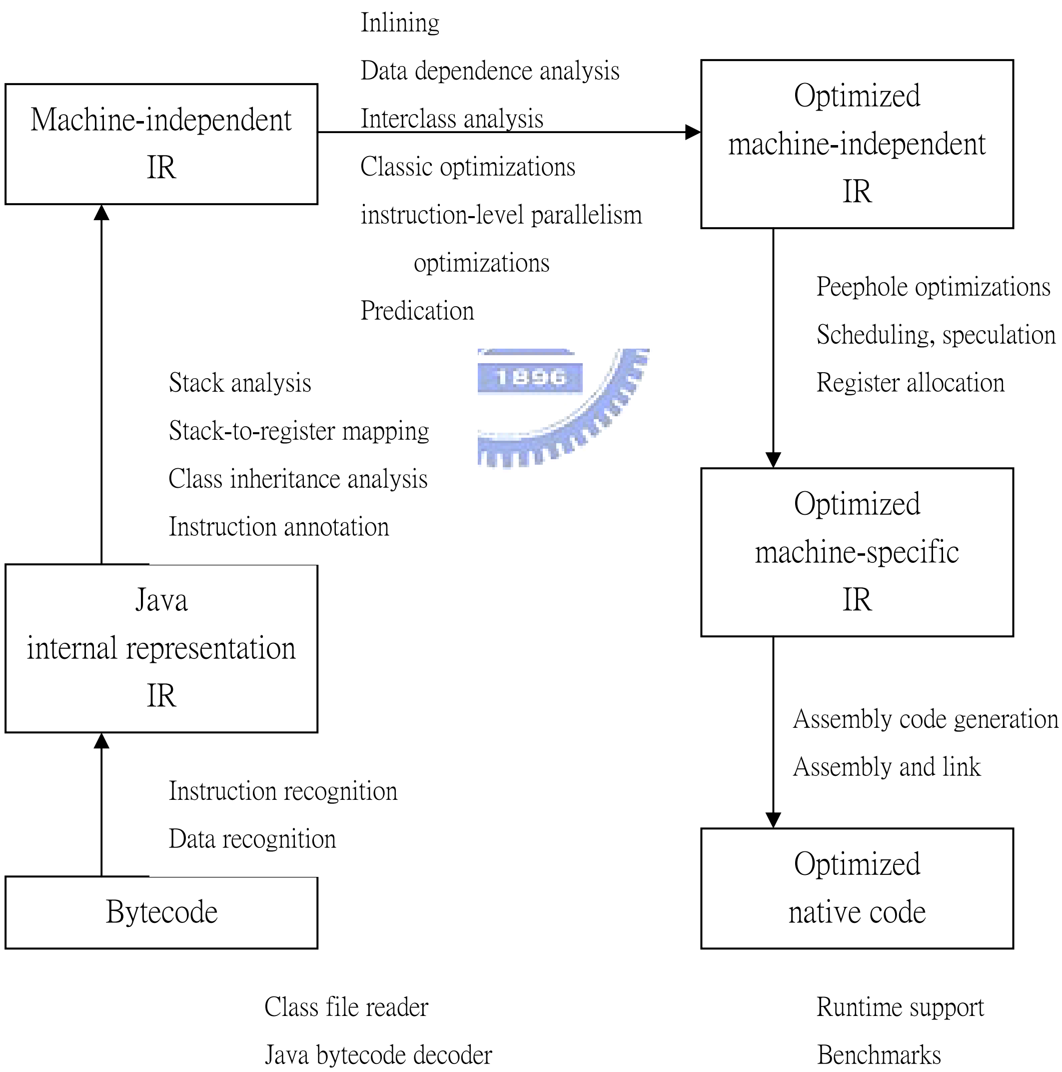


Figure 2.4 Impact NET compiler translation path

Few of native compilers do not have intermediate representations. They use mapping styles to generate native code, such as JBCC[7]. Our paper focus on those methods which they propose, and enhance them.

2.3 Bytecode Annotator

Tools of this category analyze bytecode and produce new class files with annotations which convey information to the virtual machine on how to execute the bytecode faster. We are aware of one such system[14] which passes register allocation information to a JIT compiler in this manner. They obtain speed-ups between 17% to 41% on a set of four scientific benchmarks.

2.4 Bytecode Optimization

There are two types of Java bytecode optimization. One is optimizing the bytecode directly[11]. Apply well known optimizations[2] to Java bytecodes. It has little effect that optimizing nonexpensive bytecode. To perform effective optimizations at this level, one must consider more advanced optimizations which directly reduce the use of these expensive bytecodes.

Another one is translating Java bytecode to intermediate representation, and then optimizing IR, and then generating new class file, such as Soot[12].

Chapter 3 System Design and Implementation

In this chapter, we describe the design and the implementation of our Translator. First, the architecture, which includes execution environment and internal data structure and pass1 and pass2 of our translator, is introduced. Then we will detail that how to translate every kind of bytecodes to assembly codes, and template matching. Especially, dynamic call is the important item. We will use auxiliary examples for explaining our techniques on every items.

3.1 System Architecture

In this study, we develop a translator which is effective and performance is close to JIT compiler. The first issue when we design the system is that assembly code how to simulate the operation model under JVM. In X86 assembly language, instructions can use registers, memory and stack, which are all system resource, under the X86 architecture. Because Java Virtual Machine is stack-based machine, we have to arrange available resources which is under X86 architecture to conform with Java bytecodes operation flow. There is a existing solution for this problem — JBCC[7]. JBCC provides a basic concept about mapping between Java bytecode and X86 assembly code, but it only explains roughly and are not detailed. Therefore, we decide to detail their arguments and to add some functions which can interact between JVM and assembly code, and use Java bytecodes optimization concepts[11] to convert several Java bytecode instructions to less assembly codes. Figure 3.1 shows the three compiling techniques of JBCC.

According to Figure 3.1, our system consists of three parts. The first part, the Prepass phase, collects some informations which are needed for code generation. The second part, Mapping Code Generation phase, truly generates x86 assembly code. The third part, Template Matching phase, uses Java bytecodes optimization concepts[11] to do some simple optimization actions. We also introduce the first part in the section particularly, and introduce other parts basically.

In the section, we hope to explain the whole system motion, and the translation of every type of bytecodes is discussed in section 3.2. So, we can understand our translator completely from section 3.1 and 3.2.

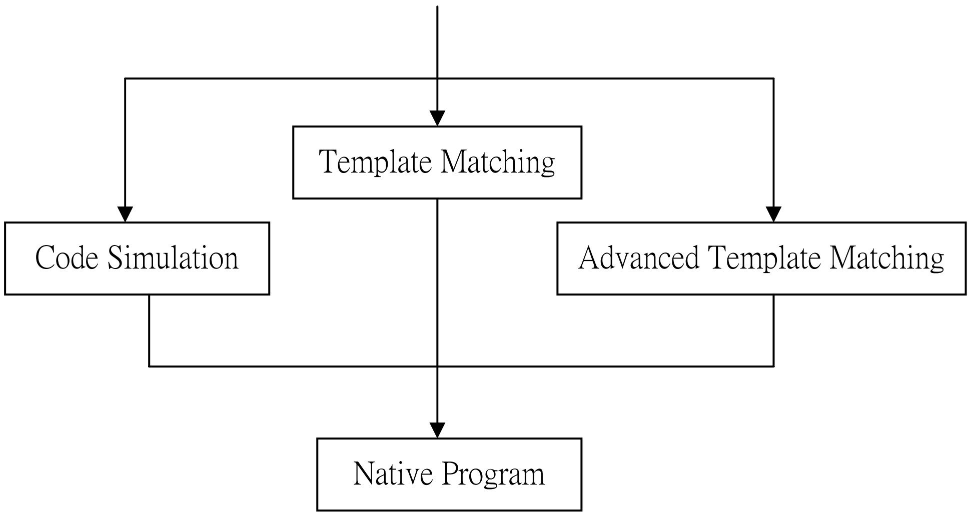


Figure 3.1 The three compiling techniques of JBCC

3.1.1 Execution Environment

Figure 3.2 explains the environment from Java applications (.class) to X86 assembly codes (.asm), and 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 with those translated assembly files by using bcc32 compiler to assembler and link those translated assembly files. Because our translater has the interaction capacity 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 to have the capacity which communicates assembly codes with Java Virtual Machine. In the front of section 3.2 will explain those contents completely.

Figure 3.3 explains how to execute those translated assembly files. We use the bcc32 compiler to assemble and link those assembly files with Java Virtual Machine. The c:\j2sdk1.4.1_01 is the directory where Java 2 sdk is installed. You need to supply correct including 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. 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 a changed OMF format.

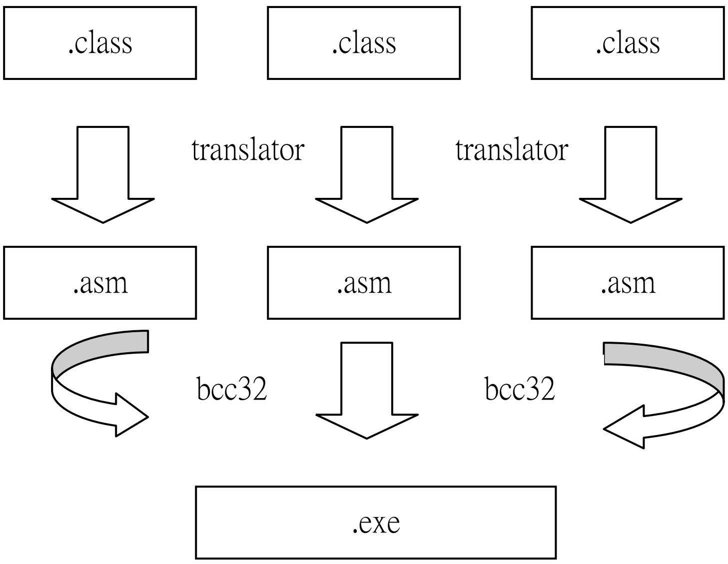


Figure 3.2 Execution environment

```
bcc32 －Ic:\j2sdk1.4.1_01\include －Ic:\j2sdk1.4.1_01\include\win32
   a.asm  b.asm  c.asm  startvm.c  c:\j2sdk1.4.1_01\lib\jvm1.lib
```

Figure 3.3 Using bcc32 to assemble and link assembly codes with JVM

According to Figure 3.3 example, there is a Java application which consists of a.class, b.class, and c.class. These three class files may come from different Java programs, but are cooperated. These three class files are translated separately by our translator, to generate a.asm, b.asm, and c.asm respectively. The a.class contains the main procedure which is starting point, so we put the a.asm on the first position. 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 a truly executable file. Assembly codes can bind the Java Virtual Machine through this compilation method, and then calls Java API from assembly codes. The tool which assembles and links X86 assembly files in bcc32 compiler, is through TASM.

3.1.2 Translation Environment and Flow

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

Figure 3.4 shows the simple translation environment. The input class file is translated to X86 assembly code, and the hidden *javap* action is performed by translator.

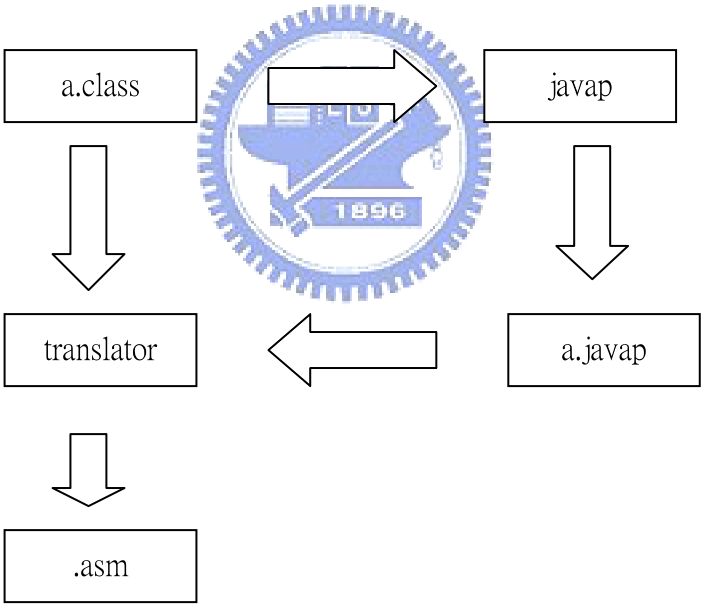


Figure 3.4 The simple translation environment

The total translation flow is shown in figure 3.5. To this end, 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 name and signature, (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.

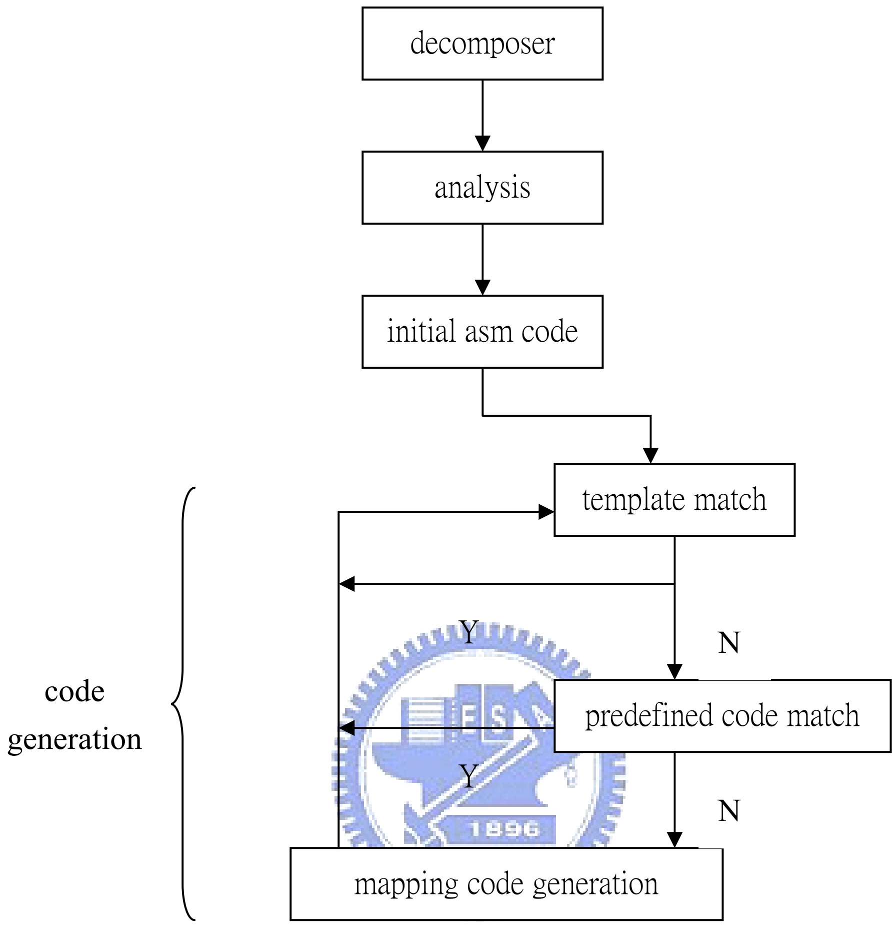


Figure 3.5 Translation flow

In stage 2, the information obtained in stage 1 is analyzed and translated into the data structures shown in Figure 3.6. Bytecodes are grouped by the methods in which they reside. All information pertaining to each instruction, including the parameters, the offset from the beginning of the method, become attributes in a node. The node is placed into a linked list (the methodcontent) which is later manipulated to insert and remove instructions.

Stage 3 initials the assembly code space and writes some known codes to the space before truly code generation. Because this stage is about mapping between Java Virtual Machine and X86 register machine, we will discuss the stage in section 3.1.3. Stage 4 converts several Java bytecode instructions to less native codes. If can not find template, the flow will enter into the stage 5. We will discuss the stage 4 in section 3.2.5. Because some bytecode instructions are simple, we can predefine assembly codes about those instructions in a text file, as shown in Figure 3.7. When the stage 2, it parses the predefined file into internal data structures for used in stage 5. If can not find predefined assembly codes, the flow will enter into the stage 6. The stage 6 map Java bytecode instruction to assembly language instructions. We will discuss the stage 6 in section 3.2.

method list	a linked list of each method´s attributes
	name , signature
	total number of local variables
	maximum number of operand stack
	static flag
	branch table
	jsr ret table
	method content
	exception content
method content	a linked list of each instruction´s attributes
	offset
	instruction name
	parameters
exception content	a linked list of each exception in a exception table
	from
	to
	target
	exception type
constant pool table	constant pool array of the class file
asm information	predefined assembly codes
allclassmethodtable	informations about dynamic dispatch

Figure 3.6 Translator´s data structure

The method list, methodcontent, exceptioncontent, asm information, allclassmethodtable are linked list, and branch table, jsr ret table are binary search tree. Figure 3.8 shows the diagram of the internal data structure.

```
//堆疊及區域變數
iconst_m1^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;
…
//流程控制
//算術邏輯轉換
…
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;
…
fmul^pop real4buf;fld real4buf;pop real4buf;fld real4buf;fmul;fstp real4buf;push real4buf;
fdiv^pop real4buf;fld real4buf;pop real4buf;fld real4buf;fdivr;fstp real4buf;push real4buf;
fneg^pop real4buf;fld real4buf;fchs;fstp real4buf;push real4buf;
…
f2d^pop real4buf;fld real4buf;fstp real8buf;push dword ptr real8buf;push dword ptr real8buf+4;
…
```

Figure 3.7 Predefined assembly codes

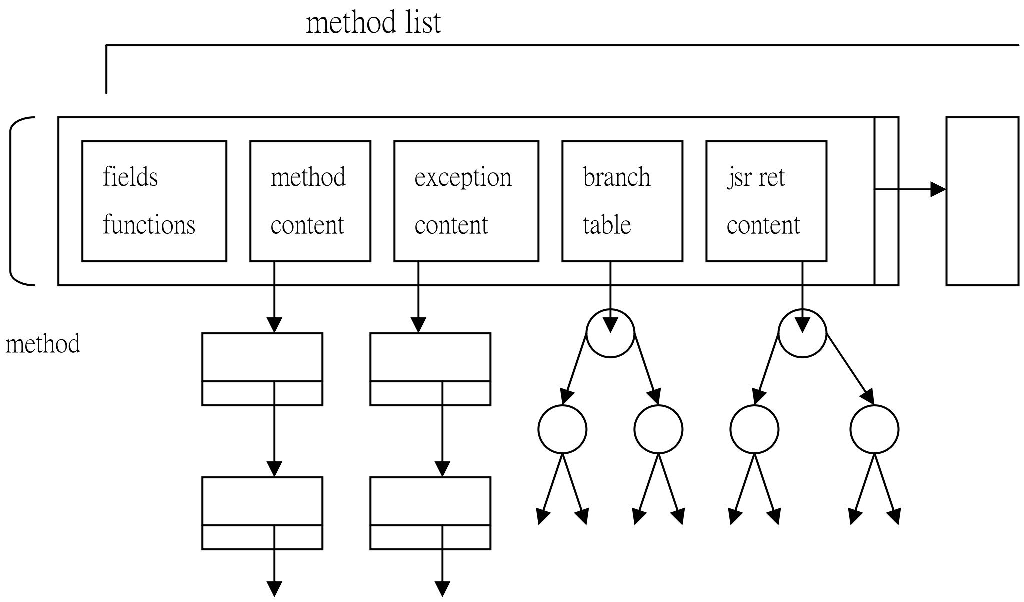


Figure 3.8 The diagram of the internal data structure

In the stage 2, we generate some internal tables which consist of *constant pool table*, *allclass methodtable*, *branch table*, *jsr ret table*, *exception table*, and *instruction map table*. The constant pool table saves the constant pool array of the class file. The type is 0 to 7 in a constant pool, and 0 is class, 1 is field, 2 is method, 3 is string, 4 is int, 5 is float, 6 is long, 7 is double. The type is convenient for programming. The allclassmethodtable has all the methods which come from every class which appears in the constant pool and their super class and super class up to java.lang.Object. A entry is a class which appears in the constant pool, and it´s methods which including inheritance up to java.lang.Object. The allclassmethodtable associates with dynamic dispatching in assembly code, and we will detail the content in section 3.2.4.

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 are all predefined assembly code. These tables are shown in Figure 3.9.

Constant Pool Table

index	type	classname	method/field name	signature	resolved

Allclassmethodtable

classname idofclass superclass methodname signature clsname callasmname … methodname signature clsname callasmname

Branch Table

index	label	mode

Jsr Ret Table

index	label

Exception Table

exception type	from	to	target

Instruction Map Table

bytecode	assembly

Figure 3.9 Internal tables of the translator

3.1.3 Map Java Stack Machine to X86 Register Machine

By studying Java bytecode instruction set, we notice that each of the Java bytecode instruction can be represented by one or more that one assembly language instructions. Operands of most bytecode instructions are associated with the operand stack of the Java Virtual Machine. Bytecode instructions take operands from the operand stack, and manipulate them, and then push results into the operand stack. Therefore, we use the native stack operation to emulate the stacks of JVM. Under the architecture of Intel X86 series CPU, we can use the native stack, registers, and memory in assembly programming level. So, map the operand stack of the frame every method to native stack, and map local variables of the frame every method to memory. In order to process dynamic dispatching, we use memory to save created objects for searching later. In the architecture of the X86 .asm assembly code, the .data section has the name which are named *.._local_ vars* for the local variables. The *.._local_ vars* is an one- dimension array, and its quantity is from the class file, and the unit of every entry in *.._local vars* is 4 bytes. The .data section has the name which are named *.._objmapclass* for save created objects. The *.._objmapclass* is the structure, and we will discuss the usage of the *.._objmapclass* in the section 3.2.4.

Those architectural registers of X86 CPU are for scratch registers. When we evaluate an expression, temporary values of the operation process and operands of the operation all employ scratch registers. Those registers are not dedicated to the particular destination, such as local variables. Those registers are purely for evaluation. The final result of the assembly codes corresponding to bytecode instruction are pushed into native stack or are saved in local variables. Referring to the following example.

```
.data
   …
   aa_local_vars          dword      2    dup(?)
   aa_objmapclass    objmapclass_table    3    dup(<-1,-1>)
   …
.code
   …
   push     [aa_local_vars+1*4]                  ; aa is the simple name of the method
   …                                             ; there are 26*26 in a class
```

Because we have to generate a .asm file, how to arrage the order of generated codes is important. A .asm file consists of three sections which are prior to .data, .data, and .code. In initial asm codes phase which are described in previous section, some known codes can be filled in these three sections, and methods are classified into three types. There are different compositions for different types in a .asm file. A type which is the main procedure, its .code section has to call the startvm.c function, and no parameters which are passed into. A type which is the first procedure in a no main procedure´s class file, it is like to the main procedure, but do not call the startvm.c function, and has parameters which are passed into. The two types are the beginning of a .asm file, and the start point is .386 assembler directive. The another type is the method which is besides the main and first procedure. It is appended to the previous two types in a .asm file, and the start point is procedure name. In order to generate a .asm file conveniently, we adopt the classification manner. Figure 3.10 shows the initial asm codes of three type methods.

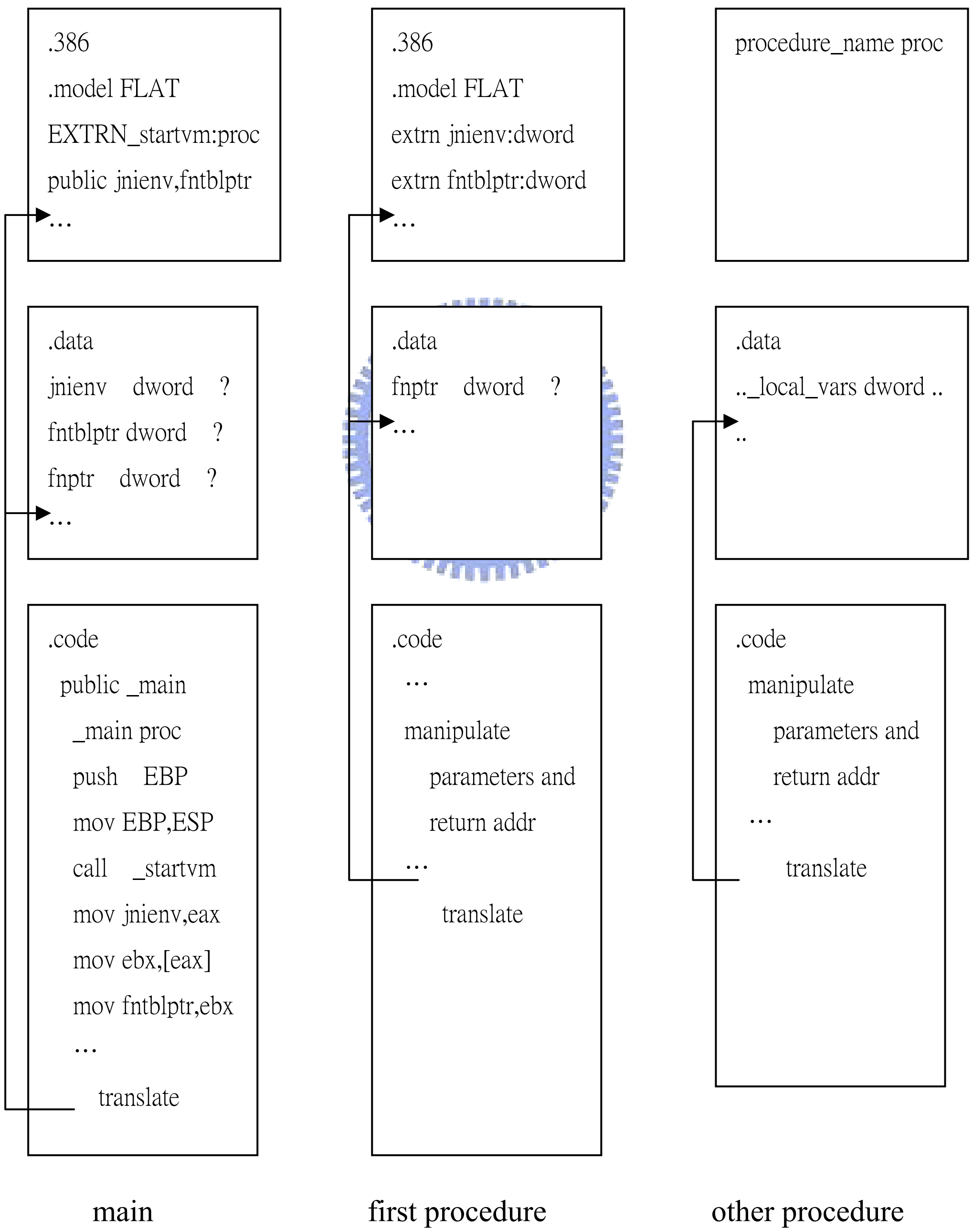


Figure 3.10 Initial asm codes of three type methods

Our translator preserves three memory blocks for three sections which are prior to .data, .data, and .code. We fill the translated assembly codes into the three memory blocks during the translating period.

3.2 System Implementation

In this section, we focus on the pass 2, and describe how the translator translates bytecode instructions. Figure 3.11 shows the flow of the translator translating.

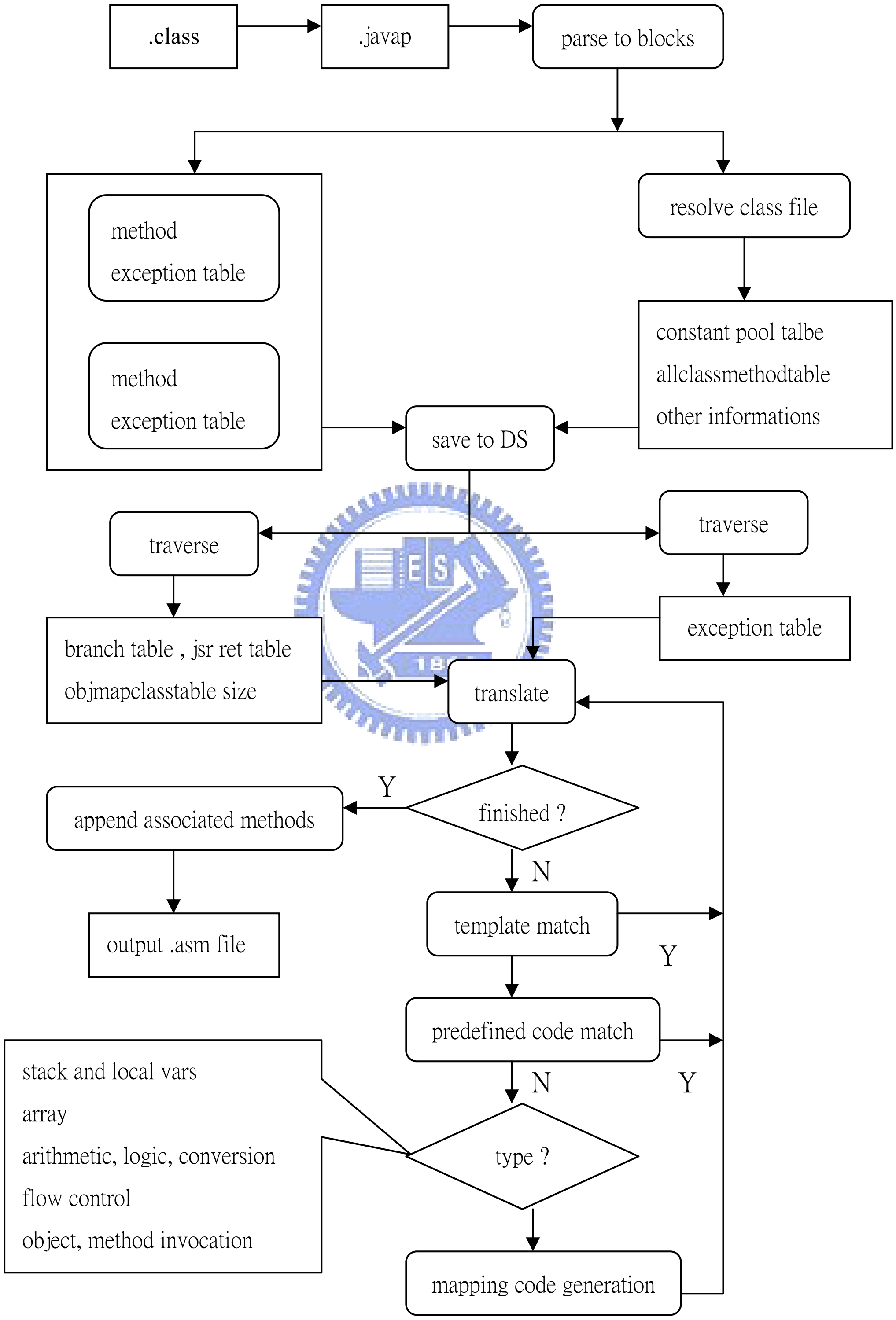


Figure 3.11 Flow of translator translating

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.

3.2.1 Start JVM From Assembly Code

A Java Virtual Machine implementation is typically shipped as a native library. Native applications can link against this library and use the invocation interface to load the Java Virtual Machine. So, we prepare for a C function which uses Win32 API to invoke Java Virtual Machine. The startvm.c is described in Appendix.

When it targets the 1.1 release, the C code begins with a call to *JNI_GetDefaultJavaVMInitArgs* to obtain the default virtual machine settings. Those settings are in the *vm_args* parameter. When it targets the 1.2 release, the C code creates a *JavaVMInitArgs* structure. The virtual machine initialization arguments are stored in a *JavaVMOption* array. The *LoadLibrary* which is Win32 API loads the jvm.dll. The *GetProcAddress* which is Win32 API obtains the function pointer of the *JNI_CreateJavaVM* function.

After setting up the virtual machine initialization structure, the C program calls *JNI_CreateJavaVM* to load and initialize the Java Virtual Machine. It fills in two return values. An interface pointer, *jvm*, to the newly created Java Virtual Machine. The *JNIEnv* interface pointer *env* for the current thread. Recall that native code accesses JNI functions through the env interface pointer. So, the value of the *JNIEnv* interface pointer *env* is returned back to the assembly code.

We use mixing mode concepts of assembly codes and C functions in some books[25,26] which introduce the assembly language. The total flow of starting JVM is that the assembly code calls the startvm.c function, and then the startvm.c function puts the value of the *JNIEnv* interface pointer *env* on the register called eax, and then the assembly code obtains the starting address of the JNI function table through an indirect mapping. Figure 3.12 shows the assembly code calls the startvm.c function. Next section will introduce the JNI function table because we can call the Java api through JNI function from assembly codes.

```
…
EXTRN    _startvm:proc
…
.data
   jnienv                dword   ?
   fntblptr              dword   ?
   fnptr                 dword   ?
   …
.code
   public _main
   _main proc
         push   EBP
         mov    EBP,ESP
         call   _startvm
         mov    jnienv,eax
         mov    ebx,[eax]
         mov    fntblptr,ebx
         …
```

Figure 3.12 The assembly code invokes the startvm.c function

3.2.2 Call Java Bytecode From Assembly Code

When we want to call the system class file of Java Virtual Machine providing, we come to the object through the JNI function. Assembly codes pass parameters to the Java API through the native stack, and the Java API puts the return value on the register called eax. Figure 3.13 is the sketch of the assembly code call the bytecode. The full name of the JNI is Java Native Interface.

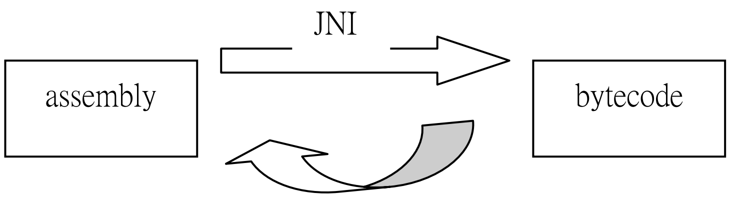


Figure 3.13 The sketch of the assembly code invokes the bytecode

First, we introduce the structure of the JNI function table[24]. The JNI is a two-way interface that allows Java applications to invoke native code and vice versa. The JNI is the native interface supported by all Java Virtual Machine implementations. The use of the JNI is not limited to specific host environments or specific application development tools. Figure 3.14 illustrates the JNIEnv interface pointer. The JNIEnv interface pointer, points to a location that contains a pointer to a function table. Each entry in the function table points to a JNI function. Native methods always access data structures in the Java Virtual Machine through one of the JNI functions.

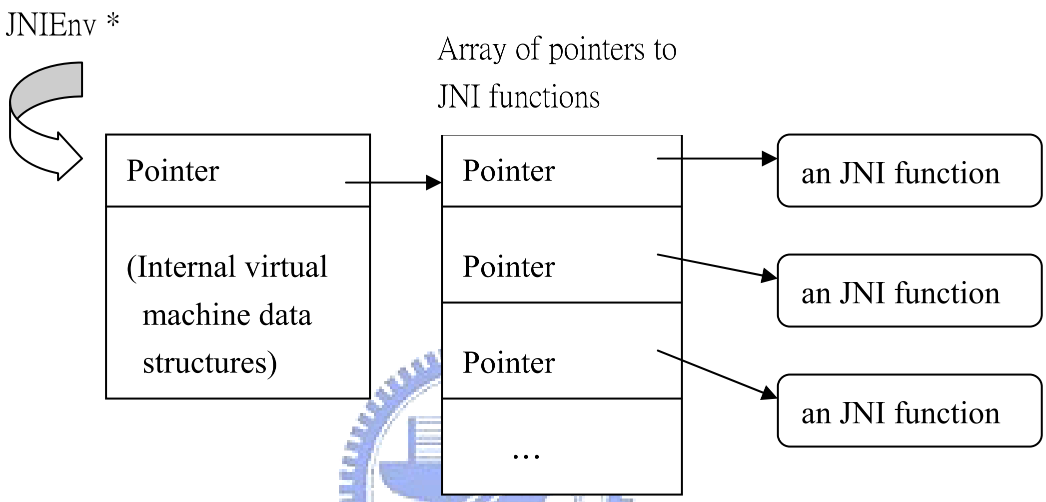


Figure 3.14 The JNIEnv interface pointer

For a developer using assembly language, it is necessary to understand clearly how the JNI provides the interface facilities[24,31]. According to figure 3.12 and 3.14, the *fntblptr* contains the starting address of the function table. There are three important points which are noticed. The first, we need to retrieve the contents of the entry in the function table which corresponds to the function we want to call. The content is the pointer which points to the JNI function we want to call and saved in *fnptr*, and we use the expression "call [fnptr]" to actually call the JNI function. Obviously, we have to multiply the zero based index of the function by 4 (since each pointer is 4 bytes long) and add the result to the starting address of the function table which we have formed in *fntblptr* earlier. The prototype of each JNI function in JNI function table is defined in the *jni.h* which is in J2SDK, and the index of each JNI function in JNI function table is the offset which is the place where the JNI function defined in the *jni.h*. We use a Macro[31] to get the entry of JNI function in JNI function table, and the following expression is the Macro. We obtain the pointer of the JNI function through the Macro expression.

```
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
```

The second, we use native stack to pass parameters to JNI function according to every JNI function prototype. The style of passing parameters is from right to left as assembly calling C function. Note that the rightmost parameter is pushed first in accordance with the stdcall convention followed by JNI. If you call the Java method through JNI function, you have to adjust esp to the place which is before calling the Java method, after calling the Java method. If you do not call the Java method, and only use JNI functions to do something, such as getting field value, parameters will be taken from native stack, and so you do not have to adjust esp. The third, If the type of the return value is long, the return value occupies two registers, otherwise occupies one register. If the type of the return value is float or double, the return value is on the floating-point register of the FPU. We will introduce detailed internal operations in later sections. Figure 3.15 illustrates the example of call Java bytecode from assembly code.

```
class sysjnicall {
        public static void main(String[] args) {
              System.out.println("abcd");
        }
}
```

Java source code

```
Method void main(java.lang.String[])
    0 getstatic #2 <Field java.io.PrintStream out>
    3 ldc #3 <String "abcd">
    5 invokevirtual #4 <Method void println(java.lang.String)>
    8 return
```

Javap of Java source code

```
…
.data
   …
   argreversebuf          dword   20   dup(?)
   jclsjobjtmp            dword   ?
   …
   index_2_clsname    db       'java/lang/System',0
   index_2_fidname    db       'out',0
   index_2_fidsig     db       'Ljava/io/PrintStream;',0
   index_2_fid        dword    ?
   index_3_strname    db       'abcd',0
   index_3_str        dword    ?
   index_4_midname    db       'println',0
   index_4_midsig     db       '(Ljava/lang/String;)V',0
   index_4_mid        dword    ?
.code
        …
        GetFnPtr    fntblptr,145,fnptr             ;GetStaticObjectField
        push        index_2_fid
        push        index_2_cls
        push        jnienv
        call        [fnptr]
        push        eax
        GetFnPtr    fntblptr,167,fnptr             ;NewStringUTF
        push        offset index_3_strname
        push        jnienv
        call        [fnptr]
        mov         index_3_str,eax
        …
        GetFnPtr    fntblptr,61,fnptr              ;CallVoidMethod
        push        [argreversebuf+0*4]
        push        index_4_mid
        push        jclsjobjtmp
        push        jnienv
        call        [fnptr]
        …
```

Figure 3.15 Example of call Java bytecode from assembly code

3.2.3 General Bytecode Translation

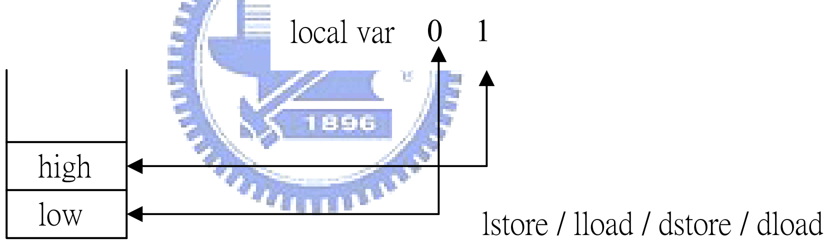
In figure 3.11, we mention that there are six bytecode types. Among them, four types which are stack and local variables、array、arithmetic and logic and conversion、flow control, belong to general bytecodes. The so-called general bytecodes imply that those bytecode instructions are not associated with object directly.

3.2.3.1 Stack and Local Variables

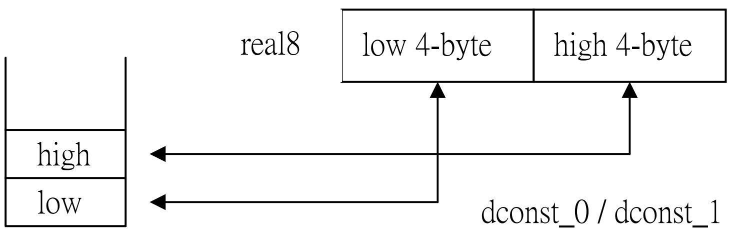
If the type of the bytecode instruction is the kind of less 32 bits, the operand will be sign-extended to an int value, and is then pushed onto the native stack.

```
bipush 3 or sipush 3  ——▶  mov eax,3
                           push eax
```

If the type of the bytecode instruction is the kind of 64 bits, the operand is the long or double. The operand which is long or double is split into two parts. For understanding easily, we illustrate the operation rules.




```
lconst_1  ——▶  mov eax,1
               push eax          ; push low 4-byte
               mov eax,0
               push eax          ; push high 4-byte
```



For long type operand, we split it into two parts. The low part is from bit 0 to bit 31, and the high part is from bit 32 to bit 63. We first push low part and then push high part. When you store the long operand to local variables, the lower index of the local variable array is stored the lower part of long operand, and the higher index of the local variable array is stored the higher part of long operand. For double type operand, we split it into two parts. The memory organization of the real8 type is first the low 4-byte and then high 4-byte. The operation rule of the double type operand is as the long type operand. We first push the low part and then push the high part.

For these bytecode instructions of *ldc* series, we push item from constant pool. The item is retrieved from constant pool table which is described in section 3.1.2. If the item is integer type, we generate a variable called *index_.._intname* which is the integer binary representation in .data section, and then push it onto the native stack. If the item is float type, we generate a variable called *index_.._floatname* which is the float binary representation in .data section, and then push it onto the native stack. If the item is long type, we generate two variables called *index_.._longhighname* and *index_.._longlowname* which are the long binary representation in .data section, and then push the *index_.._longlowname* and push *index_.._longhighname*. If the item is double type, we generate two variables called *index_.._doublehighname* and *index_.._doublelowname* which are the double binary representation in .data section, and then push the *index_.._doublelowname* and push *index_.._doublehighname*. Anyhow, first push low part and then push high part. If the item is the string type, we generate two variables called *index_.._strname* and *index_.._str* in .data section. We use the JNI function called *NewStringUTF* to construct a new *java.lang.String* object from *index_.._strname*, and then put the reference pointer in the *index_.._str*, and push it onto the native stack. (the notation "..” is the index of the item from constant pool)

Figure 3.16 illustrates the example of stack and local variable associated bytecodes translating.

```
class Fibonacci {
        public static void main(String[] args) {
                int fibonum=1;      int a=1;      int b=1;
                for(;;) {
                        fibonum=a+b;
                        a=b;
                        b=fibonum;
                        if(fibonum > 1000)
                                break;          }
                System.out.println("ok");
        }   }
```

```
Method void main(java.lang.String[])
   0 iconst_1
   1 istore_1
   2 iconst_1
   3 istore_2
   4 iconst_1
   5 istore_3
   6 goto 9
   9 iload_2
  10 iload_3
  11 iadd
  12 istore_1
  13 iload_3
  14 istore_2
  15 iload_1
  16 istore_3
  17 iload_1
  18 sipush 1000
  21 if_icmple 9
  24 goto 27
  27 getstatic #2 <Field java.io.PrintStream out>
  30 ldc #3 <String "ok">
  32 invokevirtual #4 <Method void println(java.lang.String)>
  35 return
```

```
…
.data
  …
  freal0            real4   0.0
  freal1            real4   1.0
  freal2            real4   2.0
  dreal0            real8   0.0
  dreal1            real8   1.0
  real4buf          real4   ?
  real8buf          real8   ?
  aa_return_addr    dword   ?
  aa_local_vars     dword   4   dup(?)
```

```
  index_3_strname     db    'ok',0
  index_3_str         dword    ?
  …
.code
  …
        mov    [aa_local_vars+1*4],1
        mov    [aa_local_vars+2*4],1
        mov    [aa_local_vars+3*4],1
   aa_9:
        push   [aa_local_vars+2*4]
        push   [aa_local_vars+3*4]
        pop    eax
        pop    ebx
        add    eax,ebx
        push   eax
        pop    [aa_local_vars+1*4]
        mov    eax,[aa_local_vars+3*4]
        mov    [aa_local_vars+2*4],eax
        mov    eax,[aa_local_vars+1*4]
        mov    [aa_local_vars+3*4],eax
        push   [aa_local_vars+1*4]
        mov    eax,1000
        push   eax
        pop    eax
        pop    ebx
        cmp    ebx,eax
        jle    aa_9
   aa_27:
        …
        GetFnPtr   fntblptr,167,fnptr            ; NewStringUTF
        push       offset index_3_strname
        push       jnienv
        call       [fnptr]
        mov        index_3_str,eax
        push       index_3_str
        …
```

Figure 3.16 Example of stack and local variable associated bytecodes translating

3.2.3.2 Array

We use some JNI functions to construct an array which may be one-dimension or two-dimension or multi-dimension, and retrieving the element of the array is also use the JNI function. There three kind of bytecode instructions for creating an array. The *newarray* constructs an one-dimension array whose the type of the element is primitive type (boolean、char、float、double、byte、short、int、long). The *anewarray* constructs an one-dimension array whose the type of the element is reference type. The *multianewarray* constructs a multi-dimension array, and you can repeatedly use the *anewarray* to reach the function which the *multianewarray* provides. The table 3.1 shows the JNI function relationship of bytecodes associated array. Our translator use these JNI functions to finish the operations which are characterization in these bytecode instructions associated array.

Bytecod instruction	JNI function
newarray	New<Type>Array , 175 ~ 182
anewarray	NewObjectArray , 172
iaload	GetIntArrayRegion , 203
laload	GetLongArrayRegion , 204
faload	GetFloatArrayRegion , 205
daload	GetDoubleArrayRegion , 206
aaload	GetObjectArrayElement , 173
baload	GetByteArrayRegion , 200
caload	GetCharArrayRegion , 201
saload	GetShortArrayRegion , 202
iastore	SetIntArrayRegion , 211
lastore	SetLongArrayRegion , 212
fastore	SetFloatArrayRegion , 213
dastore	SetDoubleArrayRegion , 214
aastore	SetObjectArrayElement , 174
bastore	SetByteArrayRegion , 208
castore	SetCharArrayRegion , 209
sastore	SetShortArrayRegion , 210
arraylength	GetArrayLength , 171

Table 3.1 The JNI function relationship of bytecodes associated array

Figure 3.17 illustrates example of the primitive array. Our translator is tested in many different types, but these contents is too long, and we only show the part of contents in this paper.

```
class primitivearraytest {
    public static void main(String[] args) {
        double[] ir=new double[10];
        for(int i=0;i<ir.length;i++)
            ir[i]=16.23;
    } }
```

```
Method void main(java.lang.String[])
    0 bipush 10
    2 newarray double
    4 astore_1
    5 iconst_0
    6 istore_2
    7 goto 19
   10 aload_1
   11 iload_2
   12 ldc2_w #2 <Double 16.23>
   15 dastore
   16 iinc 2 1
   19 iload_2
   20 aload_1
   21 arraylength
   22 if_icmplt 10
   25 return
```

```
  …
  index_2_doublehighname EQU    01000000001100000011101011100001b
  index_2_doublelowname    EQU    01000111101011100001010001111011b
.code
      …
      GetFnPtr      fntblptr,182,fnptr                    ; NewDoubleArray
```

```
	push	jnienv
	call	[fnptr]
	push	eax
	pop	[aa_local_vars+1*4]
	mov	[aa_local_vars+2*4],0
	jmp	aa_19
aa_10:
	push	[aa_local_vars+1*4]
	push	[aa_local_vars+2*4]
	push	index_2_doublelowname
	push	index_2_doublehighname
	pop	dword ptr real8buf+4
	pop	dword ptr real8buf
	pop	[argreversebuf]
	pop	[argreversebuf+4]
	GetFnPtr	fntblptr,214,fnptr		; SetDoubleArrayRegion
	push	offset real8buf
	mov	eax,1
	push	eax
	push	[argreversebuf]
	push	[argreversebuf+4]
	push	jnienv
	call	[fnptr]
	add	[aa_local_vars+2*4],1
aa_19:
	push	[aa_local_vars+2*4]
	push	[aa_local_vars+1*4]
	GetFnPtr	fntblptr,171,fnptr		; GetArrayLength
	push	jnienv
	call	[fnptr]
	push	eax
	pop	eax
	pop	ebx
	cmp	ebx,eax
	jl	aa_10
	…
```

Figure 3.17 Example of the primitive array

The reference type of the element of *anewarray* and *multianewarray* is associated with the constant pool. To get the type we must retrieve the entry of the constant pool table. The *multianewarray* is complex. For finishing the function of *multianewarray*, we use the following rule to reach the destination.

```
aobjlist     dword   20   dup(?)
arangelist   dword   20   dup(?)
aindexlist   dword   20   dup(?)
```

we define these three one-dimension array, and can create the up to 20 level array. The aobjlist saves the temporary reference. The arangelist saves the range of each level. The aindexlist saves the index which is used to increment during constructing each level array. Our constructing rule is like DFS (depth first search), and is as the following figure 3.18. The number of above a block which represents an array is the constructing order.

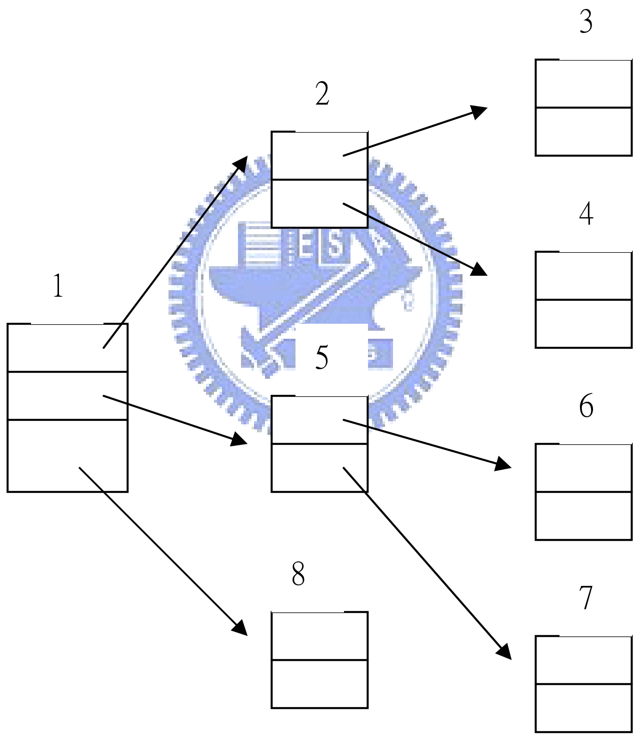


Figure 3.18 The constructing order of multianewarray implementation

Figure 3.19 illustrates example of the three dimension array. The *aaload* uses the procedure called *update_arrayelementobj* to maintain the latest reference which is loaded from the element of the array in the objmapclass structure.

```
class ThreeDTree {
        public static void main(String[] args) {
                int[][][] threeD = new int[5][4][3];
                for (int i = 0; i < 5; ++i) {
                        for (int j = 0; j < 4; ++j) {
                                for (int k = 0; k < 3; ++k) {
                                        threeD[i][j][k] = i + j + k;
                                        System.out.print(threeD[i][j][k]);
                                }       }       }       }       }
```

```
Method void main(java.lang.String[])
     0 iconst_5
     1 iconst_4
     2 iconst_3
     3 multianewarray #2 dim #3 <Class [[[I>
     7 astore_1
     8 iconst_0
     9 istore_2
    10 goto 72
    13 iconst_0
    14 istore_3
    15 goto 64
    18 iconst_0
    19 istore 4
    21 goto 55
    24 aload_1
    25 iload_2
    26 aaload
    27 iload_3
    28 aaload
    29 iload 4
    31 iload_2
    32 iload_3
    33 iadd
    34 iload 4
    36 iadd
    37 iastore
```

```
…
52 iinc 4 1
55 iload 4
57 iconst_3
58 if_icmplt 24
61 iinc 3 1
64 iload_3
65 iconst_4
66 if_icmplt 18
69 iinc 2 1
72 iload_2
73 iconst_5
74 if_icmplt 13
77 return
```

```
…
    aobjlist            dword   20  dup(?)
    arangelist          dword   20  dup(?)
    aindexlist          dword   20  dup(0)
    …
    index_2_array0    db    '[[I',0
    index_2_array1    db    '[I',0
    index_2_cls    dword    ?
.code
        …
        mov eax,5
        push eax
        mov eax,4
        push eax
        mov eax,3
        push eax
        pop     [arangelist+2*4]
        pop     [arangelist+1*4]
        pop     [arangelist+0*4]
        GetFnPtr    fntblptr,6,fnptr                    ; FindClass
        push    offset index_2_array0
        push    jnienv
```

```
    call      [fnptr]
    mov       index_2_cls,eax
    GetFnPtr      fntblptr,172,fnptr                    ; NewObjectArray
    mov       eax,0
    push      eax
    push      index_2_cls
    mov       eax,[arangelist+0*4]
    push      eax
    push      jnienv
    call      [fnptr]
    mov       [aobjlist+0*4],eax
    mov       [aindexlist+0*4],0
aa_3_L1:
    GetFnPtr      fntblptr,6,fnptr                      ; FindClass
    push      offset index_2_array1
    push      jnienv
    call      [fnptr]
    mov       index_2_cls,eax
    GetFnPtr      fntblptr,172,fnptr                    ; NewObjectArray
    mov       eax,0
    push      eax
    push      index_2_cls
    mov       eax,[arangelist+1*4]
    push      eax
    push      jnienv
    call      [fnptr]
    mov       [aobjlist+1*4],eax
    GetFnPtr      fntblptr,174,fnptr            ; SetObjectArrayElement
    push      [aobjlist+1*4]
    mov       eax,[aindexlist+0*4]
    push      eax
    push      [aobjlist+0*4]
    push      jnienv
    call      [fnptr]
    mov       [aindexlist+1*4],0
aa_3_L2:
    GetFnPtr      fntblptr,179,fnptr            ; NewIntArray
    mov       eax,[arangelist+2*4]
```

```
    push    eax
    push    jnienv
    call    [fnptr]
    mov     [aobjlist+2*4],eax
    GetFnPtr    fntblptr,174,fnptr          ; SetObjectArrayElement
    push    [aobjlist+2*4]
    mov     eax,[aindexlist+1*4]
    push    eax
    push    [aobjlist+1*4]
    push    jnienv
    call    [fnptr]
    inc     [aindexlist+1*4]
    mov     esi,[aindexlist+1*4]
    cmp     esi,[arangelist+1*4]
    jl      aa_3_L2
    inc     [aindexlist+0*4]
    mov     esi,[aindexlist+0*4]
    cmp     esi,[arangelist+0*4]
    jl      aa_3_L1
    push    [aobjlist]
    pop     [aa_local_vars+1*4]
    mov     [aa_local_vars+2*4],0
    jmp     aa_72
aa_13:
    mov     [aa_local_vars+3*4],0
    jmp     aa_64
aa_18:
    mov     [aa_local_vars+4*4],0
    jmp     aa_55
aa_24:
    …
    push    eax
    push    offset aa_objmapclass
    push    eax
    call    update_arrayelementobj
    …
```

Figure 3.19 Example of the three dimension array

3.2.3.3 Arithmetic, Logic, Type Conversion

Because these bytecode instructions associated this section are too many, we only introduce important concepts. 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 multiplication and division do not have any assembly instruction to help implementation. We use the multiplication algorithm 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.

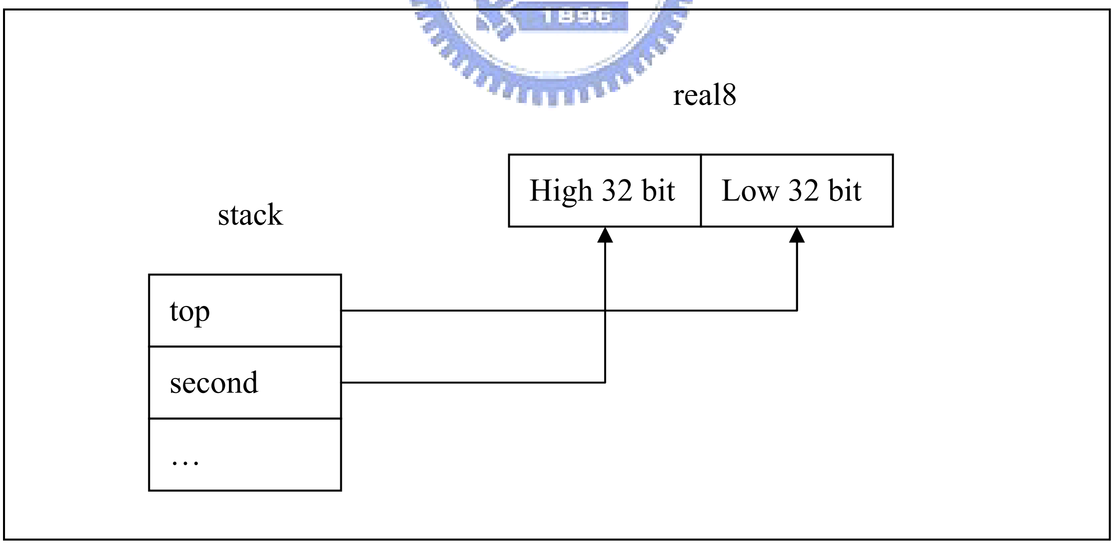
```
P=0
A=multiplier
B=multiplicand
Count=64
    While(count>0)
        If(LSB of A=1)
        Then
            P=P+B
            CF=carry generated by P+B
        Else
            CF=0
        End if
        Shift right CF:P:A by one bit position
    Count=count-1
    End while
```

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. 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, depending on the sign of P, either adding or subtracting B from P. Then P:A is left shifted 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=A+B
    Else
        Shift left P:A by one bit position
        P=A-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
```

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

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 called FPU. The FPU is not a pure stack because FPU is consisted of eight data register, each 80 bits long. The special stack can convert any nonfloating formats including 64 bit 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 stack. In 64 bit instruction, The top data of stack is high 32 bit and the second data of stack is low 32 bit, but the real8 data type the low 32 bit puts the top data of stack and the high 32 bit puts the second data type.



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

The ferm and drem bytecode instructions are not the same as that 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.

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 an infinity or divisor is a zero or both, the result is NaN.
(4) If the dividend is finite and the divisor is an infinity, the result equals the dividend.
(5) If the dividend is a zero and the divisor is finite, the result equals the dividend.
(5) 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.

The convert floating-point to integer instructions are generated by the following the rules in Java specification.
(1) If the value is NaN, the result of the conversion is an int0.
(2) Otherwise, if the value is not an 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

3.2.3.4 Flow Control

We have described the *branch table* and *jsr ret table* in section 3.1.2. Before we want to translate the bytecode instruction, we search contents of the two tables, and check the offset of the bytecode instruction to decide that if match the content. If the offset is in the content, the translator generate a label. The label format is *.._offset*. The symbol .. is the short name of the method. Because it can not has the same label in a .asm file, the symbol .. avoids the problem. Figure 3.20 illustrates the example of flow control.

```
class PrimeFinder {
    public static void main(String[] args) {
        int primeNum = 1;
        int numToCheck = 2;
        for (;;) {
            boolean foundPrime = true;
            for (int divisor = numToCheck / 2; divisor > 1;
                --divisor) {
                if (numToCheck % divisor == 0) {
                    foundPrime = false;
                    break;
                }
            }
            if (foundPrime) {
                primeNum = numToCheck;
                System.out.println(primeNum);
                if(primeNum > 50)
                   break;
            }
            ++numToCheck;
        }
    }
}
```

```
  …
  4 goto 7
  7 iconst_1
  …
 14 goto 32
 17 iload_2
 …
 21 ifne 29
 …
 26 goto 38
 29 iinc 4 -1
 32 iload 4
 34 iconst_1
 35 if_icmpgt 17
 38 iload_3
 39 ifeq 60
 …
 54 if_icmple 60
 57 goto 66
 60 iinc 2 1
 63 goto 7
 66 return
```

```
…
.code
        …
    aa_7:
        mov     [aa_local_vars+3*4],1
        …
        jmp     aa_32
    aa_17:
        …
        cmp     eax,0
        jnz     aa_29
        mov     [aa_local_vars+3*4],0
        jmp     aa_38
    aa_29:
        add     [aa_local_vars+4*4],-1
```

```
aa_32:
    push     [aa_local_vars+4*4]
    …
    cmp      ebx,eax
    jg       aa_17
aa_38:
    mov      eax,[aa_local_vars+3*4]
    cmp      eax,0
    jz       aa_60
    …
    cmp      ebx,eax
    jle      aa_60
    jmp      aa_66
aa_60:
    add      [aa_local_vars+2*4],1
    jmp      aa_7
aa_66:
    …
```

Figure 3.20 Example of flow control

The switch .. case structure in a Java program is implemented by the *lookupswitch* or *tableswitch*. According to the Java Virtual Machine Specification[5], there are a little difference among them. Our translator can translate them. We introduce an example to explain them, as figure 3.21.

```
class Struggle {
    public final static char TOMAYTO = 'a';
    public final static char TOMAHTO = 'b';
    public static void main(String[] args) {
        char say = TOMAYTO;
        for (;;) {
            switch (say) {
            case TOMAYTO:
                say = TOMAHTO;
                break;
            case TOMAHTO:
                say = TOMAYTO;
                break;       }       }       }       }
```

```
…
6 iload_1
7 lookupswitch 2: default=41
    97: 32
    98: 38
…
```

```
aa_6:
    push   [aa_local_vars+1*4]
    pop    eax
    cmp    eax,97
    je     aa_32
    cmp    eax,98
    je     aa_38
    jmp    aa_41
aa_32:
    mov    eax,98
    push   eax
    pop    [aa_local_vars+1*4]
    jmp    aa_41
aa_38:
    mov    eax,97
    push   eax
    pop    [aa_local_vars+1*4]
aa_41:
    jmp    aa_6
```

Figure 3.21 Example of lookupswitch

The *lcmp*、*fcmpl*、*fcmpg*、*dcmpl*、*dcmpg* are bytecode instructions which are comparable bytecodes under flow control. Figure 3.22 illustrates example of lcmp. For *fcmpl*, *fcmpg*, *dcmpl*, and *dcmpg*, we must use the status word of the FPU[28] to differentiate the difference. We define some values which are c3, c2, and c0 in the .data section to assist in processing those comparison operations associated with real number. Figure 3.23 illustrates example of dcmpl.

```
class lcmp {
    public static void main(String[] args) {
```

```
long i=1;
long j=0;
if(i > j)
{      i++;      }    }    }
```

```
…
6 lcmp
…
```

```
    …
    pushfd
    pop     ebx
    pop     edx
    pop     eax
    pop     edi
    pop     esi
    sub     esi,eax
    sbb     edi,edx
    js      aa_lcmp_6_sf
    test    esi,0ffffffffh
    jne     aa_lcmp_6_zf
    test    edi,0ffffffffh
    jne     aa_lcmp_6_zf
    mov     eax,0
    push    eax
    jmp     aa_lcmp_6_ok
aa_lcmp_6_zf:
    mov     eax,1
    push    eax
    jmp     aa_lcmp_6_ok
aa_lcmp_6_sf:
    mov     eax,-1
    push    eax
aa_lcmp_6_ok:
    push    ebx
    popfd
    pop     eax
```

```
        cmp     eax,0
        jle     aa_14
        push    [aa_local_vars+1*4]
        push    [aa_local_vars+2*4]
        mov eax,1
        push eax
        mov eax,0
        push eax
        pushfd
        …
```

Figure 3.22 Example of lcmp

```
   0 ldc2_w #2 <Double 2.3>
   3 dstore_1
   4 ldc2_w #4 <Double 5.9>
   7 dstore_3
   8 dload_1
   9 dload_3
  10 dcmpl
  11 ifle 14
  14 return
```

```
  …
  real4buf                real4    ?
  real8buf                real8    ?
  intbuf                  dword    ?
  longbuf                 qword    ?
  c3                      EQU      0100000000000000b
  c2                      EQU      0000010000000000b
  c0                      EQU      0000000100000000b
  …
  index_2_doublehighname EQU    01000000000000100110011001100110b
  index_2_doublelowname EQU     01100110011001100110011001100110b
  index_4_doublehighname EQU    01000000000101111001100110011001b
  index_4_doublelowname EQU     10011001100110011001100110011010b
.code
        …
```

```
    push    index_2_doublelowname
    push    index_2_doublehighname
    pop     [aa_local_vars+2*4]
    pop     [aa_local_vars+1*4]
    …
    push    [aa_local_vars+1*4]
    push    [aa_local_vars+2*4]
    push    [aa_local_vars+3*4]
    push    [aa_local_vars+4*4]
    pop     dword ptr real8buf+4
    pop     dword ptr real8buf
    .if     dword ptr real8buf+4 == 01111111111100000000000000000000b
        mov     eax,-1
        push    eax
        jmp     aa_dcmpl_10_end
    .endif
    fld     real8buf
    pop     dword ptr real8buf+4
    pop     dword ptr real8buf
    .if     dword ptr real8buf+4 == 01111111111100000000000000000000b
        mov     eax,-1
        push    eax
        jmp     aa_dcmpl_10_end
    .endif
    fld     real8buf
    fcom
    fstsw   ax
    mov     dx,ax
    test    ax,c3
    jnz     aa_dcmpl_10_equal
    mov     ax,dx
    test    ax,c0
    jnz     aa_dcmpl_10_small
    mov     ax,dx
    test    ax,c2
    jz      aa_dcmpl_10_big
aa_dcmpl_10_equal:
    mov     eax,0
```

```
        mov     edx,0
        push    eax
        jmp     aa_dcmpl_10_end
aa_dcmpl_10_small:
        mov     eax,-1
        mov     edx,0
        push    eax
        jmp     aa_dcmpl_10_end
aa_dcmpl_10_big:
        mov     eax,1
        mov     edx,0
        push    eax
aa_dcmpl_10_end:
        finit
        pop     eax
        cmp     eax,0
        jle     aa_14
aa_14:
```

Figure 3.23 Example of dcmpl

3.2.4 Special Bytecode Translation

The so-called special bytecodes imply that those bytecode instructions are associated with object directly.

3.2.4.1 Object Association

We will explain how to create object, and other bytecodes — getfield、putfield、getstatic、putstatic、checkcast、instanceof.

<u>Create Object</u>

A *new* Java instruction in the Java language implicitly generates several bytecode instructions after compiled by *javac*. For example：

```
                                              new #2
                                              dup
StringBuffer x=new StringBuffer(100);  ──→   bipush 100
                                              invokespecial #3
```

astore_1

The #2 and #3 are the entries of the constant pool.

#2：java/lang/StringBuffer

#3：java/lang/StringBuffer <init> (I)V

The *new* bytecode instruction decides the volumn, and allocates the memory space which is the part of the garbage collection. The fields of the created object are set to 0 or false or null. Finally, it push the reference of the created object onto the operand stack. Note that the created object is not be initialized, and it must be initialized through performing *invokespecial <init>*.

Figure 3.24 is the translated result of the above example through translator.

```
GetFnPtr    fntblptr,6,fnptr                  ; FindClass
push    offset index_2_clsname
push    jnienv
call    [fnptr]
mov     index_2_cls,eax
push    index_2_cls
pop eax
push eax
push eax
mov     eax,100
push    eax
pop     [argreversebuf+0*4]
pop     jclsjobjtmp
GetFnPtr    fntblptr,33,fnptr                 ; GetMethodID
push    offset index_3_midsig
push    offset index_3_midname
push    jclsjobjtmp
push    jnienv
call    [fnptr]
mov     index_3_mid,eax
GetFnPtr    fntblptr,28,fnptr                 ; NewObject
push    [argreversebuf+0*4]
push    index_3_mid
push    jclsjobjtmp
push    jnienv
call    [fnptr]
```

```
add     esp,20
push    eax
pop     [aa_local_vars+1*4]
```

Figure 3.24 Example of the assembly code of the new instruction

In assembly code, We use several JNI functions to finish the total flow of creating object. According to the above example, When the translator meets the new #2, it gets the value of entry 2 of the constant pool table, and generates two variables which are index_2_clsname and index_2_cls in the .data section. The index_2_clsname contains the value of entry 2 of the constant pool table. Then, we use the JNI function called *FindClass* to get a reference to the named class or interface. The type of the reference is *jclass*, and it stands for the loaded class, and saved in the index_2_cls. After getting the *jclass* reference, we will differentiate if the value of entry 2 of the constant pool table belongs to the system class which is the word "java/" starting. If it does not belong to system class, indicates that it is the user class, and its index will be maintained, and used for dynamic dispatching later. We detail the dynamic dispatching under user classes in next section.

Now, we have finished the translation of the *new* bytecode instruction, then process the *dup* bytecode instruction, and process the *bipush* bytecode instruction which is associated with parameters. Right now, we will process the *invokespecial*. This section only explain the *<init>* component of the *invokespecial* bytecode instruction, and other components will be explained in next section.

Now, we want to create an object through *invokespecial*. According to the above example, When the translator meets the invokespecial #3, it gets the value of entry 3 of the constant pool table, and generates three variables which are index_3_midname and index_3_midsig and index_3_mid in the .data section. The index_3_midname contains the method name of the entry 3 of the constant pool table, and the index_3_midsig contains the method signature of the entry 3 of the constant pool table. Then, we use the JNI function called *GetMethodId* to obtain the constructor id. Then, we use the JNI function called *NewObject* to create an object, and push the object reference onto the native stack.

```
index_2_clsname     db       'java/lang/StringBuffer',0
index_2_cls         dword    ?
index_3_midname     db       '<init>',0
index_3_midsig      db       '(I)V',0
index_3_mid         dword    ?
argreversebuf       dword    20   dup(?)
```

There are two places which are noticed especially. The one, the order of passing parameters into the JNI function called *NewObject* is from right to left. Because before the *invokespecial* appears, parameters are pushed onto the native stack from left to right. So, we must reverse the order of parameters by a temporary area called *argreversebuf* when call the *NewObject*. The order of passing parameter is the same rule when process other JNI functions. If the type of the parameter is float, we must change it to double. Because these calling methods in JNI functions regard the float as the double type. If you do not perform the action, you will make a mistake, and get the wrong result. The two, after performing the *NewObject*, you must adjust the register esp which points the native stack. Because these calling methods in JNI functions do not fetch the parameters from native stack by the register esp, and they fetch the parameters from native stack by the register ebp. The concept is like mixing mode of C and assembly[25,26]. Figure 3.25 illustrates the total flow of the translator creates an object. If the class of the object belong to user class, inserts the object reference into the objmapclass structure for dynamic dispatching later.

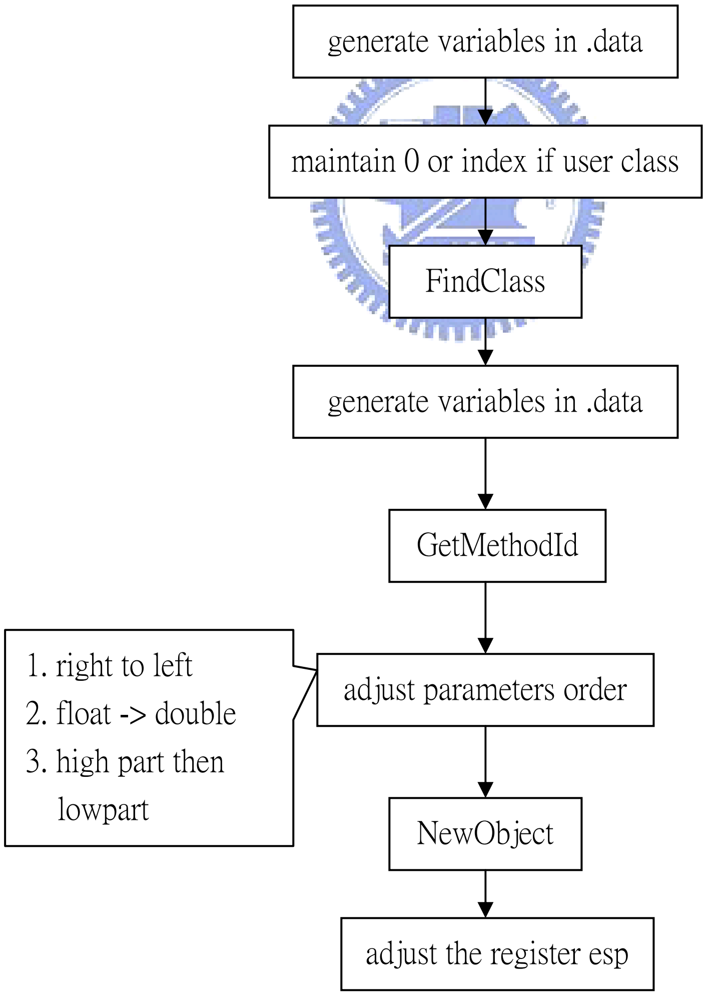


Figure 3.25 The total flow of the translator creates an object

getfield、putfield

The bytecode gets/assigns the value of the field which is instance field in an object. For example：

```
fieldexample x=new fieldexample();
x.i++;
```

```
0 new #3 <Class fieldexample>
3 dup
4 invokespecial #4 <Method fieldexample()>
7 astore_1
8 aload_1
9 dup
10 getfield #2 <Field int i>
13 iconst_1
14 iadd
15 putfield #2 <Field int i>
18 return
```

The #2 is the entries of the constant pool.

#2：fieldexample i I

A example of assembly code of getfield and putfield instructions is shown in Figure 3.26.

```
…
pop      jclsjobjtmp                      ; object reference
GetFnPtr     fntblptr,6,fnptr             ; FindClass
push     offset index_2_clsname
push     jnienv
call     [fnptr]
mov      index_2_cls,eax
GetFnPtr     fntblptr,94,fnptr            ; GetFieldID
push     offset index_2_fidsig
push     offset index_2_fidname
push     index_2_cls
push     jnienv
call     [fnptr]
mov      index_2_fid,eax
```

```
GetFnPtr    fntblptr,100,fnptr          ; Get<Type>Field
push    index_2_fid
push    jclsjobjtmp
push    jnienv
call    [fnptr]
push    eax
…                                       ; +1
pop     [argreversebuf]
pop     jclsjobjtmp                     ; object reference
GetFnPtr    fntblptr,6,fnptr            ; FindClass
push    offset index_2_clsname
push    jnienv
call    [fnptr]
mov     index_2_cls,eax
GetFnPtr    fntblptr,94,fnptr           ; GetFieldID
push    offset index_2_fidsig
push    offset index_2_fidname
push    index_2_cls
push    jnienv
call    [fnptr]
mov     index_2_fid,eax
GetFnPtr    fntblptr,109,fnptr          ; Set<Type>Field
push    [argreversebuf]
push    index_2_fid
push    jclsjobjtmp
push    jnienv
call    [fnptr]
…
```

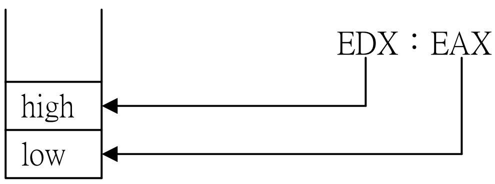
Figure 3.26 Example of assembly code of getfield and putfield instructions

We use several JNI functions to finish the bytecode *getfield* and *putfield*. According to the above example, When the translator meets the getfield #2, it gets the value of entry 2 of the constant pool table, and generates five variables which are index_2_clsname、index_2_cls、index_2_fidname、index_2_fidsig、index_2_fid in the .data section. The index_2_clsname contains the class name of entry 2 of the constant pool table. The index_2_fidname contains the field name of entry 2. The index_2_fidsig contains the field signature of entry 2. Then, we use the JNI function called *FindClass* to get a reference to the named class or interface. The type of the reference is *jclass*, and it stands for the loaded class, and saved in the index_2_cls. After getting the *jclass* reference, we use the the JNI function called *GetFieldID* to get the field id for an instance field of a class. Then we decide the jni function entry by the field signature which consists of *B*, *C*, *D*, *F*, *I*, *J*, *L*, *[*, *S*, and *Z*. Then, we call the *Get<Type>Field* which is relative to the field signature. For the *putfield*, we use the JNI function called *Set<Type>Field* to set the value of the instance field. Their translating basic rules are similar between *getfield* and *putfield*.

```
index_3_clsname     db        'fieldexample',0
index_3_cls         dword     ?
index_4_midname     db        '<init>',0
index_4_midsig      db        '()V',0
index_4_mid         dword     ?
index_2_clsname     db        'fieldexample',0
index_2_cls         dword     ?
index_2_fidname     db        'i',0
index_2_fidsig      db        'I',0
index_2_fid         dword     ?
real4buf            real4     ?
real8buf            real8     ?
argreversebuf       dword     20  dup(?)
```

There are three places which are noticed especially. The one, Because the hide of the field is a problem, we can not catch the class of the object directly during translating *getfield* and *putfield*. In the other word, When we get the parameter which is an object reference used to *getfield* or *putfield*, we can not use the JNI function called *GetObjectClass* directly to catch the class of the object. We must use the *FindClass* and then use the *GetFieldID* to catch the accurate field id. The two, after getting the value of the field, there are different processing rules for different return type.

If the type of the return value is long after *getfield*, it is saved in the edx：eax, and push the eax, then push edx. The edx has the high part of the value, and the eax has the low part of the value.



If the type of the return value is float or double, it is saved in the real register stack which belongs to the floating-point unit (FPU)[28]. The following are the translated codes, and if it is the double, we also push the low part, then push the high part.

```
float  ———→  fstp    real4buf
             push    real4buf
double ———→  fstp    real8buf
             mov     edx,dword ptr real8buf
             mov     eax,dword ptr real8buf+4
             push    edx                    ; edx has the low part
             push    eax                    ; eax has the high part
```

If it is the other types, we push the eax onto the native stack directly.

The three, the order of passing parameters into the JNI function is from right to left. If there is a parameter whose type is long or double, you must first push the high part onto the native stack and then push the low part onto the native stack as calling the JNI function truly. So, we use a temporary area called *argreversebuf* to finish those ordering works.

<u>getstatic、putstatic</u>

The bytecode gets/assigns the value of the field which is static field in a class. The translating flow of these two bytecodes are similar to *getfield* and *putfield*. The distinct point are that replace *GetFieldID* with *GetStaticFieldID*, and replace *Get<Type>Field* with *GetStatic<Type>Field*, and replace *Set<Type>Field* with *SetStatic<Type>Field*. The another distinct point is that we use the jclass reference to get the value of the static field, and do not use the object reference to get the value of the static field. Figure 3.27 illustrates the example of assembly code of getstatic and putstatic instructions.

```
class staticfieldexample {
        static int   i=1;
        public static void main(String[] args) {
                i++;
        }   }
```

```
0 getstatic #2 <Field int i>
3 iconst_1
4 iadd
5 putstatic #2 <Field int i>
8 return
```

```
…
index_2_clsname     db     'staticfieldexample',0
```

```
index_2_cls      dword     ?
index_2_fidname     db     'i',0
index_2_fidsig     db     'I',0
index_2_fid      dword     ?
        …
        GetFnPtr     fntblptr,6,fnptr                 ; FindClass
        push     offset index_2_clsname
        push     jnienv
        call     [fnptr]
        mov      index_2_cls,eax
        GetFnPtr     fntblptr,144,fnptr               ; GetStaticFieldID
        push     offset index_2_fidsig
        push     offset index_2_fidname
        push     index_2_cls
        push     jnienv
        call     [fnptr]
        mov      index_2_fid,eax
        GetFnPtr     fntblptr,150,fnptr               ; GetStatic<Type>Field
        push     index_2_fid
        push     index_2_cls
        push     jnienv
        call     [fnptr]
        push     eax
        …                                             ; +1
        GetFnPtr     fntblptr,6,fnptr                 ; FindClass
        push     offset index_2_clsname
        push     jnienv
        call     [fnptr]
        mov      index_2_cls,eax
        GetFnPtr     fntblptr,144,fnptr               ; GetStaticFieldID
        push     offset index_2_fidsig
        push     offset index_2_fidname
        push     index_2_cls
        push     jnienv
        call     [fnptr]
        mov      index_2_fid,eax
        GetFnPtr     fntblptr,159,fnptr               ; SetStatic<Type>Field
        push     index_2_fid
```

```
        push     index_2_cls
        push     jnienv
        call     [fnptr]
        …
```

Figure 3.27 Example of assembly code of getstatic and putstatic instructions

instanceof、checkcast

The bytecode test that if the object belongs to the some type. Figure 3.28 is the example of the instanceof instruction.

```
class isinstanceof {
     public static void main(String[] args) {
               isinstanceof a=new isinstanceof();
               if(a instanceof isinstanceof) {
               }
     }     }
```

```
  0 new #2 <Class isinstanceof>
  3 dup
  4 invokespecial #3 <Method isinstanceof()>
  7 astore_1
  8 aload_1
  9 instanceof #2 <Class isinstanceof>
 12 ifeq 15
 15 return
```

```
   ClassCastException_name    db    'java/lang/ClassCastException',0
   argreversebuf       dword    20    dup(?)
   jclsjobjtmp         dword    ?
   …
   index_2_clsname     db     'isinstanceof',0
   index_2_cls     dword     ?
   index_3_midname     db     '<init>',0
   index_3_midsig     db     '()V',0
   index_3_mid     dword     ?
…
        …
```

```
        push     [aa_local_vars+1*4]                ; aload_1
        GetFnPtr    fntblptr,6,fnptr                ; FindClass
        push     offset index_2_clsname
        push     jnienv
        call     [fnptr]
        mov      index_2_cls,eax
        GetFnPtr    fntblptr,32,fnptr               ; IsInstanceOf
        pop      eax
        push     index_2_cls
        push     eax
        push     jnienv
        call     [fnptr]
        push     eax
        pop      eax                                 ; ifeq 15
        cmp      eax,0
        jz       aa_15
    aa_15:
        …
```

Figure 3.28 Example of the instanceof instruction

We use the JNI function called *IsInstanceOf* to test. If the object belongs to the index_2_clsname, push 1 onto the native stack, otherwise push 0 onto the native stack. The value of 1 or 0 is used to judge by the bytecodes which belong to the flow control category.

For the *checkcast*, it does not push 1 or 0 onto the native stack. If the object does not belong to the some type, it will throws an exception called *ClassCastException*. We use the following statements to finish the function.

```
.if    eax == 0
        push     offset ClassCastException_name
        call     throw_exception_func
.endif



throw_exception_func      proc
.data
```

```
        e_name      dword      ?
        e_ref       dword      ?
.code
        pop            ebp
        pop            e_name
        GetFnPtr       fntblptr,6,fnptr
        push           e_name
        push           jnienv
        call           [fnptr]
        mov            e_ref,eax
        GetFnPtr       fntblptr,14,fnptr                  ; ThrowNew
        push           e_name
        push           e_ref
        push           jnienv
        call           [fnptr]
        push           ebp
        ret
throw_exception_func      endp
```

Table 3.2 illustrates the total JNI functions associated with object.

FindClass , 6
Get<Type>Field , 95 ~ 103
GetFieldID , 94
GetMethodId , 33
GetStaticFieldID , 144
GetStatic<Type>Field , 145 ~ 153
IsInstanceOf , 32
NewObject , 28
Set<Type>Field , 104 ~ 112
SetStatic<Type>Field 154 ~ 162
ThrowNew , 14

Table 3.2 The total JNI functions associated with object

3.2.4.2 Method Invocation

We will explain how to do dynamic dispatching, and other bytecodes — invokevirtual、invokestatic、invokespecial、invokeinterface、return series.

Dynamic Dispatch

To date, When we want to call the system API, we still use the JNI function to finish the work. But, When we want to call the user defined method, we must do the dynamic dispatching in the assembly code ourself. In order to call user defined methods in .ASM files, we design four components — *.._objmapclass*、*allclassmethodtable*、*build_objmapclasstable*、*search_objmapclasstable*.

```
objmapclass_table   struct
      jobj      dword      ?
      class     dword      ?
objmapclass_table          ends
aa_objmapclass   objmapclass_table   3   dup(<-1,-1>)
```

The *.._objmapclass* is used to save the created object whose class is user defined. Its *jobj* field saves the object reference, and its *class* field saves the index. If the class of the created object is same as thisclass which is the translated class file now, the index is 0. If the class of the created object is other user defined classes, the index is the entry of the class in the constant pool table mentioned in section 3.1.2. The allclassmethodtable mentioned in section 3.1.2 is generated in pass 1.

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

The allclassmethodtable has all the methods which come from every class which appears in the constant pool and their super class and super class up to java.lang.Object. A item is a class which appears in the constant pool, and it´s methods which including inheritance up to java.lang.Object. If the classname is same as thisclass, the idofclass is 0, otherwise idofclass is the entry number of the class in the constant pool table. The order of the methodname in a item is that methods of thisclass are first, then methods of superclass of thisclass are second, and then continue up to the java/lang/Object. When the translator meets those invoke series instructions in pass 2, the *allclassmethodtable* is useful.

```
build_objmapclasstable      proc
.data
                allobjmapclass_entry     dword     ?
                objref                        dword     ?
                objref_entry                  dword     ?
.code
                pop      ebp
                pop      objref_entry
                pop      objref
                pop      allobjmapclass_entry
                mov    ebx,dword ptr allobjmapclass_entry
      insert_item:
                cmp    (objmapclass_table ptr [ebx]).jobj,-1
                jne      next_item
                mov    eax,dword ptr objref
                mov    (objmapclass_table ptr [ebx]).jobj,eax
                mov    eax,dword ptr objref_entry
                mov    (objmapclass_table ptr [ebx]).class,eax
                jmp    insert_ok
       next_item:
                add    ebx,type objmapclass_table
                jmp    insert_item
       insert_ok:
                push           ebp
                ret
build_objmapclasstable      endp
```

The build_objmapclasstable is a procedure that is used to save the created object of user class in the *.._objmapclass*. When we new an object of an user defined class, the object reference and index are inserted into the *.._objmapclass* by the *build_objmapclasstable*. When the translator wants to translate *invokevirtual* or *invokeinterface*, the translator will generate the following codes to find the idofclass of the created object.

```
push     offset aa_objmapclass
push     eax                                 ; object reference is saved in eax
call     search_objmapclasstable
```

The *search_objmapclasstable* returns the idofclass of the created object, and the idofclass is associated with the *allclassmethodtable*.

```
search_objmapclasstable      proc
.data
        s_allobjmapclass_entry      dword     ?
        s_objref                          dword     ?
.code
        pop     ebp
        pop     s_objref
        pop     s_allobjmapclass_entry
        mov     edx,s_objref
        mov     ebx,s_allobjmapclass_entry
    s_search:
        ;check if matched ?
        cmp     (objmapclass_table ptr [ebx]).jobj,edx
        jne     s_nextitem
        mov    eax,(objmapclass_table ptr [ebx]).class
        jmp    s_finish
    s_nextitem:
        add    ebx,type objmapclass_table
        cmp    (objmapclass_table ptr [ebx]).jobj,-1
        je     s_notfind
        jmp    s_search
    s_notfind:
        mov    eax,-1
    s_finish:
        push          ebp
        ret
```

Because messages of the called method in the invoke series instructions are saved in the constant pool table, and we can use an entry number to get related data. There are method name and method signature in the data, and we make the method name and method signature as a key. The translator use the key to search the *allclassmethodtable* to show those possible calling methods. We use the .if directive concept in the assembler to make multiple calling decisions, as the following statements.

```
.if     eax == 2
        call    lower_obj_show_2
        mov     eax,2
.endif
.if     eax == 18
        call    upper_obj_show_2
        mov     eax,18
.endif
```

The eax has the return value of the procedure called *search_objmapclasstable*. When the idofclass is matched, the correct method is called. At this time, those parameters which will be used by the called function are already in the native stack, and the called function can use them from the native stack, as the following statements.

```
lower_obj_show_2      proc
    pop     aa_return_addr
    pop     [aa_local_vars+2*4]
    pop     [aa_local_vars+1*4]
    pop     [aa_local_vars+0*4]
    …
```

These parameters are saved in the local variables of the called function. The above rule also solves the overloading problem of the multiple inheritance. Because the actually called methods are decided at run time for *invokevirtual* and *invokeinterface*, and are associated with object. So, we must save the created object reference for dynamic dispatching later. But, for *invoekstatic* and *invokespecial* they are static binding, and the actually called methods are decided at compile time, and are not associated with object. So, the translator generates the calling assembly code directly, and do not make multiple calling decisions. The class name and method name and method signature from constant pool table are a key. The translator uses the key to search the *allclassmethodtable* to find the called method, and generates a calling method assembly code directly. Figure 3.29 illustrates the flow of dynamic dispatching for user methods.

When we want to call the system API, we still use the JNI function to finish the work. The flow of using the JNI function is like the contents described in section 3.2.4.1, and there are still some places which are noticed. These noticed places have explained in section 3.2.4.1. Figure 3.30 illustrates the translating flow of invoking system api.

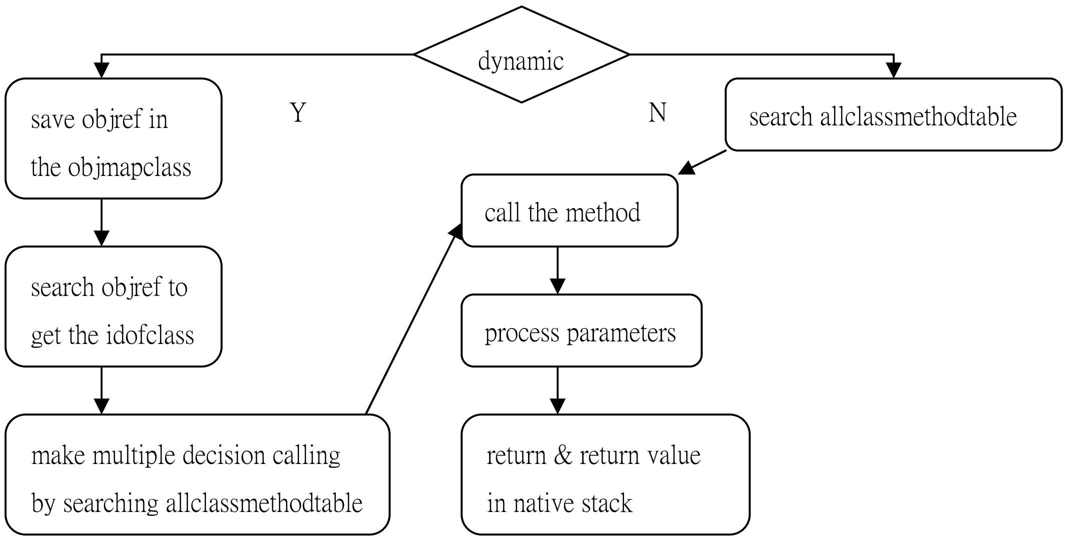


Figure 3.29 The flow of dynamic dispatching for user methods

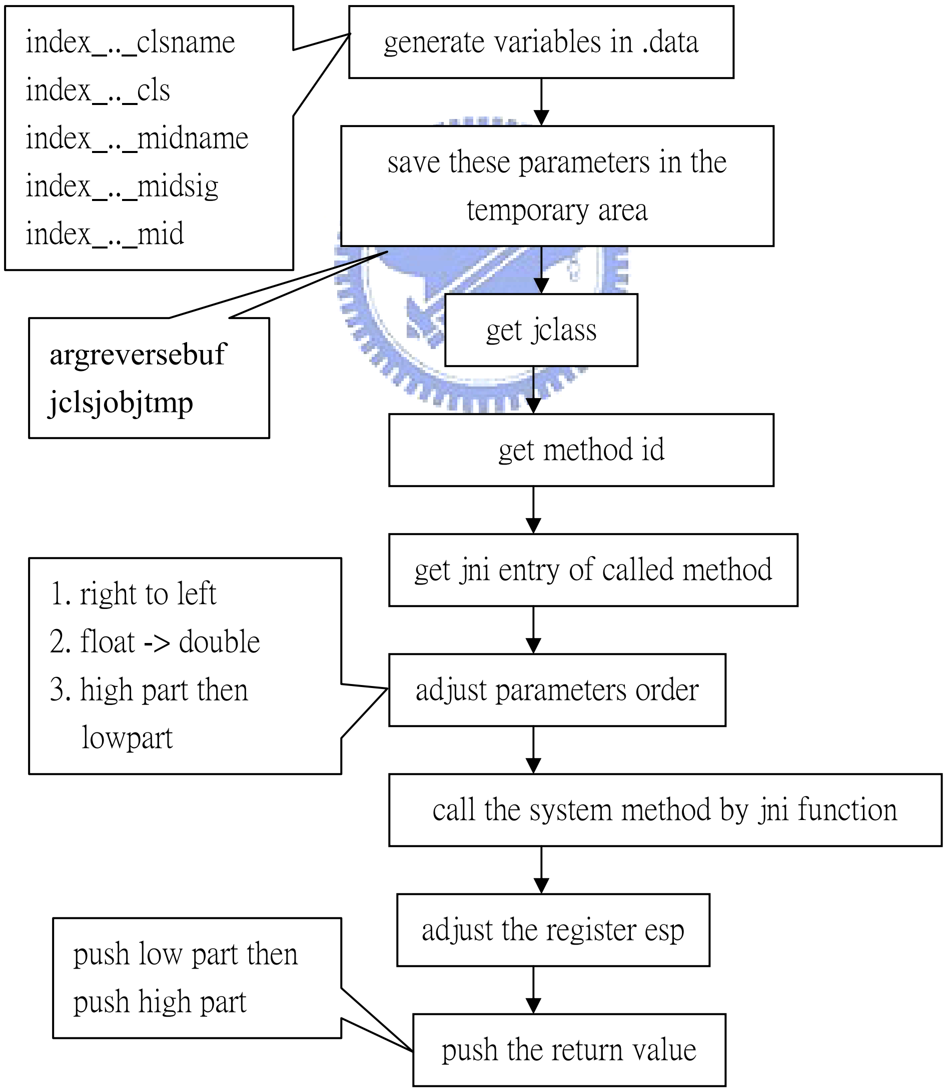


Figure 3.30 The translating flow of invoking system api

The “get jni entry of called method” is to decide the number of the jni function by the return types which consist of *B*、*C*、*D*、*F*、*I*、*J*、*L*、*[*、*S*、*Z*、*V*. For *invokestatic* or *invokespecial*, the “get jclass” use the JNI function called *FindClass* to get the class. For *invokevirtual* or *invokeinterface*, the “get jclass” use the JNI function called *GetObjectClass* to get the class. For *invokestatic*, the “get method id” use the JNI function called GetStaticMethodID to get the method id. For *invokespecial*, *invokevirtual* or *invokeinterface*, the “get method id” use the JNI function called *GetMethodId* to get the method id. Table 3.3 illustrates the JNI functions of method invocation.

Call<Type>Method , 61 34 37 40 43 46 49 52 55 58
CallStatic<Type>Method , 141 114 117 120 123 126 129 132 135 138
FindClass , 6
GetMethodID , 33
GetObjectClass , 31
GetStaticMethodID , 113

Table 3.3 The JNI functions of method invocation

invokevirtual

We show an example about calling user method and system method.

```
class   upper_obj {
          int upi;
          int show(int a,int b) {
               upi=a-b;
               return 1;   }   }
class lower_obj extends upper_obj {
        int i;
        int show(int a,int b) {
             i=a+b;
             return 2;    }   }
public class invokevirtualuserapi {
          public static void main(String[] args) {
                    lower_obj   a=new lower_obj();
                    upper_obj   b=new lower_obj();
                    int g=b.show(11,3);
                    System.out.println(g);     }     }
```

```
 0 new #2 <Class lower_obj>
 3 dup
 4 invokespecial #3 <Method lower_obj()>
 7 astore_1
 8 new #2 <Class lower_obj>
11 dup
12 invokespecial #3 <Method lower_obj()>
15 astore_2
16 aload_2
17 bipush 11
19 iconst_3
20 invokevirtual #4 <Method int show(int, int)>      //user method
23 istore_3
24 getstatic #5 <Field java.io.PrintStream out>
27 iload_3
28 invokevirtual #6 <Method void println(int)>      //system method
31 return
```

Javap of invokevirtualuserapi

```
…
extrn   lower_obj_show_2:proc,upper_obj_show_2:proc
.data
   …
   argreversebuf       dword   20   dup(?)
   jclsjobjtmp         dword   ?
   …
   index_6_cls      dword      ?
   index_6_midname      db      'println',0
   index_6_midsig      db      '(I)V',0
   index_6_mid      dword      ?
   aa_objmapclass   objmapclass_table    4    dup(<-1,-1>)
.code
         …                                          ; create object a
         push      offset aa_objmapclass
         push      eax
         mov       eax,2
         push      eax
```

```
call    build_objmapclasstable
pop     [aa_local_vars+1*4]
…                                               ; create object b
push    offset aa_objmapclass
push    eax
mov     eax,2
push    eax
call    build_objmapclasstable
pop     [aa_local_vars+2*4]
push    [aa_local_vars+2*4]
…                                               ; push 11 , 3
mov     eax,[esp+2*4]
push    offset aa_objmapclass
push    eax
call    search_objmapclasstable
.if     eax == 2
   call    lower_obj_show_2
   mov     eax,2
.endif
.if     eax == 19
   call    upper_obj_show_2
   mov     eax,19
.endif
pop     [aa_local_vars+3*4]
…                                               ; getstatic
push    [aa_local_vars+3*4]
pop     [argreversebuf+0*4]
pop     jclsjobjtmp
GetFnPtr    fntblptr,31,fnptr                   ; GetObjectClass
push    jclsjobjtmp
push    jnienv
call    [fnptr]
mov     index_6_cls,eax
GetFnPtr    fntblptr,33,fnptr                   ; GetMethodID
push    offset index_6_midsig
push    offset index_6_midname
push    index_6_cls
push    jnienv
```

```
        call      [fnptr]
        mov       index_6_mid,eax
        GetFnPtr      fntblptr,61,fnptr             ; CallVoidMethod
        push      [argreversebuf+0*4]
        push      index_6_mid
        push      jclsjobjtmp
        push      jnienv
        call      [fnptr]
        add       esp,16
        …
```

Asm code of invokevirtualuserapi

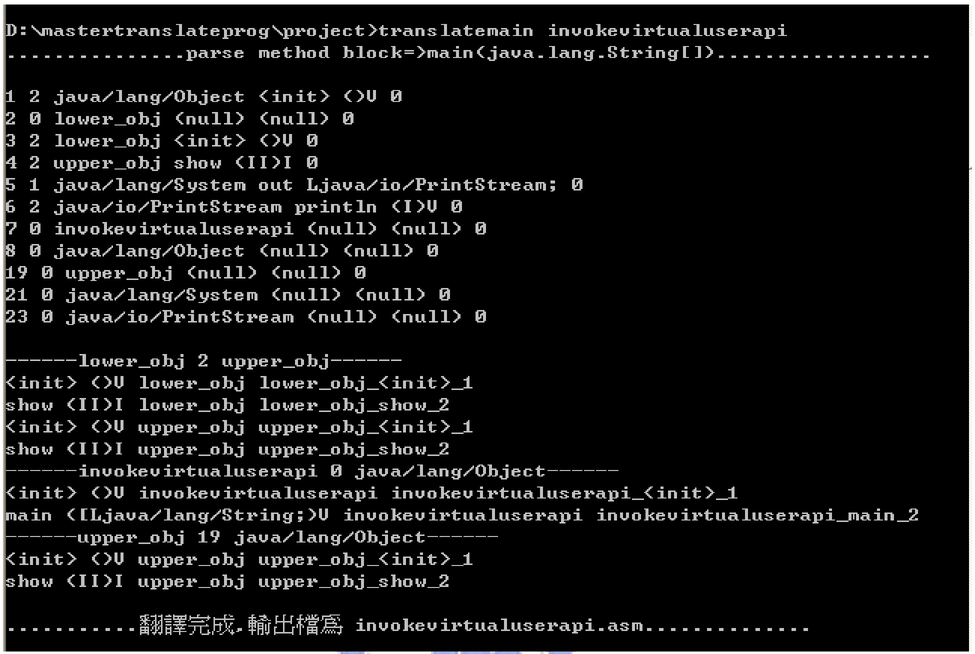
```
…
lower_obj_show_2        proc
        pop       aa_return_addr
        pop       [aa_local_vars+2*4]
        pop       [aa_local_vars+1*4]
        pop       [aa_local_vars+0*4]
        …
        mov eax,2
        push eax
        push      aa_return_addr
        ret
lower_obj_show_2          endp
```

Asm code of lower_obj

```
upper_obj_show_2        proc
        pop       aa_return_addr
        pop       [aa_local_vars+2*4]
        pop       [aa_local_vars+1*4]
        pop       [aa_local_vars+0*4]
        …
        mov eax,1
        push eax
        push      aa_return_addr
        ret
upper_obj_show_2          endp
```

Asm code of upper_obj

We show those generated assembly codes of the invokevirtualuserapi, lower_obj, and upper_obj. We also show the translating processes of the invokevirtualuserapi, lower_obj, and upper_obj, and also show the processes of compiling and executing.



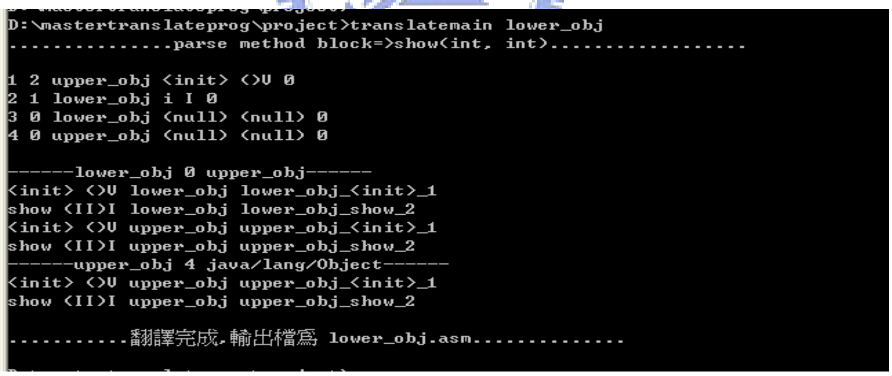
```
D:\mastertranslateprog\project>translatemain invokevirtualuserapi
...............parse method block=>main(java.lang.String[])..................

1 2 java/lang/Object <init> ()V 0
2 0 lower_obj <null> <null> 0
3 2 lower_obj <init> ()V 0
4 2 upper_obj show (II)I 0
5 1 java/lang/System out Ljava/io/PrintStream; 0
6 2 java/io/PrintStream println (I)V 0
7 0 invokevirtualuserapi <null> <null> 0
8 0 java/lang/Object <null> <null> 0
19 0 upper_obj <null> <null> 0
21 0 java/lang/System <null> <null> 0
23 0 java/io/PrintStream <null> <null> 0

------lower_obj 2 upper_obj------
<init> ()V lower_obj lower_obj_<init>_1
show (II)I lower_obj lower_obj_show_2
<init> ()V upper_obj upper_obj_<init>_1
show (II)I upper_obj upper_obj_show_2
------invokevirtualuserapi 0 java/lang/Object------
<init> ()V invokevirtualuserapi invokevirtualuserapi_<init>_1
main ([Ljava/lang/String;)V invokevirtualuserapi invokevirtualuserapi_main_2
------upper_obj 19 java/lang/Object------
<init> ()V upper_obj upper_obj_<init>_1
show (II)I upper_obj upper_obj_show_2

...........翻譯完成,輸出檔爲 invokevirtualuserapi.asm..............
```

translate the invokevirtualuserapi



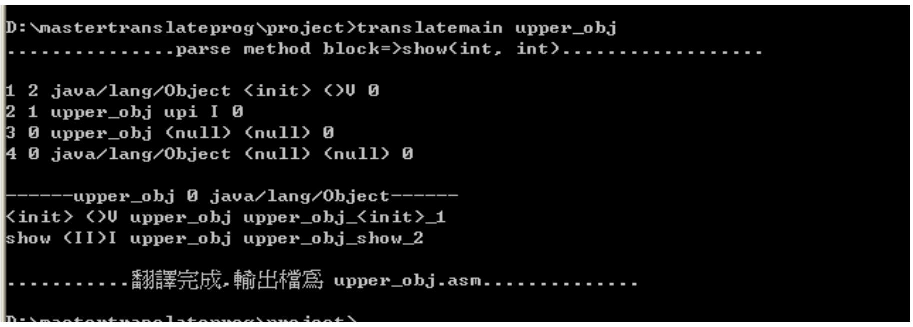
```
D:\mastertranslateprog\project>translatemain lower_obj
...............parse method block=>show(int, int)..................

1 2 upper_obj <init> ()V 0
2 1 lower_obj i I 0
3 0 lower_obj <null> <null> 0
4 0 upper_obj <null> <null> 0

------lower_obj 0 upper_obj------
<init> ()V lower_obj lower_obj_<init>_1
show (II)I lower_obj lower_obj_show_2
<init> ()V upper_obj upper_obj_<init>_1
show (II)I upper_obj upper_obj_show_2
------upper_obj 4 java/lang/Object------
<init> ()V upper_obj upper_obj_<init>_1
show (II)I upper_obj upper_obj_show_2

...........翻譯完成,輸出檔爲 lower_obj.asm..............
```

translate the lower_obj



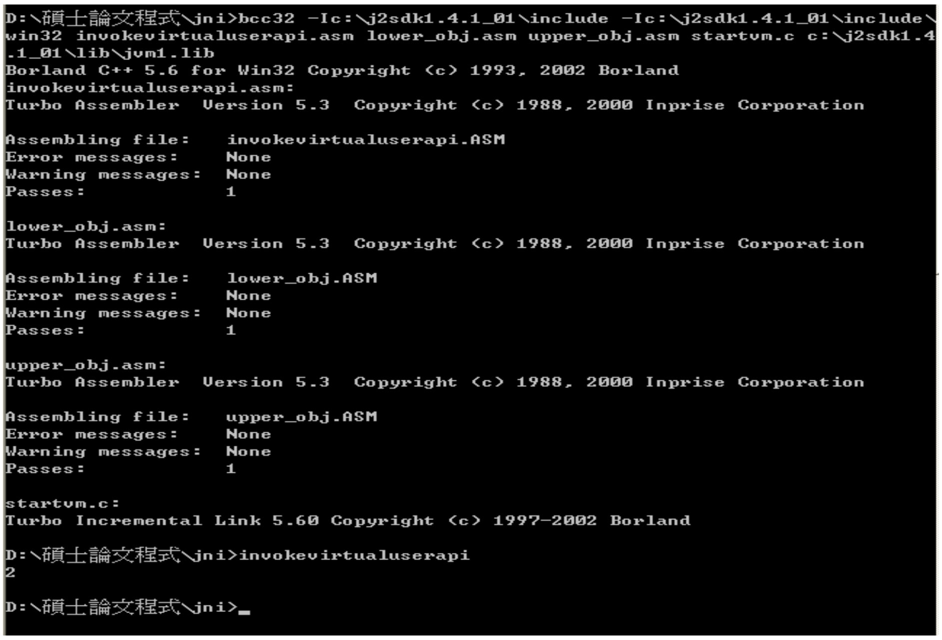
```
D:\mastertranslateprog\project>translatemain upper_obj
...............parse method block=>show(int, int)..................

1 2 java/lang/Object <init> ()V 0
2 1 upper_obj upi I 0
3 0 upper_obj <null> <null> 0
4 0 java/lang/Object <null> <null> 0

------upper_obj 0 java/lang/Object------
<init> ()V upper_obj upper_obj_<init>_1
show (II)I upper_obj upper_obj_show_2

...........翻譯完成,輸出檔爲 upper_obj.asm..............
```

translate the upper_obj



compile and run

invokestatic

The instruction is used to invoke static method. We can known the called method at compile time, and the message of the called method is recorded in the constant pool, and we can get the message through a index. We had resolved the class file in pass 1, and make the constant pool into a structure called constant pool table. For example：

invokestatic #8 // static user method: abc

When the translator want to translate this statement, it regards the number behind the symbol # as an entry of the constant pool table, and gets messages about the method by the entry. The messages contain class name, method name, and method signature. If the method is user, we use the three informations to search the *allclassmethodtable* to find the called method, and then generate the assembly calling code directly. The problems of parameters and returning value are the same operations as the *invokevirtual*.

call abc

If the method is system, the translating step is like the figure 3.30. But there are two different points. Because the *invokestatic* is related to class, and is not related to object. So, we do not have object reference, therefore we only use the JNI function called *FindClass* to get the *jclass*, and can not use the JNI function called *GetObjectClass*. Then we use the JNI function called *GetStaticMethodID* to get the method id. Because we call the different JNI function for static or non-static method, we use those JNI functions called *CallStatic<Type>Method*.

invokespecial

The instruction is used to call the following methods：

1. instance initialization method <init>.
2. private instance methods of this class.
3. methods of super class of this class.

We had described how to translate the *invokespecial* instruction to finish the <init> in the previous section. Now we will explain the another two usages. When the Java programmer uses the key word called *super* to invoke the method of parent class and invokes the private instance method in this class directly, the *javac* compiler uses the *invokespecial* to call those methods. The *invokespecial* differs from *invokevirtual* primarily in that *invokespecial* normally selects a method based on the type of the reference rather than the class of the object. In other words, it does static binding instead of dynamic binding.

The *invokespecial* is not used to invoke private class methods, just private instance methods. Private class methods are invoked with *invokestatic*. For example：it must be possible for a subclass to declare an instance method with the same signature as a private instance method in a superclass.

```
class Superclassprivate {
        private void interestingMethod() {
             System.out.println("Super");
        }
        void exampleMethod() {
               interestingMethod();
        }    }
```

```
class Subclassprivate extends Superclassprivate {
        void interestingMethod() {
             System.out.println("sub");
        }
        public static void main(String[] args) {
               Subclassprivate me=new Subclassprivate();
               me.exampleMethod();
        }    }
```

```
Method void exampleMethod()
    0 aload_0
    1 invokespecial #5 <Method void interestingMethod()>
    4 return
```

```
Method void main(java.lang.String[])
    0 new #5 <Class Subclassprivate>
    3 dup
    4 invokespecial #6 <Method Subclassprivate()>
    7 astore_1
    8 aload_1
    9 invokevirtual #7 <Method null>
   12 return
```

```
Superclassprivate_exampleMethod_3       proc
…
.code
        pop     ab_return_addr
        pop     [ab_local_vars+0*4]                 ; this reference
        …
        push     [ab_local_vars+0*4]
        call    Superclassprivate_interestingMethod_2
        push     ab_return_addr
        ret
Superclassprivate_exampleMethod_3        endp
```

Asm code of exampleMethod

```
…
   _main proc
        …
        .if     eax == 0
           call     Superclassprivate_exampleMethod_3
           mov      eax,0
        .endif
        .if     eax == 8
           call     Superclassprivate_exampleMethod_3
           mov      eax,8
        .endif
        …
```

Asm code of main

The Java Virtual Machine invokes the interestingMethod() defined in Superclassprivate even though the object is an instance of class Subclassprivate and there is an accessible interestingMethod() defined in Subclassprivate. So, the asm code of exampleMethod invokes the interestingMethod of Superclassprivate directly.

When the Java Virtual Machine resolves an *invokespecial* instruction's symbolic reference to a superclass method, it dynamically searches the current class's superclasses to find the nearest superclass implementation of the method. For example：

```
class Cat {
    void someMethod() {
    }   }
```

```
class TabbyCat extends Cat {
    void someMethod() {
        super.someMethod();
    }   }
```

```
Compiled from TabbyCat.java
…
Method void someMethod()
   0 aload_0
   1 invokespecial #2 <Method void someMethod()>
   4 return
```

```
TabbyCat_someMethod_2     proc
        …
        call    Cat_someMethod_2
        …
```

invokeinterface

The *invokeinterface* performs the same function as *invokevirtual*, it invokes instance methods and uses dynamic binding. It is related to object. The difference between these two instructions is that *invokevirtual* is used when the type of the reference is a class, whereas *invokeinterface* is used when the type of the reference is an interface. The Java Virtual Machine uses a different instruction to invoke a method on an interface reference because it can not make as many assumptions about the method table offset given an interface reference as it can given a class reference. Given a class reference, a method will always occupy the same position in the method table, independent of the actual class of the object. This is not true given an interface reference. The method could occupy different locations for different classes that implement the same interface.

When the Java Virtual Machine encounters an *invokevirtual* instruction and resolves the symbolic reference to a direct reference to an instance method, that direct reference is likely an offset into a method table. From that point forward, the same offset can be used. For an *invokeinterface* instruction, the JVM will have to search through the method table every single time the instruction is encountered, because it can not assume the offset is the same as the previous time.

The translating rule of the *invokeinterface* is the same as the *invokevirtual*. For calling the system method by *invokeinterface*, every single time the instruction encountered, we always use those associated JNI functions to finish the *invokeinterface* work. For calling the user method by *invokeinterface*, every single time the instruction encountered, we always use the same translating steps which are mentioned in the beginning of the section to call the correct method. Because the translating rule of the *invokeinterface* is the same as the *invokevirtual*, we do not illustrate an example about *invokeinterface* in this section.

<u>return instructions</u>

The return instructions consist of *ireturn*, *lreturn*, *freturn*, *dreturn*, *areturn*, and *return*. If the location of the returning series bytecode instructions is the last bytecode in the bytecode stream of a method, we do not generate any assembly code, and the returning value which was prepared by bytecodes prior to those return series bytecodes immediately were on the native stack. Because we add that pushing return address onto the native stack and *ret* actions in the initialization asm code mentioned in the figure 3.5, we can return back to the caller method. If they are not the last bytecode in the bytecode stream of a method, the returning value was also prepared, but we must generate the following assembly codes. The control can return back to the caller method by the following assembly codes.

```
push    return_addr
ret
```

For example：

```
public class stepoly {
        static int stepPoly(int x)    {
              if(x<0)   {
                    System.out.println("foo");
                    return -1;
              }
              else if(x<=5)
                    return x*x;
              else
                    return x*5+16;
        }
        public static void main(String[] args) {
            int g=stepPoly(-5);
            System.out.println(g);
        }   }
```

```
Method int stepPoly(int)
   …
   12 iconst_m1                  // parameter in the stack
   13 ireturn
   …
   21 imul                       // parameter in the stack
   22 ireturn
   …
   28 iadd                       // parameter in the stack
   29 ireturn
```

```
        …
        push     aa_return_addr
        ret
        …
        push     aa_return_addr
        ret
        …
        push     aa_return_addr
        ret
stepoly_stepPoly_2          endp
```

3.2.5 Template Match

We propose template matching technique to generate more efficient ASM codes. This template matching method is original from the paper[7], but we add some concepts of peephole pattern table[11] to those template patterns. We arrange many bytecode instructions to form some patterns. Before we want to begin to translate every bytecode instruction, we compare these patterns with adjacent bytecode instructions. If they are matched, the result indicates that these adjacent bytecode instructions can be translated to ASM codes together. We extend the concept of peephole pattern table[11] to constitute advanced patterns which let our translator to generate less native codes easily. We can implement some optimizations[2] in the template patterns. For examples：

```
bipush 9        ──►   mov   [aa_local_vars+0*4], 9
istore_0
```

```
iload_1               mov   eax, [aa_local_vars+1*4]
iload 8         ──►   mov   ebx, [aa_local_vars+8*4]
if_icmpeq 38          cmp   eax, ebx
                      je    aa_38
```

```
iload 6               mov   eax, [aa_local_vars+6*4]
iload_3               mov   ebx, [aa_local_vars+3*4]
iconst_1        ──►   add   ebx, 1
iadd                  cmp   eax, ebx
if_icmpne 38          jne   aa_38
```

We can combine two or more than two existent templates to match patterns of Java bytecode instructions. We will implement more optimizations, such as dead code elimination, and common subexpression elimination in the future. According to the experiment result, when the template matching occurs within the loop, the generated assembly codes has outstanding improvement. We hope to make better performance which is close to the JIT.

Chapter 4 Performance Evaluation and Analysis

In this chapter, we carry out a series of performance experiments in order to test the viability of our translator. Test results are first enumerated for comparisons, and later we analyze and explain the reason of causing the performance difference.

4.1 Performance Evaluation and Comparison

Handwritten several small Java programs which consist of loop test, arithmetic tests, array test, field test, and user method tests. The results of these tests are in table 4.1~4.5. Many excellent Java benchmarks exist. The Linpack executes complex algorithms which perform many computations. The BTest is for a statistical evaluation, and does a lot of shuffling around in an integer array and double computations. The results for the two excellent benchmarks are in table 4.6.

The experiment platform is a AMD Athlon 1600 runs at 1.41 GHz with 256MB RAM. The operating system is Microsoft Windows XP professional edition. Java Virtual Machine is Sun "Java.exe" in Sun J2SDK 1.4.1_01 for win32. We run these benchmarks in three approaches — interpreter, JIT, and our translator. The results of the three approaches show their execution time, and the measure unit is second.

1. loop test

The loop test iterated an empty loop 1000000000 times. The result is list in table 4.1.

Benchmark	Interpreter	JIT	Asm code
Empty loop iterated 1000000000 times	34.8	3.45	5.81

Table 4.1 Loop test

2. arithmetic tests

Four tests are enclosed in this suite. Each item is executed under 100000000 times, as shown in table 4.2.

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 tests

3. array test

A three-dimension array with 60 elements is created. Each element is assigned an integral value. The assignment is executed 1000 times repeatedly. The result is shown in table 4.3.

Benchmark	Interpreter	JIT	Asm code
One-dimension int array with 10000000 elements creation	0.09	0.09	0.09
3-dimension int array	12.28	12.06	12.48
One-dimension int array with 3 elements is created for 1000000 times	0.38	0.06	1.5

Table 4.3 Array test

4. field test

An instance which has an integral instance field and an integral class field is created. We then use a loop to access that field for 1000000 times. Table 4.4 shows the result.

Benchmark	Interpreter	JIT	Asm code
Class field accessed for 1000000 times	0.09	0.02	1.58
Instance field accessed for 1000000 times	0.09	0.02	1.61

Table 4.4 Field test

5. user method tests

An empty user instance method and an empty user static method are called 1000000 times. Table 4.5 shows the results.

Benchmark	Interpreter	JIT	Asm code
Static method invoked for 1000000 times	0.08	0.02	0.03
Instance method invoked for 1000000 times	0.08	0.02	0.08

Table 4.5 User method tests

6. excellent examples tests

Linpack and BTest are tested, as shown in table 4.6.

Benchmark	Interpreter	JIT	Asm code
Linpack	0.08	0.02	0.14
BTest	77	8	76

Table 4.6 Excellent examples tests

4.2 Benchmarks Analysis

After carefully examining codes of these benchmark programs in detail, we analysis the six clusters, and propose to some viewpoints.

1. the loop test

Take a look at the result, we notice that the performance of our translated asm codes is close to that of the JIT. This is because the interpreter uses the memory to process the loop, and does not use the registers. The JIT takes advantage of the registers to improve the performance of the loop. Our translator uses the memory to record the iteration number, and uses the registers to do these comparison and increment. The jump action of the asm code is faster than the jump bytecode instruction of the interpreter. The performance difference of our translator and the JIT is that the JIT uses the registers efficiently and does loop optimization.

2. the arithmetic tests

The performance of our translator in integer computation and long addition and long subtraction are good as expected, and if we make more template matching in the future, the performance will be close to the JIT. This is because addition and subtraction operations on multiword operands are straightforward, so we use several Intel assembly instructions to finish those computations directly, and the code size is short. But the performance of long multiplication and division is not good as expected. The possible reason is that multiplication and division of multiword operands are not straightforward, so we must use many Intel assembly instructions and computation algorithms to finish those operations, and these used algorithms called longhand multiplication algorithm and nonrestoring division algorithm[26,32] are slower than the internal operations of the interpreter.

We use the assembly instructions of the FPU to finish the float and double computations. The performance of our translator is better than the interpreter, but slower than the JIT. This is because local variables are saved in the memory, we must get the value of the local variable through memory, and then continue to compute. Therefore, the slow reason is the reading and writing of the memory.

3. the array test, field test

Our translator uses the JNI function to translate those bytecode instructions associated the array and field operations. When we use the JNI function once, the difference is very small and is unobvious. If we use the JNI function in a heavy-weight loop, the difference is obvious. The solution is that many results which many JNI functions generate are unchanged in a heavy-weight loop, and we can let these JNI functions to execute once, and save those results which these JNI functions generate for next used. This work that how to use the JNI function efficiently is the future work.

4. the user method tests

For static method calling, the performance of our translator is close to the JIT. This is because the actual callee method is decided at compiling time, and thus we can generate a calling assembly code directly. The speed of a calling assembly code is fast. For instance method calling, the speed of invoking once is faster than the interpreter. The interpreter resolves the instance method when the interpreter first meets the instance method at running time, and replaces the entry of the constant pool with direct address which may be an offset in the method table. When the interpreter meets the same instance method second, it jump to the starting address of the instance method to execute. The translated asm code always performs our designed dynamic dispatching algorithm to call the correct method every time, even though is the same instance method. Because our translated asm codes must perform dynamic dispatching despite the same instance method, but the interpreter invokes the same instance method by direct address except for first resolving. So this is why our translator has same performance as the interpreter for calling instance method in a heavy-weight loop. The solution is that smarter dynamic dispatching mechanism. If invoking the same instance method second, we do not perform the dynamic dispatching, and can invoke the instance method directly. The solution needs to analysis the data flow of the program, and this is a future work.

5. the excellent examples tests

The performance of our translator is near to the interpreter, and there are some distance between our translator and the JIT. The Linpack and BTest has a lot of shuffling around in array computations. The possible slower reason is the overhead of using the JNI functions. If we can use the JNI function efficiently, we can improve the performance.

In order to make the translated asm codes right, we do many tests by many examples during the translator developed phase. Basically, we feel that our translator is potential to deliver good performance on these examples during testing course. In the real world, the Java application programmer will not write a program which invokes a same instance method for many times, and has too many array computations. Therefore, to date our translator can gets the good performance for those general applications.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

Performance of Java is always an interesting issue. In this thesis, we described an approach to translate Java bytecode to X86 assembly code. The approach does not generates the intermediate representation, but adopts code mapping method for the bytecode instruction. The code mapping method simplifies the complexity of other native compilers which generate IR in the past.

In order to generate right assembly codes, we propose that the one-pass is used to gather informations about the class file and bytecode instructions, and the two-pass is used to generate the actual asm codes. We use the template matching to do some optimizations, and use the JNI function to communicate with the Java Virtual Machine.

In order to implement the dynamic dispatching in assembly codes, we designed a mechanism which consists of four components mentioned in section 3.2.4.2. We use the four components to finish that the environment which is a static binding can invoke the method which are decided in a dynamic binding environment. To date, the speed of our translator is faster than the interpreter, but is slower than the JIT according to our experiments. Although there are still some distance between our translator and the JIT, but the performance of our translator which is used on real-world applications is acceptable, and there are opportunity to refine it.

5.2 Future work

To date, there are still many enhancing the performance methods of Java in the native compiler category. They are differentiated between machine dependent and machine independent. In order to enhance the performance, we list here some work we wish to accomplish in the future.

- smarter register allocation.
- implement more optimizations in the template matching.
- Make more tests, such as SPEC JVM98 microbenchmark.
- Use the JNI function efficiently.
- Smarter dynamic dispatching.

We believe that our translator will has the same performance as the JIT after finishing these future work, even better.

References

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth, "Fast and effective code generation in a just-in-time Java compiler", ACM SIGPLAN Notices, 33(5):280-290,May 1998.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers Principles, Techniques and Tools", Addison-Wesley, 1986.

[3] J. Gosling, B. Joy and G. Steele, "The Java Language Specification, second edition", Addison-Wesley, 2000.

[4] Arnold, Ken, and James Gosling, "The Java Programming Language", (Reading, MA: Addison-Wesley), July 1997.

[5] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification", Addison-Wesley, Reading, MA, USA, second edition, 1999.

[6] C.-H.A. Hsieh, J.C. Gyllenhaal, and W.W. Hwu, "Java Bytecode to Native Code Translation:The Caffeine Prototype and Preliminary Results", Proc. 29th Ann. International Symp. Microarchitecture, IEEE CS Press, Los Alamitos, Calif., 1996, pp.90-97.

[7] Ye Hua, Tong WeiQin, Yao WenSheng, "Platform independence issues in compiling Java bytecode to native code", 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.

[8] C.-H.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, W.-M.W. Hwu, "Compilers for Improved Java Performance", Computer ,Volume: 30 ,Issue: 6 ,June 1997 Pages:67-75.

[9] P. Bothner, "A Gcc-based Java Implementation", Compcon '97. Proceedings, IEEE ,23-26 Feb. 1997 Pages:174-178.

[10] D. Lance, R.H. Untch, N.J. Wahl, "Bytecode-based Java Program Analysis", April 1999 Proceedings of the 37th annual Southeast regional conference.

[11] H.D. Lambright, "Java Bytecode Optimizations" , Compcon '97. Proceedings, IEEE ,23-26 Feb. 1997 Pages:206-210.

[12] R. V.-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, "Soot-a Java Bytecode Optimization Framework" , November 1999 Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research.

[13] F.G. Chen, and Ting-Wei Hou, "Design and Implementation of a Java Execution Environment" , Parallel and Distributed Systems, 1998. Proceedings., 1998 International Conference on ,14-16 Dec. 1998 Pages:686-692.

[14] A. Azevedo, A. Nicolau, and J. Hummel, "Java Annotation-Aware Just-In-Time (AJIT) Compilation System" , June 1999 Proceedings of the ACM 1999 conference on Java Grande.

[15] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, A. Yeryomin, "Overview of Excelsior JET, a high Performance Alternative to Java Virtual Machines" , July 2002 Proceedings of the third international workshop on Software and performance.

[16] Suganuma et al., "Overview of the IBM Java Just-In-Time Compiler" , IBM Systems Journal, Vol. 39, No. 1, 2000.

[17] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson, "Toba: Java for applications: A way ahead of time (WAT) compiler" , Proc. Usenix Association, pp.41-54, 1997.

[18] M. Cierniak, G.-Yuan Lueh, J.M. Stichnoth, "Practicing JUDO:Java Under Dynamic Optimizations" , May 2000 ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Volume 35 Issue 5.

[19] "Tower J" , Tower Technology., http://www.twr.com/.

[20] Java Technology, http://java.sun.com/.

[21] Bill Venners, Inside the Java 2 Virtual Machine, McGraw-Hill Companies, 2001.

[22] Jon Meyer and Troy Downing 著，JAVA 虛擬機器，蔡寶進譯，美商歐萊禮股份有限公司台灣分公司，2000 年 7 月初版。

[23] Joshua Engel, Programming for the Java Virtual Machine, Addison Wesley Longman, 1999.

[24] Sheng Liang, The Java Native Interface, Addison Wesley, Longman, 1999.

[25] Kip R. Irvine, Assembly Language for Intel-Based Computers, Fourth Edition, Pearson Education, Inc., 2003.

[26] Sivarama P. Dandamudi, Introduction to Assembly Language Programming, Springer, 1998.

[27] Ytha Yu, Charles Marut, Assembly Language Programming and Organization of the IBM PC, McGraw-Hill, inc., 1992.

[28] Richard C. Detmer, Introduction to 80x86 Assembly Language and Computer Architecture, Jones and Bartlett Publishers, 2001.

[29] IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 2002.

[30] IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference, 2002.

[31] "99% Java", Assembly and JNI, http://www.amherst.edu/~tliron/jni/assembly.html, Biswajit Sarkar.

[32] David A. Patterson, John L. Hennessy, Computer Organization & Design The Hardware/Software Interface, second edition, Morgan Kaufmann Publishers, 1997.

[33] 探矽工作室著，深入嵌入式 Java 虛擬機器，學貫行銷股份有限公司，2002 年 8 月初版。

Appendix

The following code is the startvm.c.

```
#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_01\\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;
}
```