Design and Implementation of a Java Bytecode Recompiler for ARM RISC Processors

Mirko Raner
(raner@mathema.de)

11th September 1998

Abstract

The Java programming language and the underlying Java Virtual Machine (JVM) are amongst the most influential developments in computer science in the past three years. A functioning Java runtime system has become an essential feature of many common applications. Companies that are interested in Java can choose between licensing an existing JVM implementation or developing an implementation of their own. The latter solution becomes increasingly attractive due to the high license fees.
There are several different approaches for an implementation of the JVM; currently interpreter-based solutions are the most common approach. This thesis deals with the implementation of the JVM on basis of a recompiler which translates Java bytecode to native machine code. The target architecture for the recompiler is the ARM RISC processor family (by Advanced RISC Machines) running the operating system RISC OS or its derivative NC OS.
Most issues which arise in conjunction with the translation of Java bytecode to native machine code will be discussed. The thesis will also show a possible implementation approach for such a recompiler.

Many thanks to MATHEMA Software GmbH for supporting this work.
(Internet: http://www.mathema.de)

Contents

List of Tables

List of Figures

1 Introduction

The ability to execute Java applications and applets has become very important for operating systems and browser environments. The core of a Java execution enviroment is the Java Virtual Machine (JVM), an abstract processor which executes Java intermediate code (so-called bytecode).
Thus the JVM is the basic prerequisite for the execution of Java programs. Java is not viable without a complete implementation of the JVM. However, the Java Virtual Machine Specification [LY97] does not specify how the JVM has to be implemented. This means, that a large variety of implementation approaches is conceivable. The most common approach are still interpreter-based solutions, in which the interpreter completely or partially emulates the JVM.

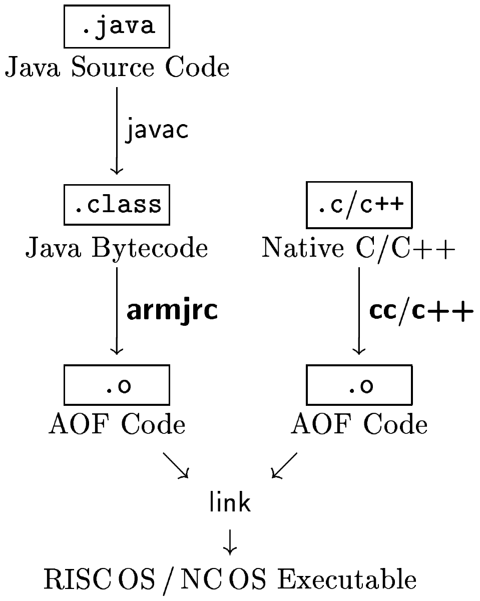
The topic of this thesis is the implementation of the Java Virtual Machine by means of a recompiler which translates the Java bytecode to the native machine code of the target architecture. Of course, this approach is not entirely new. However, it has not been discussed in great detail yet though some similar approaches exist (eg [HGH96]).

The target system for the recompiler is the ARM RISC Processor family [Jag96] by Advanced RISC Machines. Available operating systems for these processors include RISC OS and NC OS, which were also installed on the machines that were used for development (for a detailed description of the operating environment see appendix A).

1.1 Objectives and Requirements

The main objective of this thesis was to design and implement the prototype of a compiler (named armjrc – ARM Java Recompiler) which takes Java class files and translates the contained code to native ARM machine code. The output of the compiler is a file in the ARM Object Format (AOF), which is suitable for linking with standard AOF linkers (eg Acorn's tool "link" [ADT94]):

Figure 1: Interaction of armjrc with other tools



The initial prototype of the armjrc compiler was intended to meet the following requirements:

- The compiler should correctly translate all opcodes which can be directly realized as a sequence of ARM instructions (stack operations, local variable operations, arithmetical and logical operations, comparison and conditional branching, branch tables, casting, object creation, object and array access, method invocation)
- The stack and the local variables of the virtual machine should be efficiently mapped to the processor's registers, ie stack operations in the JVM should *not* simply be mapped to stack operations on the ARM
- Virtual method invocation should be supported, this can be implemented similar to virtual C++ methods
- There should be support for native method integration; the output of the recompiler must be linkable with the AOF output of C and C++ compilers
- There should be a basic native library (implemented in C or C++) which provides important Java classes for I/O and elementary system functions
- A runtime library should provide dummy (!) implementations of the more complex opcodes (eg synchronization and exception handling)
- The functions for memory allocation – but not necessarily for deallocation – should also be included in the basic runtime library
- All essential information from the input class file is preserved in the resulting AOF file, even if this information will only make sense in conjuction with a complete runtime system (eg exception tables)
- The compiler itself should be completely written in Java and should finally be able to translate itself, thus allowing bootstrapping to the target architecture

1.2 Limitations and Subsequent Projects

The approach presented is this thesis will not yield a complete implementation of the JVM. Some of Java's dynamic features cannot be realized by a compiler and need the help of a runtime system (eg garbage collection, thread scheduling, dynamic class loading).
The armjrc compiler will allow to translate a set of Java class files to `.o` files which can then be linked to a monolithic executable. This will provide the same amount of dynamics and flexibility as can be realized with C++.

In order to make the JVM implementation complete the runtime system must provide support for the following features:

- Dynamic class loading at runtime
- Multithreading and synchronization
- Automatic garbage collection
- Exception handling
- Reflection (as introduced in JDK 1.1; see [Fla97], §12)

All these things cannot be handled at compile-time and will possibly be the topic of a further thesis. This thesis could be based on the work presented here and could deal with all issues concerning the runtime system.

1.3 Demarcation against other Approaches

There is a large number of possible implementation approaches for the Java Virtual Machine. It is very important that the approach of armjrc does not get confused with other similar approaches. Basically two things must be taken into account: (1) the time of compilation and (2) the type of output which is generated.

The time of compilation distinguishes recompilers from so-called Just-In-Time compilers (JITC). A JIT compiler is usually coupled with a (rudimentary) Java interpreter and translates short bytecode sequences, single methods or even complete classes just before they are used for the first time. The JIT compiler may also exploit profiling information which has been conducted by the interpreter. A JITC speeds up execution considerably and provides Java's full dynamics through its interpreter component. However, the implementation of a JITC is very hard and expensive and the compiler code itself has to be stored in memory during runtime (which makes it inapplicable for certain environments; see [Bot97]).
In contrast to a JITC a recompiler[*)] will translate *all* the classes *before* execution. The recompiler (together with a suitable linker) will produce a single monolithic executable which can be run as a stand-alone application. As pointed out in section 1.2 some special mechanisms of Java can only be enabled with the support of a runtime system.
armjrc is clearly a pure recompiler, which may however be completed by a runtime environment in a subsequent project.

Even amongst pure recompilers there are still some significant differences. Not all of the recompilers generate native machine code for the target platform. There are some approaches which translate the incoming bytecode to C source code which is then compiled by a regular C compiler. The pipeline from a Java source code to the running application gets very long in these approaches (Java source → class file → C code → object file → executable) and involves a large number of tools (javac – recompiler – C compiler – optimizer – linker). However, C compilers often produce optimized code of good quality.
armjrc is a recompiler which produces machine code without any intermediate processing stages, the bytecode is directly translated to ARM code.

Table 1 shows an overview of current Java bytecode (re-)compiler products:

Table 1: Java Bytecode Recompiler Products

JIT Compilers	Native Recompiler	C Code Recompilers
Symantec JITspeed	IMPACT Caffeine (Intel; [HGH96])	Toba [PTB$^+$97]
Inprise AppAccelerator	JA2S (SPARC; [JRC97])	Harissa [MMBC97]
HP JIT for MPE/iX	armjrc (ARM)	J2C [And96]

[*)] synonyms are: native compiler, monolithic compiler, one-off compiler, off-line compiler or way-ahead-of-time compiler (WATC)

1.4 Outline of the Thesis

Section 2 gives an overview of the source architecture (JVM) and the target architecture (ARM). Architectural particularities which are relevant for the compilation process are discussed as well. The subsection on the JVM also discusses how some parts of the Java runtime system could be realized. The section concludes with a comparison of both architectures and an evaluation of the ARM as a target architecture for Java.

Section 3 introduces the translation approach which was used for armjrc. This section puts its focus on the actual translation process and does not deal with any pre- or post-processing.

Section 4 analyzes the steps of compilation, beginning with the parsing of the class file and ending with the generation of the AOF file. The translation process discussed in section 3 is only one part (though undoubtedly the key part) of the overall compilation process – which also includes stages like pre-processing and register mapping.

Section 5 presents a classification of the JVM instruction set. This section contains all the translation tables that are used by armjrc. Special problems that arise with certain instructions (or instruction groups) are discussed as well.

Section 6 briefly discusses the remaining problems which are not solved by armjrc. This section also contains a rough estimation of armjrc's performance in comparison to other recompilers and also shows some possibilities of code optimization.

Section 7 draws a final conclusion and evaluates to what degree the original objectives have been met.

2 Architecture Overview

In order to write a compiler which translates code from one architecture to another it is indispensable to have a detailed knowledge of both involved architectures.
This section gives a detailed overview of the source and the target architecture and discusses all issues which are relevant for the compilation process. Subsection 2.1 also contains some implementation hints for the components of the runtime system.

2.1 The Java Virtual Machine (JVM)

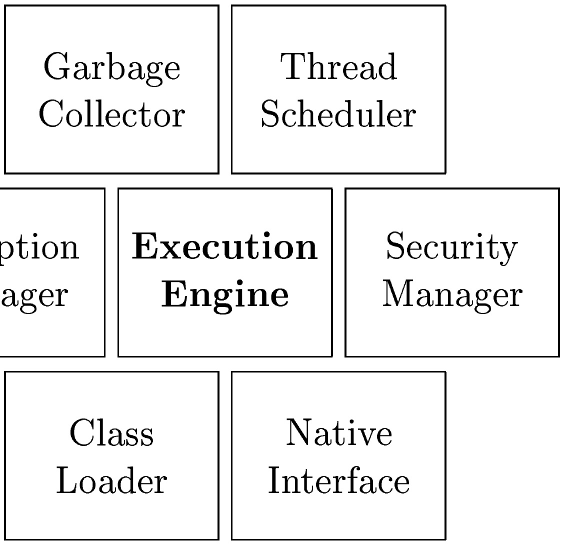
The Java Virtual Machine (JVM) is a virtual processor system which is the foundation for Sun's platform-independent Java programming language. Java source code is not compiled for a particular target architecture but is translated to so-called bytecode which is executed by the JVM [GJS96].
The JVM is a virtual system which can be implemented in many ways: by software emulation (interpreter), by recompiling to native code and of course by real processor chips which implement the JVM in hardware (eg the PSC 1000 "ShBoom Box" by Patriot Scientific [Can98]).
In §3 of the Java Virtual Machine Specification [LY97] only the following demand is postulated:

> *" To implement the Java Virtual Machine correctly, you need only be able to read the Java `class` file format and correctly perform the operations specified therein. "*

However, the JVM is not only a processor model (ie the description of an execution engine) but also comprises a complete runtime environment. Just as with real processors the JVM needs an environment which interfaces it with the surrounding hardware and software components.

Figure 2 shows the components of a Java runtime environment.

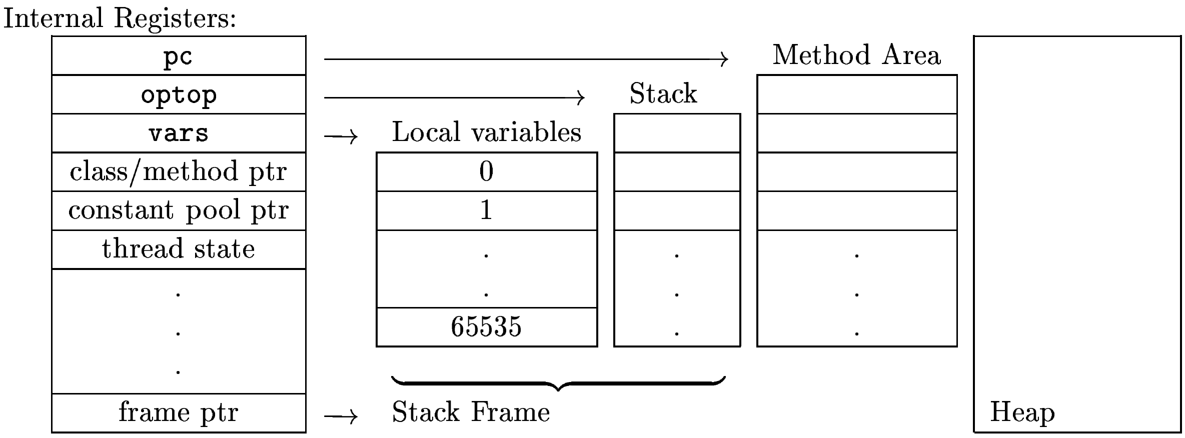
Figure 2: JVM Runtime System



2.1.1 The Execution Engine

The execution engine is the actual processing unit of the JVM and therefore the central part of the runtime system. As pointed out above, the implementation and the internal structure of the JVM is not specified. However, if the JVM execution engine was considered as a "real" microprocessor its structure would be similar to figure 3 (see [Dal97]). armjrc uses the ARM processor as execution engine, therefore it does not need to bother with the emulation of the virtual processor. Instead the emulation is done by translating the code.

Figure 3: JVM Internal Register Set



The register set on the left side of the diagram contains the internal registers of the JVM. The register set must not necessarily be exactly as in the diagram – the JVM specification makes no statement on this – but most hardware approaches implement a register set which is very similar to the diagram (eg the "ShBoom Box" [Can98]). The internal registers are not visible to the programmer and there are no instructions to access these registers directly.

The `pc` register is the program counter which points to the currently processed instruction. The execution engine of the JVM is a stack-based zero-operand architecture. That is, all essential operations are performed on a stack and not directly within registers. The `optop` register is used to manage this internal stack, it always points to the top stack element. The actual registers from the programmers view are called "local variables" in the JVM. The register `vars` points to the memory area (or to an area in the processor's register file) where the current local variables are stored. The JVM can address up to 65536 local variables which are all 32 bits wide. Large data types such as IEEE 754 double precision values or long integers need two adjacent local variables. Each stack frame on the caller stack contains one fixed-size area for the operand stack and one for the local variables.

The instruction set is explained in more detail in section 5, a complete instruction set overview can be found in [MD97], [LY97] or [Dal97].

2.1.2 The Garbage Collector

The Java runtime environment depends on a garbage collector, ie garbage collection (GC; see [JL96]) is absolutely essential for Java and Java cannot operate without it!

Neither the Java Programming Language nor the JVM itself support instructions for explicitly releasing memory. Memory can be allocated with JVM instructions like `new` or `anewarray` but deallocation is the responsibility of the garbage collector.
A really complete implementation of a Java runtime environment must therefore also comprise a fully operative garbage collector which automatically frees unused memory blocks.

However, garbage collection is a task of the runtime system and not of the compiler. armjrc translates memory allocation instructions (like `new` or `newarray`) into calls to a runtime library. The preliminary implementation of this library simply maps these calls to regular memory allocation calls. A later version of the runtime library can interface the runtime system with a complete garbage-collecting memory manager.

Applications which excessively allocate new objects will quickly flood the memory of the machine when the preliminary runtime environment is used.

2.1.3 The Exception Manager

Just as with garbage collection, exceptions are an integral part of Java as well. Exception objects and an exception handling mechanism are offered by the Java programming language and also appear on the level of the JVM. An exception manager is therfore an essential component of every Java runtime environment.

There are explicit and implicit exceptions. Explicit exceptions are generated with Java's `throw` statement or – on JVM level – with the `athrow` instruction. However, exceptions can also be generated implicitly. In Java source code a statement like `int x = y/0;` will produce an `ArithmeticException` even if there is no explicit `throw` statement. On JVM level several instructions can implicitly throw exceptions (eg `checkcast`, `idiv`, `invokevirtual`, `newarray`).

Surprisingly, there is no correspondent for the well-known Java structure `try ... catch ... finally` in the instruction set of the JVM. The begin and end of an observed block is not marked by special instructions. Instead, the compiler generates an exception table which contains the start and end address of the observed code block, the address of the exception handler and the exception type.
Only if an exception was actually thrown, the exception tables are searched for an appropriate handler. This method has the advantage that an exception handler does not impose any overhead. A code block which is guarded by one or more exception handlers executes exactly as fast as a block without any exception handlers.
The exception manager has to administer the exception tables, search for the appropriate exception handlers and propagate unhandled exceptions correctly to the calling method.

The `athrow` instruction can be translated to a library call on the ARM. All explicit exceptions can be handled that way.
The code for JVM instructions which throw implicit exceptions must check for the exception condition and throw an exception explicitly, if necessary.
Some implicit exceptions can also be handled by the ARM processor. For example, a memory access through a `null` pointer will produce a data abort exception. This can also be handled by overwriting the corresponding exception vector for the ARM (see [Jag96], §2.5).

In order to provide useful stack trace information the line number tables of the original class file must be somehow preserved in the output file.

2.1.4 The Class Loader

The class loader is the part of the Java runtime system which fetches and prepares class files from sources like the local disc drive or the network. The important mechanism of dynamic class loading is provided by the class loader.

As the first version of armjrc is a completely static solution which does not implement any of the dynamic features of Java, there is no need for a class loader. However, if a complete runtime system is build around armjrc there must be a class loader which performs the following actions:

- Loading of requested class files
- Compiling them into native code
- Interfacing the resulting `.o` with the already installed `.o` files

That is, the class loader acts as a compiler driver for armjrc and additionally replaces the functionality of link. The class loader must be able to integrate new `.o` files with an already running JVM environment.

However, all these issues are only mentioned for sake of completeness; their actual implementation lies beyond the scope of this thesis.

2.1.5 The Security Manager

Security is an important aspect of the Java application environment. The JVM must protect the underlying system against deletion, damage, corruption or disclosure of confidential data.
Security is provided by two concepts:

- Checking of `.class` files by the *bytecode verifier*
- Access control for critical resources by the `SecurityManager` class

armjrc does not have a bytecode verifier of its own. Generally the tool relies on the fact that all input files have been generated by a valid compiler and do not contain any illegal instruction sequences. However, it would be possible to trigger an external verifier program.

The class `java.lang.SecurityManager` is an abstract class which is usually implemented by a native subclass. This class does not concern the design of the recompiler or the runtime library. The implementation only depends on the JDK library for armjrc and can be exchanged without affecting the compiler or the runtime environment.

2.1.6 The Thread Scheduler

From the beginning, the JVM was designed to be a multi-threaded system and support for multithreading is an integral feature of the JVM.
The JVM runtime environment has to take care of the management of several threads, the distribution of CPU time between them and the synchronization of mutually shared objects.

Only object synchronization is reflected in the instruction set of the JVM. For achieving mutual exclusion the JVM provides the instructions `monitorenter` and `monitorexit`. These instructions can be translated to library calls which make use of the ARM's `SWP` instruction (see [Jag96], §§3.12, 4.5).

However, these two instructions are the only thread-related instructions in the instruction set of the JVM. The rest of the thread API is provided by native methods of `java.lang.Object` or `java.lang.Thread` (eg `start()`, `stop()`, `yield()`, `wait()` or `notify()`). These methods have to be implemented in the JDK class library and the recompiler does not have to deal with their translation.

2.1.7 The Native Interface

There must be a defined way of interaction between the JVM and its surrounding environment – the so-called native interface. A lot of problems cannot be handled by JVM itself and need `native` support code. When using a recompiler, all code in the system is actually native code, because the complete Java bytecode has already been translated to machine code. However, there must be an interface which allows "handwritten" code to be integrated with the output of the recompiler.

The Java Native Interface (JNI) [Gor98] is a newly established, platform-independent standard for interfacing Java applications with native code written in C or C++. However, a complete implementation of JNI is rather voluminous and therefore armjrc uses a different approach for the integration of native code. The output of armjrc adheres to the same conventions as the output of the Acorn C/C++ compiler [ACC94]. That is, the resulting AOF file has the same structure and uses the same symbol naming scheme as the C/C++ compiler. Therefore intergration with native code can simply be achieved by linking the output of the recompiler with the native .o files which have been generated by the C/C++ compiler (see also figure 1).

2.2 The Advanced RISC Machine (ARM)

The ARM [Jag96] is a typical 32-bit RISC (Reduced Instruction Set Computer) processor. It has a relatively small number of instructions and registers. The ARM is a load/store architecture, which means that memory accesses are only possible for some special instructions whilst the majority of the instructions solely operates on registers. All instructions occupy 4 bytes of memory (ie 1 word), most of the data processing instructions have two source registers (*Rn*, *Rm*) and a destination register (*Rd*) encoded. Instead of source register *Rn* also a 12-bit constant can be used. The ARM also has some particularities which are important for producing efficient ARM machine code (see section 2.2.3).

2.2.1 Register Set

ARM processors have between 31 (architecture version 2) and 37 (architecture version 4) registers, which also includes internal registers that are not visible to the programmer. There are two general purpose register banks with 15 user mode registers (the main bank) and 15 shadow registers for the privileged processing mode. Both register banks share a common PC (program counter) register. Only 16 registers are visible at once, no matter in what processing mode the processor currently is. The programmers view of the ARM registers is shown in figure 4:

Figure 4: ARM Register Set

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 / SP
R14 / LR
R15 / PC

F0
F1
F2
F3
F4
F5
F6
F7

96-bit FP Registers ↑

Register R15 serves as the program counter (PC). It can be accessed like any other register, ie a branch to a certain address can be achieved by writing the appropriate value to R15. R14 is the so-called link register (LR). Branch instructions (BL) can have their return address automatically stored in this register. Simple subroutines can return to the caller by moving the contents of the link register R14 to the program counter R15 (return from leaf procedures is encoded as `MOV PC,LR`; there is no special return instruction). Register R13 is usually used as stack pointer (SP). However, this is not dictated by the ARM architecture but a mere convention. Any other register could be used as stack pointer as well. In this document SP always refers to R13.

2.2.2 Instruction Set

The instruction set of the ARM itself is relatively small. It is divided into 7 functional groups (data processing: 16, multiply: 2, branch: 2, load/store: 8, semaphore: 2, status register access: 2, coprocessor: 5) with a total number of 37 instructions [Jag96]. The armjrc recompiler also makes use of the ARM FPA (floating-point accelerator) instructions [ARM96]. The ARM 7500FE has an integrated FPA 11 floating-point unit, ARM 3 processors can be fitted with an FPA 10 coprocessor or can use the floating-point emulator software (FPE). armjrc only uses the instruction subset shown in tables 2 and 3.

Table 2: ARM Instruction Set

MOV	Rd,Rm	Assigment: $Rd = Rm$
MVN	Rd,Rm	Negated assignment: $Rd = \tilde{}Rm$
ADD	Rd,Rm,Rn	Addition: $Rd = Rm + Rn$
ADC	Rd,Rm,Rn	Addition with carry: $Rd = Rm + Rn + C$
SUB	Rd,Rm,Rn	Subtraction: $Rd = Rm - Rn$
SBC	Rd,Rm,Rn	Subtraction with carry: $Rd = Rm - Rn - \tilde{}C$
RSB	Rd,Rm,Rn	Reverse subtraction: $Rd = Rn - Rm$
RSC	Rd,Rm,Rn	Reverse subtraction with carry: $Rd = Rn - Rm - \tilde{}C$
AND	Rd,Rm,Rn	Bitwise conjunction (AND): $Rd = Rm \,\&\, Rn$
ORR	Rd,Rm,Rn	Bitwise disjunction (OR): $Rd = Rm \mid Rn$
EOR	Rd,Rm,Rn	Bitwise antivalence (XOR): $Rd = Rm \,\hat{}\, Rn$
BIC	Rd,Rm,Rn	Bit clear: $Rd = Rm \,\&\, \tilde{}Rn$
CMP	Rm,Rn	Arithmetic comparison: $Rm == Rn$ (→ SUB)
CMN	Rm,Rn	Negated arithmetic comparison: $Rm == \tilde{}Rn$
TEQ	Rm,Rn	Equivalence: $Rm \,\hat{}\, Rn == 0$ (→ EOR)
TST	Rm,Rn	Bit test: $Rm \,\&\, Rn == 0$ (→ AND)
MUL	Rd,Rm,Rn	Multiplication: $Rd = Rm * Rn$
B	offset	Branch
BL	offset	Branch with link (LR = return address)
LDR	Rd,<Rm,Rn>	Single load (address mode dependent)
STR	Rd,<Rm,Rn>	Single store (address mode dependent)
LDM	Rd,register	Multiple load (register list)
STM	Rd,register	Multiple store (register list)
SWP	Rd,Rn,[Rm]	Swap (semaphore access)

Table 3: FPA Instruction Set

MVF	Fd,Fm	Assignment: $Fd = Fm$
MNF	Fd,Fm	Negative assignment: $Fd = -Fm$
ADF	Fd,Fm,Fn	Addition: $Fd = Fm + Fn$
SUF	Fd,Fm,Fn	Subtraction: $Fd = Fm - Fn$
RSF	Fd,Fm,Fn	Reverse subtraction: $Fd = Fn - Fm$
MUF	Fd,Fm,Fn	Multiplication: $Fd = Fm * Fn$
DVF	Fd,Fm,Fn	Division: $Fd = Fm/Fn$
RDF	Fd,Fm,Fn	Reverse division: $Fd = Fn/Fm$
FML	Fd,Fm,Fn	Fast multiplication (single precision): $Fd = Fm * Fn$
FDV	Fd,Fm,Fn	Fast division (single precision): $Fd = Fm/Fn$
FRD	Fd,Fm,Fn	Fast reverse division (single precision): $Fd = Fn/Fm$
RMF	Fd,Fm,Fn	Remainder $Fd = Fn\%Fm$
CMF	Fm,Fn	Comparison: $Fm == Fn$
CNF	Fm,Fn	Negative comparison: $Fm == -Fn$
FLT	Fm,Rd	Conversion: $Fm = Rd$
FIX	Rd,Fm	Conversion: $Rd = Fm$
LDF	Fd,<Rm>	Single FP load (address mode dependent)
STF	Rd,<Rm>	Single FP store (address mode dependent)
LFM	Rd,count,[Rm]	Multiple FP load
SFM	Rd,count,[Rm]	Multiple FP store

2.2.3 Special Features

The ARM has some special features which must be taken into account for code generation. These particularities are:

- All instructions (and not only branch instructions!) can be conditionally executed.
  In contrast to many other processors which only have conditional *branches*, the ARM allows *every instruction* to have a condition associated with it. There are 16 different conditions. Usually all instructions implicitly have the condition AL (always), which means that they will be executed independent from the current status flags. If necessary, an instruction can be given a different condition, such as EQ (equal), NE (not equal), MI (minus/negative), GT (greater than) or LT (less than).
  Many code sequences which need a conditional branch on other architectures can be coded more efficiently on the ARM. Euclid's algorithm, for example, can be implemented with only 5 instructions on the ARM (see [Jag96], §4.2.2)[*)]:

```
gcd     CMP     R0,R1
        SUBGT   R0,R0,R1
        SUBLT   R1,R1,R0
        BNE     gcd
        MOV     PC,LR
```

[*)] It must be noted that a regular SUB leaves the status flags unaffected. In order to update the status flags the S-bit of the instruction must be set (eg SUBLTS)

- Shift operations can be executed in parallel to other operations.
  The ARM has an integrated barrel shifter [Gin91] which operates in parallel to the rest of the integer unit. All data-processing and load/store instructions can additionally perform one out of five different shift operations (`LSL` – logical shift left, `LSR` – logical shift right, `ASR` – arithmetic shift right, `ROR` – rotate right or `RRX` – 1-bit rotate right with extend).
  For example, the Java statement `r1 = r2 - r3 / 4` can be efficiently encoded into the single ARM instruction `SUB R1,R2,R3,ASR #2`. The additional shift can also be used for fast multiplication with certain constants (eg the code `ADD R0,R0,R0,LSL #4` multiplies `R0` by 17).

2.3 Architectural Differences

The JVM and the ARM have completely different architectures which only have very few features in common. Table 4 summarizes the differences:

Table 4: Differences between JVM and ARM

JVM	ARM
• CISC • zero-operand architecture • stack-based • many registers (256/65536) • very unorthogonal • object-oriented concepts • designed for SW implementation	• RISC • three-operand architecture • memory/register-based • few registers (13 general) • relatively orthogonal • only low-level concepts • designed for HW implementation

2.4 Evaluation of the ARM as a Java Target Architecture

Section 2.3 revealed that there are indeed many differences between the JVM and the ARM. This suggests that the ARM is not very suitable as a target architecture for Java.
For the translation of Java bytecode the following facts cause problems with the ARM architecture:

- The ARM only has a small register set. Without `PC`, `LR` and `SP` there are only 13 registers at free disposal. Additional restrictions are imposed by the ARM Procedure Call Standard (APCS).
  Compared to architectures that have large register files (eg SPARC [Cat91]) with several "register windows" (see [Kep91]) the ARM architecture is somewhat disadvantaged.
- The ARM has different register sets for integer (`R0` – `R15`) and floating-point arithmetics (`F0` – `F7`). Though this is the case for most other microprocessors as well, special care has to be taken that values are stored in the correct type of register, when translating from Java bytecode to ARM machine code.

- In an ARM instruction there are only 12 bits for encoding constants. Constants which cannot be represented in 12 bits have to be fetched from memory. The recompiler needs extra handling for "large" constants.
- Being a 32-bit architecture there is no support for long integers (64 bit). Long integer arithmetics have to be implemented by using two 32-bit registers. In addition, the ARM does not have an integer division or integer remainder instruction. Many JVM instructions (eg `ladd`, `lsub`, `lmul`, `idiv`, `ldiv`, `irem`, `lrem`) do not have any equivalents in the ARM instruction set and require library implementations.
- Most of the special features (see section 2.2.3) of the ARM are not easy to exploit. Conditional execution can produce very efficient code but needs a very intelligent compiler. The same is true for parallel execution of shift operations.

The above list only contains those problems which are in direct connection with the target architecture. The structure of the JVM itself causes additional problems (see section 4.4.1).

Looking at the above list one might argue that the ARM is not at all a good target architecture for Java. However, most other modern microprocessors will cause similar problems. It is a fact that the structure of the JVM is completely different from the structure of nearly all existing microprocessors. Therefore actually *no* modern processor is a good target architecture for Java. There may be some processors which fit better in certain respects, but all in all none of them will really fit well.
An exception to this are new processors which are specially designed with Java in mind (eg PSC 1000 [Can98], Sun picoJava) and adhere very close to the JVM processor architecture (see figure 3).

The reasons for developing a Java bytecode recompiler that targets ARM processors were mainly requirements of the market. ARM RISC chips are very successful, especially in the domain of embedded devices and NCs (see eg [Aue97] or `http://www.arm.com`). In these domains Java support is very important – and will even gain more importance.
Therefore it is a rather prospective idea to spend some time on overcoming all the above implementation problems.

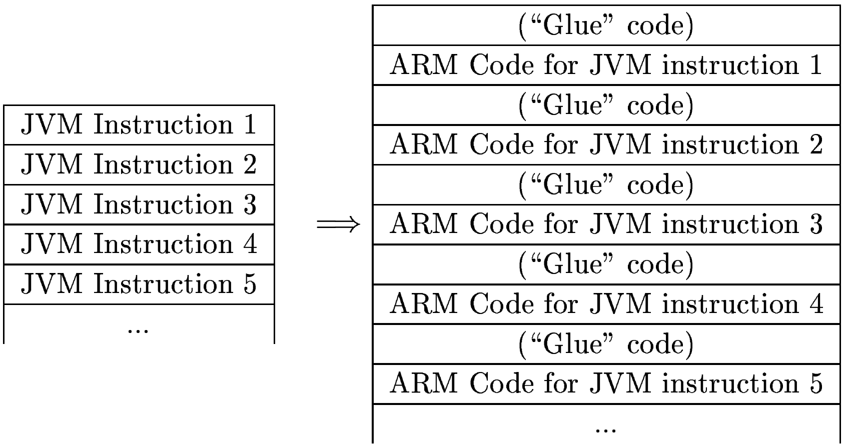
3 Translation Approach

The structure of a compiler which translates "high-level machine code" – like Java bytecode – to "low-level" RISC code is somewhat different from the structure of a regular HLL compiler (like, for example, a C compiler). Often it is possible to use much simpler concepts because the involved grammars are very simple.
Sections 3.1 to 3.4 discuss and illustrate the basic translation principle of the armjrc recompiler.

3.1 Key Idea

The idea behind armjrc is to map each bytecode to an equivalent sequence of ARM machine code instructions and thus to create a complete translation for each method. The code pieces are joined together by inserting adaptive code ("glue" code) between the code sequences if necessary:

Figure 5: Translation Principle of armjrc



To keep the compiler simple, the key idea is to have a translation lookup table which contains the equivalent code sequences for every JVM instruction.
Table 5 shows how such a translation table might look like. Note that not all JVM instructions necessarily produce ARM machine code.

Table 5: Example Translation Lookup Table

`iadd`	`ADD R0,R1,R2`

`ladd`	`ADDS R0,R2,R4`
	`ADC R1,R3,R5`

`aload`	`-`
`aaload`	`B __jr__lib__aaload`
`...`	`...`

3.2 Possible Approaches

For example, the Java statement `l1=l2-l3+4` (with `l1`, `l2`, `l3` being of type `int`) could be compiled into the following bytecode sequence:

```
iload_2  ; Push local variable #2 onto the stack
iload_3  ; Push local variable #3 onto the stack
isub     ; Subtract variable #3 from #2 and push result onto the stack
iconst_4 ; Push constant value 4 onto the stack
iadd     ; Add constant value to the result of the previous subtraction
istore_1 ; Store result in local variable #1
```

In a very naïve approach the JVM stack can be mapped directly to the ARM stack. For example, an `iadd` instruction could be translated to a code sequence which takes two words from the stack, adds them and pushes the result back onto the stack. The ARM registers only serve as temporary storage in this approach.
The translated code would look like this:

```
LDR     R0,[R11,#2*4] ; Load local variable #2 into temporary register
STMFD   SP!,{R0}      ; Push local variable onto the stack

LDR     R0,[R11,#3*4] ; Load local variable #3 into temporary register
STMFD   SP!,{R0}      ; Push local variable onto the stack (*)

LDMFD   SP!,{R0,R1}   ; Pop operands from the stack
SUB     R0,R0,R1      ; Subtract the operands
STMFD   SP!,{R0}      ; Push result onto the stack

MOV     R0,#4         ; Load temporary register with constant 4
STMFD   SP!,{R0}      ; Push constant onto stack (*)

LDMFD   SP!,{R0,R1}   ; Pop operands from the stack
ADD     R0,R0,R1      ; Add the operands
STMFD   SP!,{R0}      ; Push result onto the stack (*)

LDMFD   SP!,{R0}      ; Pop last result from stack (*)
STR     R0,[R11,#1*4] ; Store result in local variable #1
```

This kind of translation is very simple but obviously not very efficient. At first glance it becomes evident that there are a lot of superfluous instructions. A simple optimizer could remove adjacent STMFD[*)] and LDMFD statements (all instructions marked "(*)" could be completely removed by such an optimizer).
However, the ARM is not a stack-based architecture and stack-centric code will not execute very efficiently because memory access will always be much slower than register access. Even after optimization the produced code is far from what an ARM assembly programmer would call well-coded.

The above bytecode sequence could be translated into the following fragment of ARM machine code (using register R11 as pointer to the local variables):

```
LDR     R0,[R11,#2*4] ; Load local variable #2 into temp. register R0
LDR     R1,[R11,#3*4] ; Load local variable #3 into temp. register R1
SUB     R0,R0,R1      ; Subtract the variables and store result in R0
ADD     R0,R0,#4      ; Add the constant value 4 to the result in R0
STR     R0,[R11,#1*4] ; Copy result from R0 into local variable #1
```

Machine code output like this already requires a relatively intelligent translator. In the above example the stack operations have been optimized away, but local variables are still held in memory (instead of using the processor's registers).
When the registers are also used for storing the local variables the resulting code is even more simplified:

```
SUB    R1,R2,R3      ; Perform "l1 = l2 - l3"
ADD    R1,R1,#4      ; Perform "l1 = l1 - 4"
```

Generally armjrc was designed to produce code in the style of this last code example. armjrc is intended to produce good code at first go instead of creating bad code which is subsequently passed to an optimizer (which is done by many other recompilers; for example, [MMBC97]).

[*)] The suffix FD specifies the stack type which is used by multiple load/store instructions; commonly a "Full Descending" stack is used

3.3 Basic Translation Principles

In order to achieve the output quality which is claimed in section 3.2 the originally naïve approach has to be extended in several ways.

If the compiler does not simply map the JVM stack to the ARM stack and the local variables to some memory locations it has to keep track of the JVM's state during compilation. That is, the compiler has to trace how the original bytecode uses the stack and the local variables.
For this purpose the compiler keeps a mapping between the processor registers and the JVM stack and variables. Such a mapping looks like this:

L:	`V0`	`V1`	S:	`V0`	`V2`	`--`

"L:" marks the local variables and "S:" marks the stack contents. This mapping uses virtual registers (`V0`, `V1`, `V2`) instead of real, physical processor registers. In the above example, register `V0` is stored in local variable #0 and as well in the stack entry at the bottom of the stack. The virtual register `V1` is stored in local variable #1 and register `V2` is on top of stack. The stack capacity allows another word to be stored, but in the example mapping this word is not occupied (`--`).
armjrc uses this representation for keeping a "state model" of the JVM. According to [Gos95] such a state model can be set up for every point in a bytecode.

It quickly becomes evident that for such an approach the JVM instruction set is split into two types of instructions:

- Code-generating instructions
- State-affecting instructions

For code-generating instructions the compiler will really produce equivalent ARM machine code. For example, `iadd`, `iaload` or `invokevirtual` are clearly instructions for which machine code has to be generated. Of course, many of these instructions also affect the state of the JVM model which is tracked by the compiler, so they also have the property of being "state-affecting" as well.

JVM instructions which are solely state-affecting do not cause the compiler to produce any code. Only the state of the tracked JVM model is updated according to the nature of the instruction. All push and pop instructions of the JVM are only state-affecting. Actually all these instructions only help to overcome the inability of the JVM to perform operations directly on local variables. The push and pop instructions simply transport data to the stack and back to the local variables. The reason for their existence is implied by the dual "register model" of the JVM which consists of a separate operand stack and a set of local variables.
The ARM is not a stack-based architecture and can perform all operations directly on its registers. In best case both the stack and the local variables are stored in the ARM register set. However, the limited number of registers might require some swapping (see section 4.6).

Table 6 shows the JVM instructions that are merely state-affecting and do not produce any ARM code.

Table 6: State-affecting JVM Instructions

iload	iload_0	iload_1	iload_2	iload_3	
lload	lload_0	lload_1	lload_2	lload_3	
fload	fload_0	fload_1	fload_2	fload_3	
dload	dload_0	dload_1	dload_2	dload_3	
istore	istore_0	istore_1	istore_2	istore_3	
lstore	lstore_0	lstore_1	lstore_2	lstore_3	
fstore	fstore_0	fstore_1	fstore_2	fstore_3	
dstore	dstore_0	dstore_1	dstore_2	dstore_3	
pop	pop2				
dup	dup2	dup_x1	dup2_x1	dup_x2	dup2_x2
swap					

Not all existing recompiler implementations take into account that these instructions only perform some kind of register renaming (or register aliasing) and should not produce any machine code output. For example, Toba [PTB+97] and Harissa [MMBC97] both also generate code for the merely state-affecting instructions (eg Harissa translates an `iload_1` to the C code `si0 = vi1`).

3.4 Example armjrc Translation Process

Obviously the right column of a translation table as presented in table 5 need not only contain ARM instructions but also certain transition codes which update the state of the JVM model kept by the compiler ("JVM transition").

Table 7 shows an example translation. The first column ("Bytecode") contains the original JVM instruction sequence. The second column contains the JVM transitions which influence the tracked JVM state and the third column shows actually produced ARM instructions. In the last column the tracked JVM state is shown.

Column two and three show exactly the translation sequence which has been fetched from the translation lookup table. The translation sequence may contain JVM transitions as well as ARM instructions. The registers used here (`r0`, `r1`, etc) are only "ambiguous letters" and do not refer to the real processor registers. They just provide a common context for the ARM instructions and JVM transitions which belong to the same sequence. During the compilation process the recompiler builds a mapping to the virtual registers (`V0`, `V1`, etc).

For example, the JVM transition `LOAD 1 r0` tells the compiler that `r0` now refers to the virtual register which is stored in local variable #1. The virtual registers are mapped to physical registers (`R0`, `R1`, etc) in a later phase of compilation.

Table 7 shows the translation process for the simple class listed in appendix D.1.

Table 7: Example Translation Process

Bytecode	JVM Transitions	ARM Instructions	State of JVM Model
			L: V0 V1 S: -- -- --
aload_0	LOAD 0 r0		
			L: V0 V1 S: V0 -- --
dup	DUP		
			L: V0 V1 S: V0 V0 --
getfield #0	POP r0 PUSH R1	LDR r1,[r0,<#0>]	
			L: V0 V1 S: V0 V2 --
iload_1	LOAD 1 r0		
			L: V0 V1 S: V0 V2 V1
iadd	POP r1 POP r2 PUSH r0	ADD r0,r1,r2	
			L: V0 V1 S: V0 V3 --
putfield #0	POP r0 POP r1	STR r0,[r1,<#0>]	
			L: V0 V1 S: -- -- --
return		MOV PC,LR	

Having a look at the virtual register mappings it becomes clear that the merely state-affecting instructions are simply register renaming operations. For example, after the dup instruction the value of V0 is accessible via three different locations (local variable #0 and the top two stack elements). For the stack computation model of the JVM this is absolutely necessary but on the ARM this triple replication is not required at all.

4 Compilation Steps

The compilation of a Java class file into an AOF file needs several stages of processing. The armjrc recompiler uses the following stages:

- Parse class file
- Load armjrc header file of superclass
- Construct memory layout, create and write header file
- Create empty AOF file
- Preprocess and orthogonalize bytecode
- Translate bytecode
- Map virtual to real registers
- Assemble translated code, generate constants, resolve branches
- Write code

Section 4 describes these processing stages in greater detail.

4.1 Parsing of Class Files

The first step in the compilation process is the parsing of the input file. The contents of the `.class` file have to be extracted and prepared in way that armjrc can easily process them.

4.1.1 Overall Structure of a Class File

A graphical tree overview of the class file format can be found in the section "Class File Format Overview" on page VII.
A more detailed description of all class file components is contained in [MD97] or [LY97].

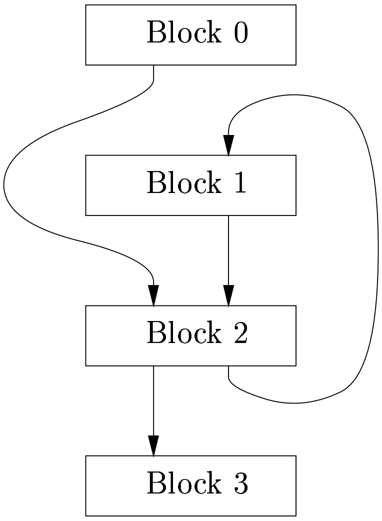
4.1.2 Block Structure of the Bytecode

Like all programming languages the Java bytecode is not a linear sequence of instructions but a graph of so-called "basic blocks" [ASU89]. It is necessary to subdivide the bytecode into these basic blocks. The translation must be done block by block. An example will demonstrate this:

```
block0: iconst_0
        istore_1
        goto block2
block1: iinc 1 1
block2: iload_1
        bipush 10
        if_icmplt block1
block3: return
```

The above bytecode (it is the translation of `for (int i = 0; i < 10; i++);`) decomposes into four basic blocks. The overall block structure is shown in figure 6:

Figure 6: An Example Block Structure



The parser of armjrc will already decompose the bytecode into basic blocks. This saves efforts for the later processing stages.

4.1.3 Parsing Problems

Generally parsing of Java bytecode does not pose any major problems. The “grammar” of Java bytecode is flat and the parser simply needs to determine how many bytes an instruction needs and can then proceed to the next instruction.

Unfortunately, there are some exceptions in the JVM instruction set:
For `tableswitch` and `lookupswitch` instructions the address of the next instruction cannot simply be calculated by adding a constant. Instead, the instruction arguments must be evaluated in order to determine the total length of the instruction. The structure of these two instructions has efficiency reasons; if the jump tables were located in the constant pool an additional dereferencing operation would be necessary. The disadvantage of this design is that parsers always have to consider a special behaviour for these two instructions.
Another problem is caused by the `wide` instruction. This instruction does not simply set the high-byte of the register index but indeed changes the format of the next instruction. When applied to `iinc`, the instruction is prolonged by two bytes; when applied to `iload`, `istore`, `ireturn`, etc, the instruction is prolonged by only one byte.
The above mentioned opcodes require a special handling and therefore increase the complexity of the parser.

4.2 Constructing the Memory Layout for Classes

For compiling a class file the recompiler needs to know some details of the memory layout for the compiled class. In order to compile a class file completely the following memory layout information is required.

- The layout of the virtual method table (VMT)
- The layout of the object itself (its instance variables)

This information can only be determined if the layout of the superclass is known to the compiler. C and C++ compilers determine the memory layout of objects and VMTs by the order in which the members appear in the `.h` file [ES91]. In Java the memory layout can be constructed by parsing all superclasses of the concerned class. However, this would be an unbearable effort, especially for classes which are at the end of a long inheritance path.

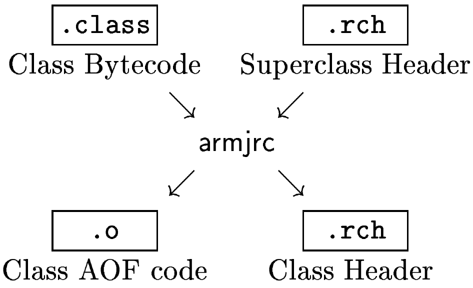
4.2.1 Native Classes

Another question is how the memory layout is determined when native code is involved. The `.h` files for classes written in C or C++ can have a rather complex structure. Though the `.h` file contains all information which is required for the integration of a native class it is again impractical to include a complete parser for C/C++ header files.

4.2.2 Header Files

In order to avoid parsing of all superclasses and of native header files armjrc introduces a new header file mechanism with its own header file format `.rch` (**r**ecompiler **h**eader file; see appendix B). This file format is relatively simple to parse.
When compiling a Java class armjrc expects to find `.rch` files for the superclass. When the memory layout has been constructed a `.rch` file for the processed class is written to the file system.
Figure 7 shows which files are loaded and saved:

Figure 7: armjrc Header File Mechanism



Of course the compiler will also need `.rch` files for all other classes which are referred by the processed class.

It is extremely important that the `.rch` file is written before the actual compilation starts. The compiler might trigger sub-processes for the compilation of dependent classes which might already need the memory layout of the processed class.

4.2.3 Object Memory Layout

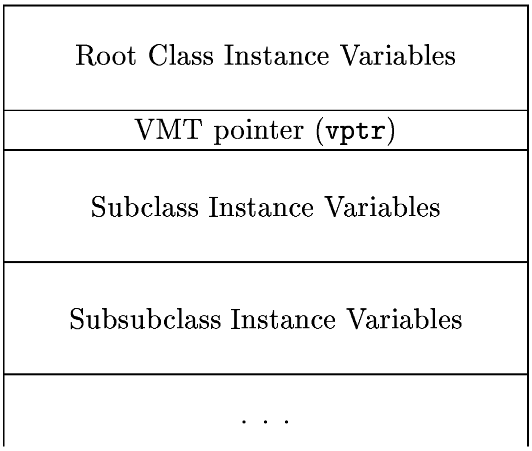
For every involved source class – no matter whether its Java bytecode or native C/C++ code – armjrc needs an `.rch` file. To give a rough impression of how these files look like, here is a listing of `Object.rch`, the header file for the class `java.lang.Object`:

```
# Object.rch
# Recompiler header file for armjrc
# Mirko Raner, 31.07.1998

# Object layout:

0x0004 vtbl
0x0008 end(object)

# Virtual method table:

0x000C equals (Ljava/lang/Object;)Z
0x0014 hashCode ()I
0x001C notify ()V
0x0024 notifyAll ()V
0x002C wait ()V
0x0034 wait (J)V
0x003C wait (JI)V
0x0044 clone ()Ljava/lang/Object;
0x004C finalize ()V
0x0058 end(vtbl)
```

The `vtbl` entry (`0x0004`) marks the offset of the VMT pointer in the object data. Location `0x0000` in the object is not described in the file and is intended for the native object lock variable (needed for thread synchronization).

Surprisingly, in C++ code the VMT pointer is *not* stored at location `0x0000` of the object data structure. Instead the VMT pointer comes *after* the instance variables of the root class (`java.lang.Object` in all cases). Obviously the Acorn C/C++ compiler (amongst other compilers) adopts the memory layout which is presented in [ES91], §10.7c and shown in figure 8:

Figure 8: Object Layout for C++/armjrc


Root Class Instance Variables
VMT pointer (vptr)
Subclass Instance Variables
Subsubclass Instance Variables
. . .

In order to cooperate with existing C/C++ `.o` files, armjrc has to use the same memory layout, otherwise linking with native classes will fail.

The layout of the VMT is again dictated by the output format of the Acorn C/C++ compiler:

Figure 9: VMT Layout for C++/armjrc

<table>
<tr><td>0x0000</td><td>0</td><td>0</td></tr>
<tr><td>0x0004</td><td colspan="2">0</td></tr>
<tr><td>0x0008</td><td>d_0</td><td>i_0</td></tr>
<tr><td>0x000C</td><td colspan="2">Pointer to Method #0</td></tr>
<tr><td>0x0010</td><td>d_1</td><td>i_1</td></tr>
<tr><td>0x0014</td><td colspan="2">Pointer to Method #1</td></tr>
<tr><td>.
.
.</td><td colspan="2">.
.
.</td></tr>
<tr><td>.</td><td>0</td><td>0</td></tr>
<tr><td>.</td><td colspan="2">0</td></tr>
</table>

The d_n and i_n fields are half-words which may contain the delta values needed for multiple inheritance and pointers to members (see [ES91], §8.1.2c and §10.8c). Java supports neither of these concepts and therefore these fields are not used. They have to be set to zero in order to cooperate with native C++ code which accesses them for dynamic method lookup.

The presented VMT structure is directly put into the data segment of the output AOF file (see 4.3.1). First, armjrc creates the method table of the superclass (by parsing the `.rch` file). After that, it adds the newly defined methods of the class and replaces the pointers to overridden methods.

4.2.4 Symbol Naming and Signature Conversion

One objective of armjrc was to make integration of "handwritten" C++ classes as easy as possible. For example, it should be possible to provide a native implementation of the `println(int)` method (in class `java.io.PrintStream`) by simply writing this C++ code:

```
void java_io_PrintStream::println(int value) { ... }
```

That is, there should be no need for a header file generator and the method names in the C++ implementation should still be intuitive, ie they should be the same as the method names in Java. This is somewhat different to the approach taken by JNI [JNI97].
For example, the above method name is mangled by the C++ compiler into the following symbol name:

```
println__19java_io_PrintStreamFi
```

The Acorn C++ compiler – like most C++ compilers – follows the name mangling scheme which is presented in [ES91], §7.2.1c .

armjrc must use the same name mangling scheme in order to ensure interoperability with C++ classes. However, the original Java signature for the above method would be completely different, namely:

```
java/io/PrintStream/println(I)V
```

Therefore armjrc needs a “signature converter” which takes Java signatures and converts them into the corresponding C++ symbol names (according to the naming scheme from [ES91]).
When armjrc generates a new VMT all symbols must conform to the C++ naming scheme, ie the Java signatures must be sent through the signature converter before they are put into the AOF file.

In contrast to the naming scheme of Java descriptors (see [LY97], §4.3) the C++ name mangling scheme does not encode the return type of a method. However, this will not cause any conflicts, because return types cannot be used to distinguish methods in Java (and most other programming languages).

4.2.5 Memory Layout of Primitive Data Types and References

Table 8 shows the memory layout which is used by armjrc for primitive Java types and Java references. Each data type needs either 4 byte or 8 byte, even if there are some bytes unused. More compact storage is only provided for arrays of type `byte` or `boolean` (see section 5.3.7). `float`s and `double`s are stored in IEEE 754 single and double format (ARM FPA mode `S` and `D`; see [ARM96], §9.3). `long`s are stored high-word first; the JVM specification does not specify this, any implementation can freely – but consistently – choose the word order.
Unused bytes are marked with “O” in the table.

Table 8: Memory Layout of Java Types

	Type	←Low Addresses | High Addresses→	
B	`byte`	B OOO	
S	`short`	SS OO	
I	`int`	I I I I	
J	`long`	J J J J	J J J J
F	`float`	FFFF	
D	`double`	DDDD	DDDD
C	`char`	CC OO	
Z	`boolean`	Z OOO	
L	(Reference)	LLLL	

4.3 Creation of the AOF Data Structure

AOF (ARM Object Format) is the object format which is understood by most ARM linkers. The armjrc compiler produces AOF files which resemble the output of the Acorn C/C++ compiler and can be linked with Acorn link.

4.3.1 The ARM Object Format (AOF)

AOF is a derivative of Acorn's highly flexible and extensible "chunk file format". Both file formats are documented in [PRM92] (appendix D) or [ADT94] (appendix E).

An AOF file consists of five different "chunks" which contain all the data that is required by the linker. These five chunks are:

- Header (`OBJ_HEAD`):
  The header chunk contains information about the object file type, the version number, the number of symbols and the number of areas. In addition this chunk contains header data for the areas defined in the `OBJ_AREA` chunk.
- Areas (`OBJ_AREA`):
  The area chunk contains the "areas" of the object file. armjrc will create a code area (`C$$code`) and a data area (`C$$data`). The code area contains the generated machine code, the data area contains static data such as the VMT or pre-initialized static variables.
  Both areas are followed by a table of relocation directives. This table consists of a number of 8-byte entries that tell the linker which memory words need to be relocated and in what way.
- Identification (`OBJ_IDFN`):
  This chunk contains a null-terminated string which gives human-readable information about the producer of the file. The identification of armjrc is:

  ```
  MATHEMA armjrc vsn 1.0.0 (Developed by Mirko Raner) [Jul 27 1998]
  ```
- Symbol Table (`OBJ_SYMT`):
  The symbol table contains the symbols which are used in the object file. Every symbol has a name, an attributes word and optionally a value and an area name.
  Symbolic names are represented by indices into the string table (`OBJ_STRT` chunk). Therefore all symbol table entries have the same length.
- String Table (`OBJ_STRT`):
  The string table contains the names of all symbols and all other string data which is used by any of the other chunks. The table consists of a series of null-terminated strings.

4.3.2 Internal Data Structures

All units which appear in the description of AOF have corresponding classes in the armjrc implementation: for example, chunks, areas, symbols or relocations.
During the compilation process the output file structure is constructed using these classes. When the compiler has finished, the complete AOF file structure is saved to the file system.

4.4 Preprocessing Stage: Orthogonalizing the Bytecode

Both the JVM and the ARM have some peculiarities in their architectures which make it hard to translate code between them. If the working principle of the compiler shall closely adhere to the translation lookup table approach presented in section 3.1 some preprocessing is necessary.

4.4.1 Unorthogonality of the JVM

The JVM and its bytecode have often been criticized for their missing structural orthogonality. Having a closer look at the instruction set of the JVM it becomes obvious that it was in many respects the product of a growing process and not of a perfectly planned design.
Some of these unorthogonalities make the implementation of Java bytecode recompilers very hard. Some things which should be treated differently are treated as equal by the JVM, and some things which really should be treated equally are handled differently.
The unorthogonalities which cause problems for `armjrc` are:

- Some types need two local variables/stack elements instead of one (`long`/`double`)
- The instructions `getfield`/`putfield` and `getstatic`/`putstatic` push or pop a variable number of words, depending on the type of the member which is being accessed. For instance, `getfield aClass/anInteger I` will push one word onto the stack, whilst `getfield aClass/aDouble D` will push two words.
  For some obscure reason these instructions are untyped in contrast to the majority of other instructions (eg `iload`, `lload`, `fload`, `dload`, `aload`).
- The `iinc` instruction is not only used for *incrementing* a local variable but also for *decrementing*. One might argue that decrementing is basically identical to "incrementing by a negative value". However, when it comes to machine code generation, different instructions are needed.
- Constants have different representations depending on their value. There are 4 different ways to represent a constant. For example, the Java source statement `return <constant>;` will be translated differently when 1, 100, 10000 or 1000000 is put in for `<constant>`: the compiler will generate `iconst_1`, `bipush 100`, `sipush 10000` and `ldc <Integer 1000000>` respectively (see appendix D.3).
  This implementation was chosen in favour of efficiency and code size. However, when it comes to translation 4 different instructions have to be handled which basically perform the same task.
  Just like `getfield/putfield` the JVM instructions for constant loading (`ldc`, `ldc_w` and `ldc2_w`) are untyped, ie the constant type cannot be determined from the instruction itself.
- `tableswitch/lookupswitch` and `wide` have (or cause) a variable instruction length. This has already been discussed in section 4.1.3.

4.4.2 Unorthogonality of the ARM

Unorthogonal solutions to certain problems are not an exclusive feature of the JVM. The ARM processor is not a perfect machine either. Thus, both the origin architecture and the target architecture have some flaws which complicate the translation process.

The ARM is an exemplary RISC processor and therefore does not have any problems which different (or even variable) instruction lengths. Every ARM instruction occupies 1 word (4 bytes), no matter whether it is a simple `CMP R0,R1` or a more complex `LDRNEB R8,[R9,-R3,ASR #2]`. Of course, this very compact representation also implies some limitations. Especially when it comes to the encoding of immediate constants the limited space causes problems.

The data processing instructions of the ARM have 12 bit for the represenation of an immediate constant. In order to cover a larger range of numbers these 12 bit are split into 8 bit for an immediate value and 4 bit for a rotation value (see [Jag96], [Gin91]):

11 10 9 8	7 6 5 4 3 2 1 0
4 bit rotate	8 bit immediate

This representation covers all constants which can be calculated by the following formula:

```
constant = immediate ROR (rotate * 2)
```

The ROR operator symbolizes a rightward rotation of the bit pattern on its left by the rotation distance on its right. For instance, the instruction ADD R0,R1,#&3E80 is perfectly valid whilst the instruction ADD R0,R1,#&3E70 cannot be encoded and would be refused by an assembler.
Constants which cannot be encoded as immediate values must be loaded from memory. This requires an additional memory access and might also occupy another register. In order to use the constant &3E70 the following code would be necessary (register R0 can be re-used here):

```
        LDR     R0,[PC,#c_3E70] ; pc-relative addressing
        ADD     R0,R1,R0
            ...
c_3E70  DCD     &3E70           ; generate word &3E70 (assembler directive)
```

However, the compiler should avoid this indirect loading of constants and should use immediate constants where possible.

Another problem is caused by floating-point arithmetics. Floating-point arithmetics are not implemented orthogonally to integer arithmetics. Instead of using the regular (integer) registers R0 to R15 the ARM floating-point accelerator (FPA) uses a dedicated set of floating-point registers F0 to F7. For instance, a JVM imul is translated to MUL R0,R1,R2 but an fmul is translated to FML F0,F1,F2. Therefore the recompiler has to manage two different register sets. This separation of integer and floating-point registers is the case for most processor archictectures and complicates the translation process to some extent.

4.4.3 Modifications of the Bytecode

In order to overcome some of the problems described in section 4.4.1 and 4.4.2 the armjrc recompiler has a preprocessing stage which applies some modifications to the incoming bytecode by replacing some of the opcodes.
The new opcodes replace the “ambiguous” instructions which appear in the bytecode. It must be noted that all modifications are only applied *internally* and that none of the new opcodes will appear in a class file or somewhere else outside of armjrc. Nonetheless, the newly introduced opcodes do neither overlap with the regular JVM opcodes nor with the _quick opcodes (opcode values 203 – 228) which are used internally by Sun's optimized JVM (see [LY97], [MD97]). Therefore armjrc will not cause any problems when it should be coupled with a JVM that uses this optimization.
Not all the problems which were mentioned in section 4.4.1 and 4.4.2 are solved by introducing these new opcodes. However, the replacement of opcodes can be handled by a simple preprocessor and makes the later stages of the compilation process a lot easier.
Table 9 shows the new instructions which are introduced by the internal preprocessor of armjrc.

Table 9: New Opcodes Introduced by the armjrc Preprocessor

Opcode	Mnemonic	Replacement for ...
229 (0xE5)	idec_armjrc	iinc with negative increments
230 (0xE6)	igetfield_armjrc	getfield of int/short/byte/boolean/char
231 (0xE7)	lgetfield_armjrc	getfield of a long
232 (0xE8)	fgetfield_armjrc	getfield of a float
233 (0xE9)	dgetfield_armjrc	getfield of a double
234 (0xEA)	iputfield_armjrc	putfield of int/short/byte/boolean/char
235 (0xEB)	lputfield_armjrc	putfield of a long
236 (0xEC)	fputfield_armjrc	putfield of a float
237 (0xED)	dputfield_armjrc	putfield of a double
238 (0xEE)	igetstatic_armjrc	getstatic of int/short/byte/boolean/char
239 (0xEF)	lgetstatic_armjrc	getstatic of a long
240 (0xF0)	fgetstatic_armjrc	getstatic of a float
241 (0xF1)	dgetstatic_armjrc	getstatic of a double
242 (0xF2)	iputstatic_armjrc	putstatic of int/short/byte/boolean/char
243 (0xF3)	lputstatic_armjrc	putstatic of a long
244 (0xF4)	fputstatic_armjrc	putstatic of a float
245 (0xF5)	dputstatic_armjrc	putstatic of a double
246 (0xF6)	ildc_armjrc	ldc of an int constant
247 (0xF7)	fldc_armjrc	ldc of a float constant
248 (0xF8)	aldc_armjrc	ldc of a string constant
249 (0xF9)	ildc_w_armjrc	ldc_w of an int constant
250 (0xFA)	fldc_w_armjrc	ldc_w of a float constant
251 (0xFB)	aldc_w_armjrc	ldc_w of a string constant
252 (0xFC)	lldc_w_armjrc	ldc2_w of a long constant
253 (0xFD)	dldc_w_armjrc	ldc2_w of a double constant

4.5 Translating the Bytecode

In the translation stage each JVM instruction is translated to a sequence of "compilation items" by simple lookup in a translation table. Usually, this sequence will consist of concrete ARM instructions (which will appear in the AOF file later) and JVM transitions (which help the recompiler to keep track of the internal state of its JVM model). So, a "compilation item" can either be an ARM instruction or a JVM transition.
However, it is not that simple. The actual translation process still poses some problems which have to be solved.

4.5.1 Translation Order

As pointed out in section 4.1.2 the Java bytecode has a block structure, which is preserved by armjrc. Due to this block structure the compiler cannot simply start at the first JVM instruction and proceed until it has reached the last instruction. Instead, the compiler has to process the bytecode block by block. The blocks cannot always be processed in a linear order, because the compiler sometimes does not yet know the initial state of the JVM model for them. Only branches from other blocks yield a valid JVM model and the compiler has to delay the processing of a block until it is the branch target or successor of some other – already processed – block.

For example, the code which is shown in figure 6 needs the translation order 0 – 2 – 1 – 3.
In order to keep track of conditional branches and switches the compiler needs to maintain a stack structure or an "open list" which contains all basic blocks which could already be processed (but haven't been processed yet).

4.5.2 Other Problems

In order to retain the simple translation process presented in section 3.1, the actual translation stage does not yet produce the final machine code. In the subsequent processing stages several things have to be done:

- Registers must be mapped and – if necessary – swapped in and out (section 4.6)
- Constants must be generated (by `LDR`, by `MOV` or immediate; section 4.7.1)
- For non-leaf methods the caller-saved registers have to be saved (section 4.7.2)
- Intra-method branch offsets have to be calculated (section 4.7.3)
- Relocation directives for the linker have to be generated (section 4.7.3)

Most of these actions will insert additional "glue code" into the already compiled method.

4.6 Register Mapping

As discussed in section 2 the ARM and the JVM have completely different processor architectures. The ARM only has 16 registers whilst the JVM potentially has 65536 registers (local variables) and an additional operand stack. In order to produce a good code quality, the JVM local variables and the stack have to be mapped to the ARM registers in an efficient way.

Of course it is always possible that there is not a sufficient number of registers available. In such cases some registers have to be swapped out to memory locations. However, this should be avoided whenever possible.

The compiler does not have an entirely free choice for the register mapping. If the compiler output is to be linked with native object files (eg produced by a C++ compiler) it must conform to the ARM Procedure Call Standard (APCS), which is a set of rules for the usage of registers.

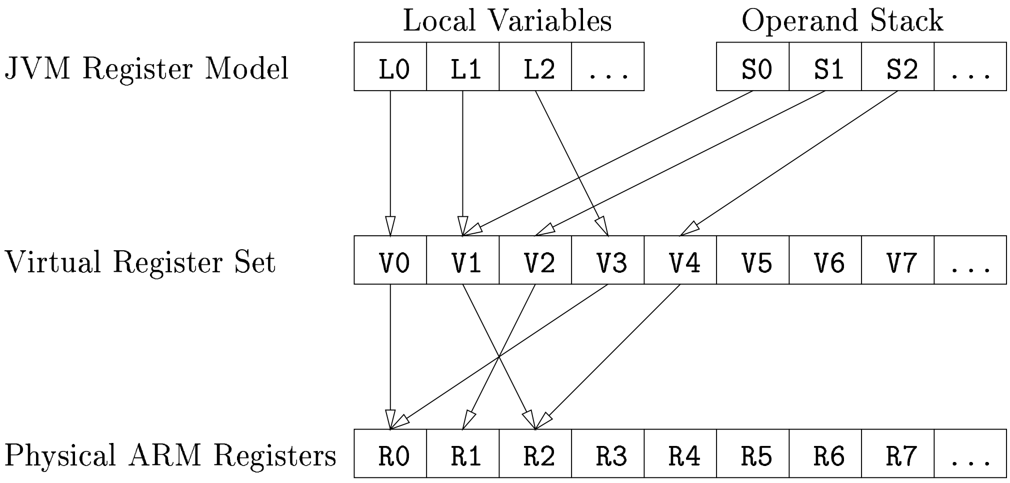
4.6.1 General Mapping Principle

The armjrc recompiler uses a two-stage mapping process in order to establish a mapping between the "registers" of the JVM and the registers of the ARM:

- The first stage maps the local variables and the operand stack of the JVM to "virtual registers". This mapping is done by applying transitions to an internally kept JVM model (see section 3.4)
- The second stage allocates physical ARM registers for each virtual register. This mapping is obtained by a live range analysis described in section 4.6.5.

Figure 10 shows the complete mapping process.

Figure 10: Register Mapping Process of armjrc



4.6.2 Statistical Facts

There are some interesting statistical facts to be noted for the use of local variables and the operand stack on the JVM.

Research conducted by the IMPACT Caffeine team [HGH96] revealed some facts about the stack balance of Java bytecode. The Caffeine register mapping model only works when the "residue" of each basic block is 0, ie the number of pushes is equal to the number of pops and no data is left on the stack on exit from the block. The Caffeine compiler indeed checks this residue. It was observed that code coming from "valid" sources (such as Sun's original javac) always has a balanced stack state, though other compilers are conceivable which might produce code without this property.
armjrc credulously assumes that all incoming code has a stack residue of 0 and does not explicitly verify this fact. Therefore armjrc might run into problems with bytecodes produced with other compilers than the original javac.

The armjrc tool allows to extract some statistical data from class files[*)]. For statistical purposes the `classes.zip` file of JDK 1.0.2[**)] for Solaris was analyzed. Table 10 and 11 summarize the results of this analysis.

[*)] This can be done with the command-line option `-statistics` of armjrc

[**)] The `classes.zip` file for JDK 1.1 is not very representative concerning instruction usage because it contains a lot of table data for internationalization (I18N), which leads to some untypical bytecode distributions

Table 10: Stack Depth Statistics for `classes.zip`

```
cumulative stack depth:           50% |
    0:    57 
    1:   809 #########
    2:  2021 ######################
    3:  2927 #################################
    4:  3520 #######################################
    5:  3933 ############################################
    6:  4162 ###############################################
    7:  4310 ################################################
    8:  4363 #################################################
    9:  4392 #################################################
   10:  4401 #################################################
   11:  4408 #################################################
   12:  4413 #################################################
   13:  4418 #################################################
   14:  4421 ##################################################
```

Table 11: Cumulative Use of Local Variables for `classes.zip`

```
cumulative local variable count:  50% |
    0:    78 
    1:  1307 ##############
    2:  2348 ##########################
    3:  3130 ###################################
    4:  3589 ########################################
    5:  3843 ###########################################
    6:  4045 #############################################
    7:  4166 ###############################################
    8:  4229 ###############################################
    9:  4290 ################################################
   10:  4320 ################################################
   11:  4349 #################################################
   12:  4372 #################################################
   13:  4387 #################################################
   14:  4395 #################################################
   15:  4401 #################################################
   16:  4409 #################################################
   17:  4409 #################################################
   18:  4410 #################################################
   19:  4412 #################################################
   20:  4413 #################################################
   21:  4415 #################################################
   22:  4419 #################################################
   23:  4419 #################################################
   24:  4419 #################################################
   25:  4419 #################################################
   26:  4419 #################################################
   27:  4419 #################################################
   28:  4420 #################################################
   29:  4420 #################################################
   30:  4420 #################################################
   31:  4421 ##################################################
```

The statistics show that 66% of all the methods (2927 of 4421) need a maximum stack depth of 3. If even 4 registers could be reserved for the stack contents, this would enable efficient stack usage for nearly 80% (3520 of 4421) of the methods!

The local variable statistics (table 11) again shows that roughly 53% of the methods need only 2 local variables or less (2348 of 4421). 81% of all methods need up to 4 variables. Only 2.28% of the methods need more than 10 variables.

None of the methods use more than 31 variables. This implies that the `wide` opcode is never used to access the local variables between 256 and 65535. Nonetheless it is possible to write code which uses more than 256 variables (see appendix D.4). However, these are pathological cases which will never occur in regular code.

Though the ARM only has a very small number of registers (in comparison to the JVM), the above statistics prove that it should be possible to map the JVM registers efficiently to ARM registers – at least for a large percentage of the code.

4.6.3 The ARM Procedure Call Standard (APCS)

The ARM Procedure Call Standard (APCS) is a set of rules which ensures that AOF files produced with different compilers (C, C++, FORTRAN, PASCAL and of course armjrc) can be linked together without any problems. Basically APCS defines how the ARM registers have to be used and how arguments are passed between procedures (or methods).
APCS is documented in appendix C of the RISC OS 3 PRM [PRM92] or in appendix F of the Desktop Tools Manual [ADT94].

In the APCS the ARM registers are denoted by different register names. The APCS register names are shown in table 12 and 13.

Table 12: APCS Integer Register Names

R0	a1	Argument 1 or scratch register or integer result
R1	a2	Argument 2 or scratch register
R2	a3	Argument 3 or scratch register
R3	a4	Argument 4 or scratch register
R4	v1	Register variable
R5	v2	Register variable
R6	v3	Register variable
R7	v4	Register variable
R8	v5	Register variable
R9	v6	Register variable
R10	sl	Stack limit
R11	fp	Frame pointer
R12	ip	Scratch register
R13	sp	Stack pointer
R14	lr	Link register or scratch register
R15	pc	Program counter

Table 13: APCS Floating-Point Register Names

F0	f0	FP scratch register
F1	f1	FP scratch register
F2	f2	FP scratch register
F3	f3	FP scratch register
F4	f4	FP register variable
F5	f5	FP register variable
F6	f6	FP register variable
F7	f7	FP register variable

Registers a1 to a4 (R0 – R3) are used as scratch registers and for passing arguments and return values. On return from a procedure these registers may be corrupted, a1 may contain a return value. If a1 to a4 shall be preserved across a procedure call, the calling procedure must save them on the stack (a1 to a4 are therefore also known as "caller-saved" registers).
The APCS defines v1 to v6 (R4 – R9) as register variables. Procedures must preserve these registers on entry and restore them on exit (if they are corrupted within the procedure). A procedure can assume that v1 to v6 are not corrupted by a call to another procedure (the registers are therefore called "callee-saved").
The registers sp, lr and pc have their regular meaning, ip is a (caller-saved) scratch register, fp is the frame pointer which points to the most recently created stack backtrace structure and sl is the stack limit which allows explicit stack-limit checking.
The floating-point registers f0 to f3 are caller-saved scratch registers, f4 to f7 have to be preserved by callees.

There are several variants of APCS. amrjrc uses an APCS variant with a combined 26-bit program counter and status register[*)], explicit stack-limit checking and floating-point argument passing via the stack.

Passing of arguments and return values is slightly simplified for armjrc (in comparison to C/C++) because all object structures are passed by pointers (and not by value). Therefore there are no problems with the by-value passing of structs or objects.
armjrc obeys the simple rules which are demanded by APCS. An argument list is treated as a list of 32-bit words. References, ints and floats need 1 word, longs and doubles need 2 words. Up to the fourth word this list is passed in a1 to a4, all remaining words are put onto the stack (accessible via the sp register).

Arguments *must* be passed in the above described way, otherwise linking with C/C++ object files will not work.
Unfortunately, these rules can lead to very disadvantageous constellations. For example, the (non-static) Java method void dummy(int a, int b, double c, float d) gets its arguments passed in a very peculiar way:

R0	R1	R2	R3	[SP,#0]	[SP,#4]
this	a	b	c		d

[*)] ARM architecture versions 1, 2 and 2a split the PC register into a 26-bit program counter and a 6-bit program status register (PSR); architecture versions 3 and 3M provide an emulation of this 26-bit mode [Jag96]

As can be seen easily, the `double` value `c` has been split between a register and the stack. It requires several instructions to get this scattered value into one of the floating-point registers.

If a procedure has a return value, this value is given back in `f0` if it is a floating-point result, in `a1`/`R0` if it is a single-word result (reference or `int`) and via the stack if it is some other value (especially a `long`).

4.6.4 Consequences of APCS for the Register Mapping

A very simple consequence of APCS for register mapping is that registers `R0` to `R3` (`a1`−`a4`) should be allocated preferentially. These registers are "scratch" registers and do not need to be preserved. In contrast to this, `R4` to `R9` (`v1`−`v6`) need to be saved to the stack on method entry (and restored on exit). This requires two additional memory accesses, which should be avoided. In non-leaf methods (ie methods which again call other methods) `a1` to `a4` will be needed very frequently for parameter passing. Therefore the preferential use of these registers will only be practicable for leaf methods.

APCS substantially restricts the use of the processor registers. The final register mapping must not only be efficient but also conforming to APCS. On method entry, method exit and as well before and after a method invocation APCS requires a certain mapping of JVM registers to ARM registers. For example, before a method invocation the stack element which contains the object reference must be placed in `R0` and any arguments must be placed in `R1` to `R3` or on the stack.
An `invokevirtual` of the example method from section 4.6.3 would require the following mapping to ARM registers (S_0 refers to the top element on the JVM operand stack, S_{-1} to the next element, and so on): [*)]

R0	R1	R2	R3	[SP,#0]	[SP,#4]
this	a	b	c		d
S_{-5}	S_{-4}	S_{-3}	S_{-2}	S_{-1}	S_0

Due to APCS it is possible that mapping conflicts appear in a method (eg when the `this` reference, which is mapped to `R0` on entry, is passed to another method as first argument, which must be mapped to `R1`).
Table 14 shows another example for such a mapping conflict. The mapping to ARM registers is written below the virtual registers, if the space is left empty this means that there is no obligatory mapping. The example only shows the obligatory mappings required by APCS, the empty spaces have to be filled in by the register mapping process. Conflicts have to be resolved by inserting appropriate "glue code" that establishes the correct register mapping (see section 4.6.6).

[*)] By the way, the description of `invokevirtual` in [MD97] contains a mistake: the argument order on the stack is not "*arg1, arg2, ... , argN, objectref*" but "*argN, ... , arg2, arg1, objectref*" (downwards from top of stack)

Table 14: An Example Register Mapping Conflict

Bytecode	JVM Tr.	ARM Instr.	State
			L: V0/R0 V1/R1 V2/R2 S: -- -- --
aload_2	LOAD 2 r0		
			L: V0/R0 V1/R1 V2/R2 S: V2/R2 -- --
iload_1	LOAD 1 r0		
			L: V0/R0 V1/R1 V2/R2 S: V2/R2 V1/R1 --
iload_0	LOAD 0 r0		
			L: V0/R0 V1/R1 V2/R2 S: V2/R2 V1/R1 V0/R0
iadd	POP r1 POP r2 PUSH r0	 ADD r0,r1,r2	
Mapping dictated by initial APCS constraints:			L: V0/R0 V1/R1 V2/R2 S: V2/R2 V3 --
***** MAPPING CONFLICT FOR V2 *****			
Correct mapping required for invokevirtual:			L: V0 V1 V2/R0 S: V2/R0 V3/R1 --
invokevirtual println(int)	...	...	
			L: V0 V1 V2/R0 S: -- -- --
return		MOV PC,LR	

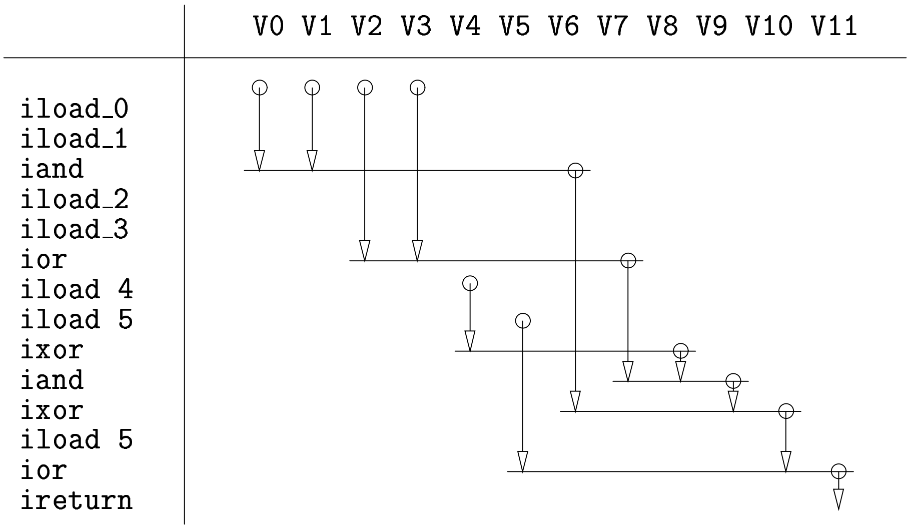
4.6.5 Analyzing "Def-Use Chains"

As can be seen in table 7 (and table 14 as well) every new result in the JVM will produce a PUSH transition which allocates a new virtual register. New virtual registers will even be allocated for intermediary results. Objective of the armjrc register mapping is to map the virtual registers to real, physical registers as efficiently as possible. This also means that several virtual registers share one physical register if their live ranges do not overlap.

One approach for efficient register mapping is "live range disambiguation by def-use chains grouping", presented in [HGH96]. The basic idea of this approach is the "def-use chain", which determines the live range of a virtual register. The def-use chain of a virtual register starts at the instruction where the register is assigned a value for the first time ("def") and ends at the last instruction where the register is read or referred to ("use").

Table 11 shows an example def-use chain analysis for the class shown in appendix D.2. On entry, the 6 int arguments are passed in V0 to V5. However, only V1 to V3 are passed in actual registers (R0 – R3), whilst V4 and V5 are passed via the stack locations [SP,#0] and [SP,#4]. Generally, the def-use chain analysis only refers to values which are located in physical processor registers. Therefore, in the example only V0 to V3 are regarded as "defined" from the beginning, V4 and V5 are regarded as "defined" when they are copied from the stack to real registers.

Figure 11: An Example of a Def-Use Chain Analysis



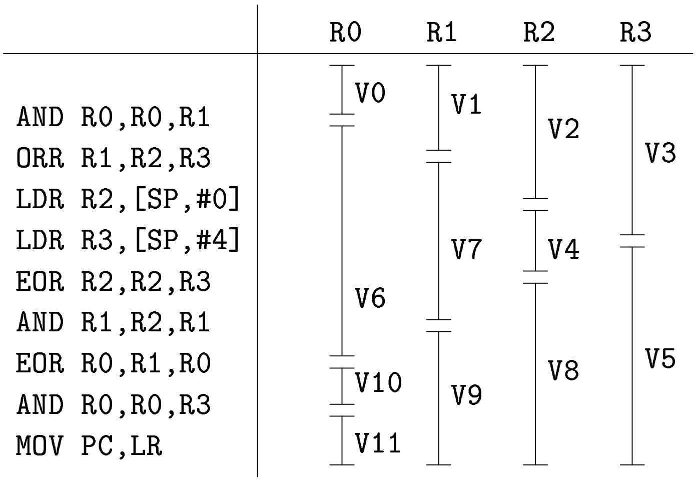
```
iload_0
iload_1
iand
iload_2
iload_3
ior
iload 4
iload 5
ixor
iand
ixor
iload 5
ior
ireturn
```

Obviously, some virtual registers have very short live ranges and could easily share one common physical register. For example, the live ranges of V0, V6, V10 and V11 are non-overlapping; these virtual registers could all be placed in the same physical register.

Before doing any virtual-to-physical register mapping, armjrc conducts such a def-use chain analysis. When armjrc needs a new register for some (maybe intermediary) result, it first looks if there is any register which could be re-used (according to the def-use chain analysis). The allocation principle of armjrc is very close to the getreg function described in §9.6 of [ASU89].

armjrc compiles the example bytecode into the following ARM machine code, which only uses 4 physical registers (instead of 12):

Figure 12: Register Usage and Live Range



```
AND R0,R0,R1
ORR R1,R2,R3
LDR R2,[SP,#0]
LDR R3,[SP,#4]
EOR R2,R2,R3
AND R1,R2,R1
EOR R0,R1,R0
AND R0,R0,R3
MOV PC,LR
```

4.6.6 Resolving Mapping Conflicts

Basically the register mapping process interpolates between the register mappings which are dictated by APCS. Conflicts have to be resolved by moving values from one register to another register or to memory.
For the example conflict presented in table 14 the following "nominal/actual" mapping comparison can be set up:

Registers	V0	V1	V2	V3
actual	R0	R1	R2	
nominal			R0	R1

The register mapper takes such a configuration as input and returns the completed registers mappings and possibly some glue code which resolves any conflicts.
For this example the def-use chain analysis yields that the live range of V3 just starts at the time where the live ranges of V0 and V1 both end. Therefore V3 can indeed be placed in R1 without any problem. The only remaining problem is that V2 is actually located in R2 but should be located in R0. As the life range of V0 has expired, R0 is at free disposal again and the value from R2 can be moved there.
The mapping process will have this result ("--" indicates that the live range of the virtual register has expired and there is no mapping to a physical register required anymore):

Registers	V0	V1	V2	V3
actual	R0	R1	R2	R1
(glue code)	MOV R0,R2			
nominal	--	--	R0	R1

Some of the very common conflicts can be solved by simple patterns which are kept in a kind of "pattern library" in the register mapper.

The first very typical conflict is this:

Registers	V0	V1
actual	R0	R1
nominal	R1	--

V0 is mapped to R0 but should be mapped to R1. The live range of the virtual register located in R1 has just expired.
The pattern which solves this conflict is simply MOV R1,R0.

Another common conflict involves the swapping of two registers. This is the case when the destination register still holds another valid virtual register which cannot be overwritten:

Registers	V0	V1
actual	R0	R1
nominal	R1	R0

This register swapping can be solved by three MOVs and the help of an additional temporary register. However, registers are in short supply and every programmer knows that there is a better solution which uses three XOR operations and does *not* need a temporary register.
The solution pattern for the the swap conflict is:

```
EOR     R0,R0,R1
EOR     R1,R1,R0
EOR     R0,R0,R1
```

When there are absolutely no more uncommitted registers available it is necessary to swap registers from and to memory (eg the top of the stack):

Registers	V0	V1
actual	[SP]	R1
nominal	R0	[SP]

In architecture version 2a of the ARM processor a new instruction was introduced, which solves this problem in a very elegant way:[*] `SWP R0,R1, [SP]`. This also works if the swap-in register and the swap-out register are identical (eg `SWP R0,R0, [SP]`).

4.7 Assembling the Code

When the recompiler has reached the code assembly stage the code is already relatively complete. That is, there is a sequence of ARM instructions which already refer to the physical registers of the processor. Some minor things still have to be fixed, though.

4.7.1 Generating Constants

In general, a constant-pushing instruction like `iconst_1` or `sipush 15984` can be translated to `MOV R0,#1` or `LDR R0,[PC,#constant15984]` (see section 4.4.2).
For efficiency reasons `armjrc` delays the code generation for such instructions and treats constants like special virtual registers.
A little piece of code demonstrates why:

```
iconst_1
iload_0
iadd
istore_0
```

Without delayed constant generation the following ARM code would be generated (assuming that local variable #0 is mapped to `R0`):

```
MOV     R1,#1
ADD     R0,R0,R1
```

It is obvious that this could be re-written as `ADD R0,R0,#1` – but the compiler cannot know that. Therefore constants are treated as special virtual registers which are mapped right after the regular register mapping took place.
If the constant is small enough to fit into the instruction it will be encoded as an immediate constant, otherwise a `MOV` or an `LDR` is inserted as glue code (possibly even more glue code is inserted when there are no more free registers).
Therefore constant-pushing instructions are sometimes merely state-affecting and sometimes also code-generating (see also section 3.3 and 5.3.4).

[*] The `SWP` instruction also serves as a semaphore instruction for thread locking; `SWP` triggers a special bus signal which makes the instruction "atomic"

4.7.2 Preserving Registers

When register mapping and constant generation is done, a list of used registers can be made up for every method. In many cases the list of used registers will also contain the callee-saved registers `R4` to `R9` (`v1`–`v6` in APCS nomenclature). These registers must not be corrupted and have to be preserved on the stack.
In addition to this, non-leaf methods will corrupt the link register when they call another method (eg with a `BL` instruction). Therefore `LR` also has to be saved on the stack for such methods.

A method which corrupts `R4` to `R9` and also calls other methods needs to be surrounded by the following glue code:

```
STMFD   SP!,{R4-R9,LR} ; save registers to the stack
.
. (method code)
.
LDMFD   SP!,{R4-R9,PC} ; restore registers
```

The `LDMFD` statement replaces the usual `MOV PC,LR` at the end of the method: the saved link register is restored directly to the program counter.
If a method does not corrupt all of the callee-saved registers or does not call any subroutines the register list will be smaller than in the above example.

4.7.3 Intra-Method Branches and Relocation Directives

After the last glue code has been inserted the final address of each instruction is fixed, ie no more displacements will take place. At this time the offsets for any branches within the method can be calculated. There are two types of branches:

- Intra-method branches:
  Branches within a method as generated by – for example – a `goto`, a `jsr` or an `iflt`.
- Inter-method branches:
  Branches between methods as generated by calls to the runtime library or non-virtual method invocations.

For intra-method branches the distance between the calling instruction (eg a `B` or a `BL`) and the called method has to be calculated and encoded into the instruction.
All other branches are resolved by the linker. For such branches a relocation directive has to be generated. This relocation directive contains the address of the branch instruction, the symbol name of the branch target and the relocation type (see AOF specification in [PRM92] or [ADT94]).

5 JVM Instruction Classification and Translation Tables

The Java Virtual Machine is a typical CISC (Complex Instruction Set Computer) architecture. The instruction set contains 201 different instructions. The armjrc recompiler has to find an equivalent sequence of ARM instructions for each JVM instruction. The preprocessing stage even adds some more "temporary" instructions (see section 4.4.3).
Section 5.1 to 5.3 discuss how particular instructions can be translated.

5.1 Instruction Set Overview

The JVM instruction set can be subdivided into the following instruction groups [MD97]:

- arithmetic operations
- logical operations
- numeric conversions
- constant operations
- stack manipulation
- flow control
- local variable operations
- array manipulation
- object and array creation
- object manipulation
- method invocation
- exception generation
- synchronization

Of course there is a trivial dichotomy of "simple" instructions and "complex" instructions. In general an `iadd` is much easier to translate than an `invokevirtual`. However, the pre- and postprocessing stages of armjrc make sure that the translation of all instructions only requires a simple table lookup.
Opcodes which would cause problems with this approach are either replaced by the preprocessing stage (see section 4.4.3) or will be "resolved" after the translation.

5.2 Bytecode Statistics

Table 15 shows the frequency of certain opcodes in the `classes.zip` file of JDK 1.0.2 (for other statistical information refer to section 4.6.2).

Table 15: Most Frequent Opcodes in `classes.zip` (Top 10 only)

```
instruction usage:
aload_0         15131 ##################################################
invokevirtual   14490 ###############################################
getfield         9875 ################################
aload_1          5191 #################
dup              5028 ################
aload            4658 ###############
invokespecial    4216 #############
bipush           4187 #############
ldc              3982 #############
iload            3419 ###########
```

In most cases the opcodes with the highest frequency will be `aload_0` (accessing the `this` pointer), `invokevirtual` (calling a method) and `getfield` (accessing an instance variable) – no matter what kind of class files are analyzed.

Opcodes which do not normally occur in compiler output are `nop`, `goto_w`, `jsr_w` and `dup2_x2`. It must be supposed that existing `javac` implementations do not use these instructions. The `javac` compiler as of JDK 1.1 obviously makes no use of the `swap` instruction either.

The `fstore_0` and `dstore_0` opcodes appear very seldom, too (0 occurrences in `classes.zip` of JDK 1.0 and 1.1). The reason for this strange effect is probably that all non-`static` methods use variable #0 for the `this` reference, and most `static` methods obviously have an `int`, `long` or reference as first variable (ie variable number #0). The static methods of class `java.lang.Math` which have a `double` or `float` as first argument are either implemented `native` or do not store back any results to the local variable.

The above statistical data can be very useful for the implementation of the actual bytecode translator. It allows to decide which codes are most important and which ones should be implemented preferentially in order to get first results as fast as possible.

5.3 Translation Tables

Section 5.3.1 to 5.3.11 discuss the translation of particular JVM instruction groups.

5.3.1 Arithmetic Operations

Arithmetic operations can be translated very easily into ARM machine code. The only problem is the mapping of the stack positions to ARM processor registers.

Table 16 shows an overview of the ARM machine code fragments which correspond to the arithmetic JVM instructions. The registers `rn` and `fm` in the table have to be mapped to the actual registers of the ARM by the compiler.

The JVM transitions are not listed in the table because they are in all cases intuitive and would only uselessly enlarge the table. For example, all binary `int` operations will have leading `POP R1,POP R2` transitions and a trailing `PUSH R0` transition. `long` operations use two adjacent registers and any floating-point operations use transitions with the floating-point registers.

Table 16: Translation of Arithmetic Instructions

Operation	i...	l...	f...	d...
...add	ADD r0,r1,r2	ADDS r0,r2,r4 ADC r1,r3,r5	ADFS f0,f1,f2	ADFD f0,f1,f2
...sub	SUB r0,r1,r2	SUBS r0,r2,r4 SBC r1,r3,r5	SUFS f0,f1,f2	SUFD f0,f1,f2
...mul	MUL r0,r1,r2	BL __jr_lmul	FML f0,f1,f2	MUFD f0,f1,f2
...div	BL __jr_idiv	BL __jr_ldiv	FDV f0,f1,f2	DVFD f0,f1,f2
...rem	BL __jr_irem	BL __jr_lrem	RMFS f0,f1,f2	RMFD f0,f1,f2
...neg	RSB r0,r0,#0	RSBS r0,r0,#0 RSC r1,r1,#0	MNFS f0,f0	MNFD f0,f0

The `lmul`, `idiv`, `ldiv`, `irem` and `lrem` operations have no corresponding ARM instruction and must be implemented by jumps into the runtime library (see appendix C).

5.3.2 Logical Operations

Logical operations can be translated very easily, too. A problem is that the simple translation approach of armjrc cannot exploit the parallel execution of shifting operations (see section 2.2.3). The Java code `r1 += r2 >>> r3;` will be translated by armjrc to something like:

```
MOV     R0,R2,LSR R3
ADD     R1,R1,R0
```

A good ARM assembly programmer will instantly realize that the `MOV` instruction can be left out and the temporary register `R0` is not really needed:

```
ADD     R1,R1,R2,LSR R3
```

Unfortunately this optimization cannot be done during translation and requires postprocessing by an optimizer.

Table 17 shows the translation of the logical instructions. Shift operations for `long` arguments are translated as jumps to the runtime library because their implementation is too long to be assembled inline (at least six ARM instructions).
Again `PUSH`/`POP` transitions have been omitted for the sake of brevity.

Table 17: Translation of Logical Instructions

Operation	i...	l...
...shl	MOV r0,r1,ASL r2	BL __jr_lshl
...shr	MOV r0,r1,ASR r2	BL __jr_lshr
...ushr	MOV r0,r1,LSR r2	BL __jr_lushr
...and	AND r0,r1,r2	AND r0,r2,r4 AND r1,r3,r5
...or	ORR r0,r1,r2	ORR r0,r2,r4 ORR r1,r3,r5
...xor	EOR r0,r1,r2	EOR r0,r2,r4 EOR r1,r3,r5

5.3.3 Numeric Conversions

The JVM has 6 numeric data types (`byte`, `short`, `int`, `long`, `float`, `double`) and a “pseudo-numeric” type for Unicode characters (`char`). `boolean` is treated as a `byte`.
See [GJS96] or [LY97] for details.

Internally the JVM only supports the types `int`, `long`, `float` and `double`. For this reason, there are only 15 numeric conversions in the instruction set of the JVM:

Table 18: Overview of Numeric Conversion Instructions of the JVM

From/To	int	long	float	double	byte	short	char
int	-	i2l	i2f	i2d	i2b	i2s	i2c
long	l2i	-	l2f	l2d	-	-	-
float	f2i	f2l	-	f2d	-	-	-
double	d2i	d2l	d2f	-	-	-	-
byte	-	-	-	-	-	-	-
short	-	-	-	-	-	-	-
char	-	-	-	-	-	-	-

Table 19 shows the translation of these instructions:

Table 19: Translation of Conversion Instructions

Operation	i...	l...	f...	d...
...2i		POP r0	POP f0 FIXZ r0,f0 PUSH r0	POPD f0 FIXZ r0,f0 PUSH r0
...2l	POP r0 MOVS r2,r0 MVNMI r1,#0 PUSH r1 PUSH r2		POP f0 BL __jr_f2l PUSH r0 PUSH r1	POPD f0 BL __jr_d2l PUSH r0 PUSH r1
...2f	POP r0 FLTS f0,r0 PUSH f0	POP r0 POP r1 BL __jr_l2f PUSH f0		POPD f0 MVFS f0,f0 PUSH f0
...2d	POP r0 FLTD f0,r0 PUSHD f0	POP r0 POP r1 BL __jr_l2d PUSHD f0	POP f0 MVFD f0,f0 PUSHD f0	
...2b	POP r0 MOV r0,r0,LSL #24 MOV r0,r0,ASR #24 PUSH r0			
...2s	POP r0 MOV r0,r0,LSL #16 MOV r0,r0,ASR #16 PUSH r0			
...2c	POP r0 MOV r0,r0,LSL #16 MOV r0,r0,LSR #16 PUSH r0			

In table 19 the JVM transitions are included as well. They appear left-aligned in the table, whilst actual ARM instructions are right-aligned.

Some of the translations might need clarification:

- Most of the FPA mnemonics can have some suffixes which influence their operation mode: `Z` indicates that "round towards zero" should be used instead of "round to nearest" (the default), `S` and `D` determine the precision which is used (single or double).
- The translation sequences of `i2b`, `i2c` and `i2s` use shifting operations in order to clear certain bits and to perform the sign extension. This is more efficient than setting or clearing bits with `ORR` and `BIC` because the constants `0xFFFF`, `0xFFFF0000` and `0xFFFFFF00` cannot be encoded as immediate constants and would require an additional memory access.
- As the ARM and the ARM FPA have no direct support for 64-bit integer arithmetics the conversion between `long`s and floating-point types is a little complicated and is therefore handled by the runtime library.

5.3.4 Pushing Constants onto the Stack

Table 20 lists all JVM instructions which push a constant value onto the stack.

Table 20: JVM Instructions for the Generation of Constants

iconst_0	iconst_1	iconst_2	iconst_3	iconst_4	iconst_5
iconst_m1					
lconst_0	lconst_1				
fconst_0	fconst_1	fconst_2			
dconst_0	dconst_1				
aconst_null					
bipush	sipush				
ildc_armjrc		fldc_armjrc		aldc_armjrc	
ildc_w_armjrc		fldc_w_armjrc		aldc_w_armjrc	
lldc_w_armjrc		dldc_w_armjrc			

The last eight instructions in table 20 are only generated by the preprocessor and replace "untyped" instructions from the standard instruction set of the JVM.
In the translation phase these instructions will just push constants onto the stack of the internally stored JVM model. In the "constant generation stage" the compiler will have a look at the values and decide whether they can be encoded immediately or have to be generated with `MOV` or `LDR`.

5.3.5 Stack and Local Variable Operations

As pointed out in section 3.3 the push/pop operations for local variables are only "state-affecting" and do not generate any ARM machine code. Table 6 in section 3.3 lists all these instructions.

In the original JVM instruction set there is one local variable operation which is not a load/store operation: the `iinc` operation, which also directly operates on local variables and not on the stack. In order to make compilation easier this opcode is changed to `idec_armjrc` by the preprocessor when the encoded increment is negative.

For a regular `iinc` or `idec_armjrc` the offset is always small enough to be encoded as an immediate constant. When the instruction is preceded by a `wide` instruction (which is not as uncommon as one might think) this efficient encoding may be not possible because the offset is too large.
In such cases armjrc does not try to use the immediate encoding and always fetches the offset constant from memory. This may yield sub-optimal results but is a very quick and elegant way to solve the problem.

The translation of `iinc` is summarized in table 21 (JVM transitions have been omitted).

Table 21: Translation of the `iinc` Instruction

Operation	(regular)	`wide`

`iinc`	`ADD r0,r0,#<increment>`	`LDR r1,[PC,#<adr_of_increment>]`
		`ADD r0,r0,r1`

`idec_armjrc`	`SUB r0,r0,#<increment>`	`LDR r1,[PC,#<adr_of_increment>]`
		`SUB r0,r0,r1`

5.3.6 Flow Control

A large number of flow control instructions is provided by the JVM. These instructions can be split into unconditional branches, conditional two-way branches and multiple branches.
Table 22 is an overview of the flow control instructions of the JVM.

Table 22: Overview of Flow Control Instructions of the JVM

Unconditional branches	`goto` `ret`	`goto_w` `ret_w`	`jsr`	`jsr_w`

Conditional branches	`ifeq`
	`ifne`
	`if_acmpeq`
	`if_icmplt`
		`ifnull`
		`ifnonnull`
		`if_acmpne`
		`if_icmpgt`
			`iflt`
			`ifgt`
			`if_icmpeq`
			`if_icmple`
				`ifle`
				`ifge`
				`if_icmpne`
				`if_icmpge`

Multiple branches	`lookupswitch tableswitch`			

The translation of unconditional branches is absolutely trivial: `goto/goto_w` are realized as a `B` instruction (branch), `jsr/jsr_w` are realized as `BL` (branch and link) and `ret/ret_w` are translated to MOV PC,LR or merged with a stack restore operation (eg LDMFD SP!,{R4,R5,PC}; see also section 4.7.2).

Two-way branches of the type if*condition* are easy to translate, too. Each of the JVM conditions eq, ge, gt, le, lt and ne corresponds to an equivalent ARM condition (EQ, GE, GT, LE, LT and NE respectively; a complete list of all 16 ARM conditions and the corresponding bit patterns can be found in [Jag96], §3.3.1 or [Gin91]). ifnull and ifnonnull are translated like ifeq and ifne respectively. The translation pattern is:

if*condition* <target>	POP r0 CMP r0,#0 B*CONDITION* <equivalent_of_target>

For example, an ifgt instruction might me translated to CMP R0,#0 followed by a BGT.

The translation pattern for branches of the type if_icmp*condition* is very similar:

if_icmp*condition* <target>	POP r0 POP r1 CMP r0,r1 B*CONDITION* <equivalent_of_target>

Here again, the if_acmpeq/if_acmpne instructions can be treated like if_icmpeq/if_icmpne.

The Java Virtual Machine provides two different types of multi-way branch: tableswitch and lookupswitch. Both instructions are used as translation of Java's switch statement, the javac compiler decides which one is used (see [LY97], §7.10).

tableswitch is more efficient but is only applicable when the case values of the switch statement are all very close together. For example, a switch with the case values 1000, 1001, 1002, ..., 1010 could be realized as a tableswitch, but a switch with the case values 1, 10, 100, 1000, ..., 10000000 could not.
On the ARM a tableswitch can be implemented like this: *)

```
CMP     R0,#<highest_index>
SUBLT   R0,#<lowest_index>
LDRLT   PC,[PC,R0,LSL #2]
B       <branch_target_default>            ; index is out of range
DCD     <branch_target_lowest_index>
DCD     <branch_target_lowest_index_plus_1>
.
.
.
DCD     <branch_target_highest_index>
```

This code example assumes that the switch value is passed in R0.

When the case values are too scattered to be fitted into a table the javac compiler will select the lookupswitch instruction. As its name suggests, lookupswitch performs a "look up" for each

*) This is a well-known construction to ARM assembly programmers, it can also be found in [Jag96]; however, the example code there contains a mistake: in the code listing on page 4-7 the instruction LDRLT PC,[PC,LSL #2] should of course read LDRLT PC,[PC,R0,LSL #2]

incoming switch value, ie it *searches* for the appropriate case. Of course, this is very inefficient. When armjrc generates the translation sequence for a lookupswitch it must take into account whether the case values can be encoded immediately or have to be fetched from memory (see section 4.4.2).
For example, a lookupswitch with the case values 1, 100, 10000 and 1000000 looks like this:

```
        CMP     R0,#1
        BEQ     <branch_target_1>
        CMP     R0,#100
        BEQ     <branch_target_100>
        LDR     R1,[PC,#c_1e4]
        CMP     R0,R1
        BEQ     <branch_target_10000>
        LDR     R1,[PC,#c_1e6]
        CMP     R0,R1
        BEQ     <branch_target_1000000>
        B       <branch_target_default>
c_1e4   DCD     10000
c_1e6   DCD     1000000
```

Again the switch value is passed in R0. Additionally the code requires R1 to be available as a temporary register.

There are five more instructions in connection with flow control: dcmpg, dcmpl, fcmpg, fcmpl and lcmp. In the prototype version of armjrc these instructions are implemented as jumps into the runtime library. As these instructions appear rather frequently – and inlined implementations are indeed possible – this will be repaired in a final version.

5.3.7 Array Manipulation

Like most other Java runtime environments armjrc organizes arrays in only three different element sizes: 1 byte, 4 byte or 8 byte. shorts and chars are stored as 4 bytes, booleans are stored as 1 byte.

armjrc assembles array instructions inline, ie for efficiency reasons the ARM code for these instructions is *not* placed in the runtime library.
Before an array access is performed the code ensures that the array index i is in the range $-1 < i < n$ where n is the size of the array. When this is not the case an exception will be thrown (by a call into the runtime library).
The size and type of the array are stored *below* the begin of the actual array structure, ie when the array pointer has the value 0x1000 the array size is stored at location 0x0FFC. This memory layout has been adopted from [HGH96] (see §5 there) and is an optimization of Sun's original memory layout.
The code for the array bounds check needs an additional temporary register for all store operations (R12 is used). For load operations this is not necessary because one of the registers can be re-used.

The arraylength instruction, which determines the length of an array, is translated as a jump to the runtime library.

The translations of array access instructions can be found in table 23.

Table 23: Translation of Array Instructions

Operation	...aload	...astore
i...	LDR r2,[r1,#-4]	LDR R12,[r1,#-4]
f...	CMP r2,r0	CMP R12,r0
s...	CMNGT r0,#0	CMNGT r0,#0
c...	BLE __jr_arrayexception	BLE __jr_arrayexception
a...	LDR r2,[r1,r0,LSL #2]	STR r2,[r1,r0,LSL #2]
l...	LDR r2,[r1,#-4]	LDR R12,[r1,#-4]
d...	CMP r2,r0	CMP R12,r0
	CMNGT r0,#0	CMNGT r0,#0
	BLE __jr_arrayexception	BLE __jr_arrayexception
	MOV r3,r1	MOV R12,r1
	LDR r2,[r3,r0,LSL #3]![*)]	STR r2,[R12,r0,LSL #3]!
	LDR r3,[r3,#4]	STR r3,[R12,#4]
b...	LDR r2,[r1,#-4]	LDR R12,[r1,#-4]
	CMP r2,r0	CMP R12,r0
	CMNGT r0,#0	CMNGT r0,#0
	BLE __jr_arrayexception	BLE __jr_arrayexception
	LDRB r2,[r1,r0]	STRB r2,[r1,r0]

5.3.8 Field Access

As already discussed in section 4.4.3 the armjrc preprocessor will replace `getfield`, `putfield`, `getstatic` and `putstatic` instructions when they refer to an `int`, a `long` a `float` or a `double`. For access of objects the original opcodes are retained. After that the translation is easy:

Table 24: Translation of Get/Put Instructions

Operation	...get...	...put...
...field	LDR r0,[r1,#<offset>]	STR r0,[r1,#<offset>
...static		
i...field_armjrc		
i...static_armjrc		
f...field_armjrc		
f...static_armjrc		
l...field_armjrc	LDR r0,[r2,#<offset>]	STR r0,[r2,#<offset>
l...static_armjrc	LDR r1,[r3,#<offset>+4]	STR r1,[r3,#<offset>+4]
d...field_armjrc		
d...static_armjrc		

[*)] The “!” in an LDR/STR instruction indicates that a write-back takes place, ie in the above code the newly calculated address is also stored in the base register (R3 = R3 + R0 * 8); the above code shows an elegant way of accessing the adjacent memory locations R3 + R0 * 8 and R3 + R0 * 8 + 4

5.3.9 Object Creation and Inspection

In order to provide maximum flexibility, all memory allocation functions are placed in the runtime library. When armjrc is extended to a complete garbage-collected runtime system, the initial "dummy" implementations (which basically call `malloc`) can be easily replaced by implementations which use a GC memory manager.
There are allocation procedures for objects (`new`), primitive one-dimensional arrays (`newarray`), one-dimensional object arrays (`anewarray`) and multi-dimensional arrays (`multianewarray`).

In – admittedly indirect – conjunction with memory allocation there are two more important instructions: `checkcast` and `instanceof`. These two instructions are translated into library calls as well because they require a larger amount of code. The code must determine the compatibility of two types (given their VMT pointers).

Table 25 summarizes the translation of the above discussed instructions.

Table 25: Translation of Memory Allocation and Object Inspection

`new`	`BL __jr_new`
`newarray`	`BL __jr_newarray`
`anewarray`	`BL __jr_anewarray`
`multianewarray`	`BL __jr_multianewarray`
`instanceof`	`BL __jr_instanceof`
`checkcast`	`BL __jr_checkcast`

Like the original `putfield` and `putstatic` methods, `multianewarray` is an instruction with a variable number of stack pops. Depending on the number of specified dimensions a variable number of dimension sizes has to be popped. As arrays of arbitrary dimension can be created with this instruction there is not possibility to resolve this problem with the preprocessor. The compiler has to take special care of this problem.

5.3.10 Method Invocation

The Java Virtual Machine has four different instructions for method invocation: `invokevirtual`, `invokespecial`, `invokestatic` and `invokeinterface` [MD97].
All these instructions have different semantics and require different code to be generated:

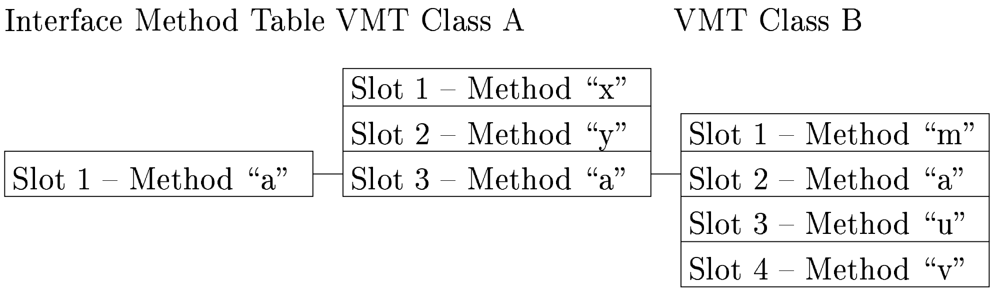
- `invokevirtual`:
  This is the "regular" method invocation in Java, that is, it uses dynamic method lookup and looks in the VMT of the object for the actual address of the code to be called. Usually the VMT pointer is at offset `0x0004` of the object's data structure (see section 4.2.3). APCS dictates that the object reference is passed in `R0`.
  The following code is generated for a virtual method invocation:

  ```
  LDR     R12,[R0,#4]                ; get VMT pointer
  MOV     LR,PC                      ; save return address
  LDR     PC,[R12,#<method_offset>]  ; virtual method invocation
  ```

  `R12/ip` is used as a scratch register for the VMT pointer.

- `invokespecial`:
  This is a method invocation on an object without dynamic method lookup. The code that has to be called is already known at compile-time. Therefore a lookup in the VMT is not necessary. `invokespecial` is used for calls to constructors, `private` methods and `super` methods. As there is no need for a VMT lookup, an `invokespecial` can be simply translated to a `BL` instruction.
- `invokestatic`:
  Basically, `invokestatic` is identical to `invokespecial`. The only difference is that there is no object context passed in `R0`.
- `invokeinterface`:
  Invocation via Java interfaces (see [Fla97], §3) is the most complex of all method invocation types. The problem is that the pointer to an interface method is usually located in different VMT slots for different classes. For example, for class "A" a certain interface method is located in slot 3 and for class "B" the corresponding method is located in slot 2. As it is always possible to subclass an existing class and implement some new interfaces at the same time, this problem cannot be simply avoided.

Figure 13: VMT Slot Allocation for Interface Methods



So for interface methods the index into the VMT is not a constant anymore. The only way to solve this problem is to introduce another indirection. Before an interface method is invoked an "interface mapper" has to look at the object and find out in which slot of the VMT the requested method is located:

```
                                           ; R0 points to the object
        MOV     R12,#<method_offset>       ; move method index to R12
        BL      __im_<interface_name>      ; call interface mapper
                                           ; returns method pointer in R12
        MOV     LR,PC                      ; save return address
        LDR     PC,[R12,#0]                ; virtual method invocation
```

For every interface class armjrc generates such an interface mapper (its symbol name always has the prefix `__im_`). In a special start-up procedure (at runtime) all classes which implement a certain interface must register with the mapper for this interface. Thus the mapper can construct lookup tables which allow to find out the correct slots.

5.3.11 Exceptions and Synchronization

Three important JVM instructions have not been mentioned yet: `athrow` for exception generation and `monitorenter/monitorexit` for thread synchronization.

As pointed out in section 1.2 thread synchronization and exception handling are features which require the support of a runtime system. In the prototype version of armjrc these functions only have dummy implementations in the runtime library (see appendix C).

`monitorenter/monitorexit` could be implemented with the `SWP` instruction of the ARM [Jag96]. The basic mechanism for `athrow` could be realized by the standard C library functions `setjmp` and `longjmp` (see [vdL94], [ACC94]).

Possibly the runtime system in its final version might also have to apply some modifications to the code which is produced by armjrc in order to make synchronization and exception handling work.

6 Outlook

6.1 Completing the Runtime Environment

As already indicated in section 1.2 armjrc only is an implementation of a primitive Java execution engine by means of a recompiler. Most of the features which made Java popular are not yet supported by armjrc.
In order to convert armjrc to a full JVM implementation a complete runtime system has to be built around it. The components of such a runtime system have been described in section 2.1.

6.2 Optimizing the Code

Though armjrc is a relatively intelligent compiler which in many cases produces good code at first go, there is still a lot of potential for a subsequent optimization stage.
Several optimizations could be applied to the code:

- Full exploitation of conditional execution:
  For example, the Java code `if ((r0 == 0) && (r1 == 1)) r2 = 2;` could be efficiently translated into:

  ```
  TEQ     R0,#0
  TEQEQ   R1,#1
  MOVEQ   R2,#2
  ```

  In the current implementation armjrc will not find such optimal translations.

- Tail call optimization:
  This is an optimization technique that optimizes procedure calls which are the last statement within the calling procedure (see APCS notes in [ADT94]). For example, a `BL` followed by a `MOV PC,LR` can be optimized to a single `B` instruction – provided that the correct stack configuration has been set up.

- Exploitation of intra-link-unit calls:
  armjrc treats all branch targets which are not within the same method as inter-link-unit calls and generates relocation directives for the linker. However, calls to other methods *within the same class* (ie within the same link unit) can be implemented more efficiently [ADT94].
- Optimization by Class Hierarchy Analysis (CHA):
  [MMBC97] proposes an optimization approach which transforms virtual method calls to non-virtual calls where possible. As the non-virtual invocation mechanism is more efficient, this can save a lot of time.
- Full exploitation of parallel shift operations.
  See section 5.3.2 for an explanation.
- Fast multiplication with contants:
  The `MUL` instruction on the ARM is very slow – it may need up to 16 additional clock cycles in comparison to an `ADD` or a `SUB`. Section 2.2.3 mentions an approach how parallel shifting can be used for fast multiplication with constants.
  In general, multiplication with a constant of the form $2^n + i$ (where i is $\{-1, 0, 1\}$) can be performed in 1 cycle and a multiplication with a constant of the form $(2^{n_1} + i_1) \times (2^{n_2} + i_2)$ can be performed in 2 cycles.

6.3 Performance Expectations

At the time of finishing this thesis armjrc was unfortunately not yet capable of translating a complete benchmark suite (like eg CaffeineMark or the UCSD benchmark suite). For this reason it was not possible to get meaningful benchmark results and to compare the performance of armjrc with that of its competitors. Anyway only the C code recompilers can be taken for a direct comparison because their code can be translated to ARM – all other recompilers produce code for a certain architecture.

A lot of time has been spent to fit armjrc with a very intelligent code generator. Compared to the rather naïve approaches of [PTB+97] and [MMBC97] (which generate lots of actually useless code) armjrc produces very efficient code – even without additional optimization.

It seems realistic that armjrc can achieve a performance comparable to the JA2S compiler for the SPARC [JRC97]. Except for some special cases JA2S produces code which roughly performs as fast as equivalent C code (see [JRC97], §4.4 for a detailed benchmark analysis). However there are still a lot of possibilites to make armjrc-generated code even faster (see section 6.2).

The advantage of armjrc over C code recompilers is that the compilation process is much simpler and less tools are needed to get from a Java source code to an executable application.

7 Conclusion

The project "armjrc" required *much more* efforts than initially expected.

In the limited time which was available it was hardly possible to implement a fully operative Java bytecode recompiler. Some of the techniques and features which were presented in this thesis are not yet fully implemented and tested. In nearly all compilation stages of armjrc there are still some minor issues which wait for a better solution – and maybe tests might even prove that some of the current solutions are inadequate.

During the implementation a lot of unexpected problems were revealed. Indeed the ARM is a rather unsuitable target architecture for Java. Especially the lack of 64-bit arithmetics on the ARM caused some difficulties.
However, most of the encountered problems were due to the structure of the JVM itself or to the huge differences between the two architectures.

The armjrc prototype is only the first step towards a complete Java implementation for the ARM. The implementation of the final armjrc will cost some more time and efforts. And even after that there is only a "static" Java implementation without any of Java's unique features. The implementation is not completed until there is a fully operative runtime system which supports multi-threading, garbage collection, dynamic class loading and many other features. Needless to say that the implementation of the runtime system is an even larger task than the implementation of the mere recompiler.

However, the first step is done and the increasing popularity of ARM processors in all kinds of electronic devices suggests that it is a promising idea to conduct further research in this direction.

A Operating Environment

The armjrc compiler itself was written in Java. It was developed with Sun's JDK 1.1.6 on a two-processor Sun SPARCstation 20 under Solaris 2.5.

As target system for the compiled code the following machines were used:

- An Acorn Archimedes workstation running RISC OS 3.1 on an ARM 3 processor fitted with an ARM FPA 10 floating-point coprocessor (ARM architecture version 2a)
- An Acorn NetStation NC running NC OS 1.1 on an ARM 7500FE (ARM architecture version 3)

Native libraries were mostly developed on the first machine with Acorn's C/C++ development environment [ACC94].
armjrc is not yet suitable for the new DEC StrongARM SA-110 [MWA+96] processor (ARM architecture version 4). Therefore the produced code might cause problems on RISC OS 3.7 machines, which have these new chips fitted.

B Recompiler Header File Format (`.rch`)

The format of `.rch` files is very simple-structured and easy to parse. It was designed as a simple counterpart to the `.h` files of C and C++. Whilst `.h` files implicitly determine the offsets of class members by the order in which they appear in the file, the `.rch` format explicitly contains numeric offsets. The order of the items within the file is therefore not significant.

`.rch` files only use characters from the ASCII character set. Any characters with Unicode values above 127 must be represented by Unicode escapes (eg `\u03B0`; see [GJS96], §3.3). A German method name `übersetzen` must be represented as `\u00FCbersetzen`, for instance.

The `.rch` file format can be defined by the following BNF grammar (BNF notation according to [Eng88]):

```
<rch file> ::= {<line>}
<line> ::= <empty line>|<comment line>|<offset line>
<empty line> ::= <LF>
<comment line> ::= #<comment><LF>
<offset line> ::= <special offset>|<member offset>
<member offset> ::= <offset><SPACE><name><SPACE><type>
<special offset> ::= <object end>|<vtbl end>|<vtbl>
<object end> ::= <offset><SPACE>end(object)
<vtbl end> ::= <offset><SPACE>end(vtbl)
<vtbl> ::= <offset><SPACE>vtbl
<type> ::= <data type>|<method signature>
<offset> ::= 0x<hex number>
```

Some of the symbols used in this grammar might need clarification:
`<LF>` and `<SPACE>` refer to the linefeed (ASCII 9) and space (ASCII 32) character respectively.

`<name>` refers to a valid Java identifier name (see [GJS96], §3.8).
`<data type>` refers to one of the internal JVM type descriptors (eg `J` for Java type `long`, `Ljava/lang/String;` for string objects or `[[F` for a two-dimensional array of `float` values).
`<method signature>` refers to an internal JVM method signature (eg `(Ljava/lang/Object;)V`, `(II)J` or `()Z`).
`<hex number>` is a hexadecimal number like `C5E2D080` or `CAFEBABE`.

As pointed out above, the lines in the file may be permuted in any order. However, there are certain items which are obligatory: each `.rch` file *must* contain a `<vtbl>`, a `<vtbl end>` and an `<object end>` entry. Otherwise the compiler will be unable to determine the memory layout of objects and VMTs.

C Java Runtime Library Functions Needed by **armjrc**

`__jr_anewarray`	Allocation of a one-dimensional object array
`__jr_arrayexception`	Generation of an `ArrayIndexOutOfBoundsException`
`__jr_arraylength`	Determination of the length of an array
`__jr_athrow`	Exception generation (for arbitrary `Throwables`)
`__jr_checkcast`	Dynamic cast checking
`__jr_d2l`	`double` to `long` conversion
`__jr_dcmpg`	`double` comparison (with 1 on NaN)[*)]
`__jr_dcmpl`	`double` comparison (with -1 on NaN)[*)]
`__jr_f2l`	`float` to `long` conversion
`__jr_fcmpg`	`float` comparison (with 1 on NaN)[*)]
`__jr_fcmpl`	`float` comparison (with -1 on NaN)[*)]
`__jr_idiv`	`int` division
`__jr_instanceof`	Type inspection
`__jr_internalexception`	Exception in the runtime environment
`__jr_irem`	`int` remainder
`__jr_l2d`	`long` to `double` conversion
`__jr_l2f`	`long` to `float` conversion
`__jr_lcmp`	`long` comparison[*)]
`__jr_ldiv`	`long` division
`__jr_lmul`	`long` multiplication
`__jr_lrem`	`long` remainder
`__jr_lshl`	`long` arithmetic shift left
`__jr_lshr`	`long` arithmetic shift right
`__jr_lushr`	`long` logical shift right
`__jr_monitorenter`	Thread lock acquisition
`__jr_monitorexit`	Thread lock release
`__jr_multianewarray`	Allocation of a multi-dimensional array
`__jr_new`	Allocation of an object
`__jr_newarray`	Allocation of a one-dimensional primitive array

[*)] Only included in the prototype runtime library; will be compiled inline in the final version

D Java Example Programs

D.1 Example Program BankAccount.java

For the first runs of the armjrc prototype the simple class BankAccount was used for testing. This is the program code:

```
public class BankAccount
{
    private int balance;

    public void deposit(int amount)
    {
        balance += amount;
    }
}
```

D.2 Example Program Logic.java

In order to test the def-use chain analysis of armjrc this little program, which excessively produces intermediary results, was used:

```
public class Logic
{
    public static int logic(int a, int b, int c, int d, int e, int f)
    {
        return ((a & b) ^ ((c | d) & (e ^ f))) | f;
    }
}
```

D.3 Representation of Constants

The following code shows how the representation of constants changes with the value of the constant. Though all 4 methods have the same structure they are compiled into different code. Disassembling the produced `.class` file with `javap -c` reveals the different translations.

```
public class Constants
{
    public static int return1()
    {
        return 1;
    }

    public static int return100()
    {
        return 100;
    }

    public static int return10000()
    {
        return 10000;
    }

    public static int return1000000()
    {
        return 1000000;
    }
}
```

D.4 Use of the wide Opcode

The class MakeWideTest will generate a Java source code called WideTest.java. When the generated file is translated by javac a class file which uses 513 local variables will be the result.

```
import java.io.*;
public class MakeWideTest
{
    public static void main(String[] arg) throws Exception
    {
        PrintWriter out;
        out = new PrintWriter(new FileOutputStream("WideTest.java"));

        out.println("public class WideTest");
        out.println("{");
        out.println("\tpublic static void main(String[] arg)");
        out.println("\t{");
        out.println("\t\tint v000 = 0;");
        out.println("\t\tint v001 = 1;");

        for (int index = 2; index <= 0x200; index++)
        {
            out.println("\t\tint v"+hex(index)+" = v"+hex(index-1)+
                " + v"+hex(index-2)+";");
        }
        out.println("\t\tSystem.out.println(\"v200 = \"+v200);");
        out.println("\t}");
        out.println("}");
        out.close();
    }

    private static String hex(int number)
    {
        String hex = Integer.toHexString(number);
        while (hex.length() < 3) hex = "0" + hex;
        return hex;
    }
}
```

E Glossary of Abbreviations

AOF: →ARM Object Format

APCS: →ARM Procedure Call Standard

API: Application Programming Interface

ARM: Advanced →RISC Machine

CISC: Complex Instruction Set Computer

FPA: Floating-Point Accelerator

FPE: Floating-Point Emulator

GC: Garbage Collection

HLL: High-Level Language

I18N: Internationalization

IEEE: Institute of Electrical and Electronics Engineers

IMPACT: Illinois Microarchitecture Project utilizing Advanced Compiler Technology

JDK: Java Development Kit

JIT: Just In Time

JITC: →JIT Compiler

JNI: Java Native Interface

JVM: Java Virtual Machine

LR: Link Register

NC: Network Computer

PC: Program Counter

PRM: Programmer's Reference Manual

RISC: Reduced Instruction Set Computer

SP: Stack Pointer

SPARC: Scalable Processor Architecture

VMT: Virtual Method Table

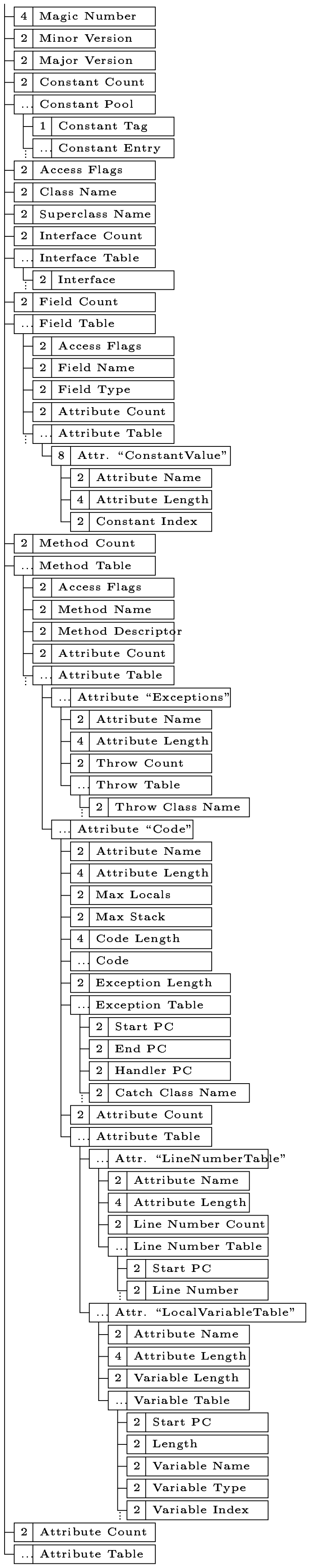
WAT: Way Ahead of Time

WATC: →WAT Compiler

Class File Format Overview

The tree structure on the right is a visualization of the class file format.
The numbers in the left part of the boxes indicate how many bytes a structure needs, "..." means that the structure has a variable length which is determined by a preceding length entry.

If a structure is marked with a "⋮" this indicates that more than one entry with this structure may follow (eg the line number table can consist of several "Start PC"/"Line Number" entries).



References

[ACC94] Acorn Computers Technical Publications Department, Cambridge. *Acorn C/C++*, 1994. ISBN 1 85250 166 9.

[ADT94] Acorn Computers Technical Publications Department, Cambridge. *Desktop Tools*, 1994. ISBN 1 85250 164 2.

[And96] Yukio Andoh. j2c / CafeBabe Java .class to C Translator. URL: `http://tech.webcity.ne.jp/~andoh/java/j2c.html`, 1996.

[ARM96] Advanced RISC Machines Ltd., Cambridge. *ARM 7500FE Data Sheet*, 1996. Document number ARM DDI 0077B; available in the WWW – URL: `http://www.arm.com/Documentation/UserMans/PDF/ARM7500FE.html` .

[ASU89] Alfred V. Aho / Ravi Sethi / Jeffrey D. Ullmann. *Compilerbau.* Addison-Wesley, Bonn, 1989.

[Aue97] Kersten Auel. Bestandsaufnahme Network Computer – Am besten nichts Neues. *iX 11/97*, pages 110 – 117, November 1997.

[Bot97] Per Bothner. Compiling Java for Embedded Systems. URL: `http://www.cygnus.com/news/whitepapers/compiling.html`, 1997.

[Can98] Tom Cantrell. Silicon Update – ShBoom Box. *Circuit Cellar INK Journal, issue #92*, pages 80 – 84, 1998.

[Cat91] Ben J. Catanzaro, editor. *The SPARC Technical Papers.* Springer, New York (NY), 1991. ISBN 0 387 97634 5.

[Dal97] Matthias K. Dalheimer. *Java Virtual Machine.* O'Reilly, Köln, 1997.

[Eng88] Hermann Engesser, editor. *Duden Informatik.* BI-Wissenschaftsverlag, Mannheim, 1988.

[ES91] Margaret A. Ellis / Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading (MA), 1991.

[Fla97] David Flanagan. *Java in a Nutshell.* O'Reilly, Sebastopol (CA), 2nd edition, 1997.

[Gin91] Mike Ginns. *Archimedes Assembly Language.* Dabs Press, Manchester, 2nd edition, 1991. ISBN 1 870336 20 8.

[GJS96] James Gosling / Bill Joy / Guy Steele. *The Java Language Specification.* Addison-Wesley, Reading (MA), 1996.

[Gor98] Rob Gordon. *Essential JNI.* Prentice Hall, Upper Saddle River (NJ), 1998.

[Gos95] James Gosling. Java Intermediate Bytecodes. *ACM SIGPLAN Notices – SIGPLAN Workshop on Intermediate Representations*, 30(3):111 – 118, March 1995.

[HGH96] Cheng-Hsueh A. HSIEH / John C. GYLLENHAAL / Wen-Mei W. HWU. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Paris, 1996.

[Jag96] Dave JAGGAR. *Advanced RISC Machines Architectural Reference Manual.* Prentice Hall, London, 1996.

[JL96] Richard JONES / Rafael LINS. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons Ltd., Chichester, 1996.

[JNI97] SUN MICROSYSTEMS, INC., Mountain View (CA). *Java Native Interface Specification*, May 1997. Revision 1.1.

[JRC97] Patrick JONIEC / Olivier RICHARD / Franck CAPPELLO. Une étude quantitative de l'exécution du Byte-code Java : interpréter ou compiler ? Technical Report LRI-TR-1092, Université de Paris, Paris, 1997.

[Kep91] David KEPPEL. Register Windows and User-Space Threads on the SPARC. Technical Report #91-08-01, University of Washington, Seattle (WA), 1991.

[LY97] Tim LINDHOLM / Frank YELLIN. *The Java Virtual Machine Specification.* Addison-Wesley, Reading (MA), 1997.

[MD97] Jon MEYER / Troy DOWNING. *Java Virtual Machine.* O'Reilly, Sebastopol (CA), 1997.

[MMBC97] Gilles MULLER / Bárbara MOURA / Fabrice BELLARD / Charles CONSEL. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, Portland (OR), 1997.

[MWA$^+$96] James MONTANARO / Richard T. WITEK / Krishna ANNE / Andrew J. BLACK / Elizabeth M. COOPER / Daniel W. DOBBERPUHL / Paul M. DONAHUE / Jim ENO / Gregory W. HOEPPNER / David KRUCKEMYER / Thomas H. LEE / Peter C. M. LIN / Liam MADDEN / Daniel MURRAY / Mark H. PEARCE / Sribalan SANTHANAM / Kathryn J. SNYDER / Ray STEPHANY / Stephen C. THIERAUF. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703 – 1714, November 1996.

[PRM92] ACORN COMPUTERS TECHNICAL PUBLICATIONS DEPARTMENT, Cambridge. *RISC OS 3 Programmer's Reference Manual*, 1992. ISBN 1 85250 110 3.

[PTB$^+$97] Todd A. PROEBSTING / Gregg TOWNSHEND / Patrick BRIDGES / John H. HARTMAN / Tim NEWSHAM / Scott A. WATTERSON. Toba: Java For Applications – A Way Ahead of Time Compiler. Technical report, University of Arizona, Tucson (AZ), 1997.

[vdL94] Peter VAN DER LINDEN. *Expert C Programming – Deep C Secrets.* SunSoft Press / Prentice Hall, Englewood Cliffs (NJ), 1994.

NOTES

This document was typeset using LaTeX 2_ε, figures were drawn with xfig and rendered using the EPIC and EEPIC packages of LaTeX.
The final version was translated to PostScript with dvips 5.58f and printed on a Hewlett-Packard LaserJet 5L with the help of gs (GhostScript).
The printer was generously made available by Jan HILDEBRANDT whom I wish to thank at this place.