Other Architectures

Norm Matloff
University of California, Davis

Chapter 1

Example of RISC Architecture: MIPS

1.1 Introduction

The term **RISC** is an acronym for reduced instruction set computer, the antonym being **CISC**, for complex instruction set computer.

During the 1970s and early 1980s, computers became more and more CISC-like, with richer and richer instruction sets. The most cited example is that of the VAX family (highly popular in the early 1980s), whose architecture included 304 instructions, including such highly specialized instructions as **poly**, which evaluates polynomials. By contrast, the UC Berkeley RISC I architecture, which later became the basis for what is now the commercial SPARC chip in Sun Microsystems computers, had only 32 instructions.

At the same time, the single-chip CPU was starting out on its path to dominance of the market. The more components a computer has, the more expensive it is to manufacture, so single-chip CPUs became very attractive.[1] But the problem then became that the very limited space on a chip—this space is colloquially called "real estate"—made it very difficult, if not impossible, to fit a CISC architecture onto a single chip.[2]

The VAX was a clear casualty of this. When the Digital Equipment Corporation (DEC) tried to produce a "VAX-on-a-chip," they found that they could not fit the entire instruction set on one chip. They were then forced to implement the remaining instructions in software. For example, when the CPU encountered a **poly** instruction, that would cause an "illegal op code" condition, which triggered a **trap** (i.e. internal interrupt); the trap-service routine would then be a procedure consisting of ordinary **add** and **mul** instructions which compute the polynomial.

[1]And there is a performance gain as well. Multi-chip CPUs suffer from the problem that off-chip communication is slower than within-chip.

[2]It would seem that the solution is to simply make larger chips. However, this idea has not yet worked. For example, larger chips have lower **yield** rates, i.e. lower rates of flaw-free chips. Of course, one can try to make chips denser as well, but there are limits here too.

At this point, some researchers at IBM decided to take a fresh look at the whole question of instruction set formulation. The main point is optimal use of real estate. In a CISC CPU chip, a large portion of the chip is devoted to circuitry which is rarely, if ever used.[3] For example, most users of VAX machines did not make use of the **poly** instruction, and yet the circuitry for that instruction would occupy valuable-but-wasted real estate in the chips comprising a VAX CPU.

True, for those programs which did use the **poly** instruction, execution speed increased somewhat,[4] but for most programs the **poly** instruction was just getting in the way of fast performance.

Other factors came into play. For example, the richer the instruction set, the longer the decode portion of the instruction cycle would take. Even more important is the issue of pipelining, which is much harder to do in CISC machine, due to lack of uniformity in the instruction set (in a CISC machine, instructions tend to have different lengths).

For this reason, the IBM people asked, "Why do we need all this complexity," and they built the first RISC machine. Unfortunately, IBM scrapped their RISC project, but later David Patterson of the Division of Computer Science at UC Berkeley became interested in the idea, and developed two RISC CPU chips, RISC I and RISC II, which later became the basis of the commercial SPARC chip, as mentioned earlier.

Patterson's team did not invent the RISC concept, nor did they invent the specific features of their design such as delayed branch (see below). Nevertheless, these people singlehandedly changed the thinking of the entire computer industry, by showing that RISC could really work well. The team considered the problem as a whole, investigating all aspects, ranging from the questions of chip layout and VLSI electronics technology to issues of compiler writing, and through this integrated approach were able to make a good case for the RISC concept. Manolis Katevenis, a Ph.D. student who wrote his dissertation on RISC under Professor Patterson, won the annual ACM award for the best dissertation in Computer Science.[5]

At about the same time, Professor John Hennessy at Stanford University (who was later to become president of the university) was conducting his own RISC project, leading to his founding of the MIPS Corporation, which produced his RISC chip.

Silicon Graphics, Inc. eventually bought MIPS Inc. and SGI uses MIPS processors in its servers and so on, and though SGI is no longer one of the big players in those segments of the industry, MIPS processors are popular in the world of embedded applications.

Today virtually every computer manufacturer has placed a major emphasis on RISC. Even Intel, the ultimate CISC company, has turned to RISC for its most powerful chips. Sun Microsystems has the SPARC chip,

[3]Actually, most CISC chips use a technique called **microcode**, which would make the use of the word "circuitry" here a bit overly simplistic, but we will not pursue that here.

[4]You should think about this. First, **poly**, being a single instruction, would require only one instruction fetch, compared to many fetches which would be needed if one wrote a function consisting of many instructions which would collectively perform the work that **poly** does. Second, at the circuitry level, we might be able to perform more than one register transfer at once.

[5]Published as *Reduced Instruction Set Computer Architectures for VLSI*, M. Katevenis, MIT Press, 1984.

IBM and Motorola have the Power PC chip series used in Macs,[6] and so on.

Of course, this does not imply that RISC is inherently "superior," and as electronic technologies advance in the future, there may well be some resurgence of CISC ideas, at least to some degree.

1.2 A Definition of RISC

There is no universally-agreed-upon definition of RISC, but most people would agree at least to the following characteristics:

- Most instructions execute in a single clock cycle.
- "Load/store architecture"—the only instructions which access memory are of the LOAD and STORE type, i.e. instructions that merely copy from memory to a register or vice versa, such as Intel's

  ```
  movl (%eax), %ebx
  ```

  and

  ```
  movl %ecx, (%edx)
  ```

  Instructions like Intel's **addl %eax, (%ebx)** are not possible.
- Every instruction has the same length, e.g. 4 bytes. By contrast, Intel instructions range in length from 1 to several bytes.
- **Orthogonal architecture** — any operation can use any operand. This is in contrast, for example, to the Intel instruction STOS, which stores multiple copies of a string.[7] Here the only available operands are the registers EAX, ECX and EDI; one is not allowed to use any other register in place of these.
- The instruction set is limited to only those instructions which are truly justified in terms of performance/space/pipeline tradeoffs. The SPARC chip, for instance, does not even have a multiply instruction. If the compiler sees an expression like, say,

  ```
  i = j * k;
  ```

  in a source code file, the compiler must generate a set of add and shift instructions to synthesize the multiply operation, compared to the Intel case, in which the compiler would simply generate an **imull** instruction.

[6]Though Apple has announced plans to switch to Intel starting 2008.
[7]In this context it must be used with the REP prefix.

- Hardwired implementation, i.e. not microcoded.[8]

All but the last of the traits listed above make pipelining smoother and easier.

1.3 Beneficial Effects for Compiler Writers

Computer architects used to partly justify including instructions like **poly** in the instruction set by saying that this made the job of compiler writers easier. However, it was discovered that this was not the case, for several reasons.

- Compiler writers found it difficult to write compilers which would automatically recognize situations in (say) C source code in which instructions like **poly** could be used.
- Compiler writers found that some specialized instructions that were actually motivated by high-level language constructs did not match the latter well. For example, the VAX had a CASE instruction, obviously motivated by the Pascal **case** and C **switch** constructs; yet it turned out to be too restrictive to use well for compiling those constructs.
- CISC generally led to very nonorthogonal architectures, which made the job of compiler writers quite difficult. For instance, in Intel, the requirement that the EAX, ECX and EDI registers be used in STOS, plus the requirement that some of these registers be used in certain other instructions, implies that compiler writers "don't have any spare registers to work with," and thus they must have the compiler generate code to save and restore "popular" registers such as EAX; this is a headache for the compiler writer.

All of this inhibits development of compilers which produce very fast code.

Most RISC machines do ask that the compiler writers do additional work of other kinds, though, in that they have to try to fill **delay slots**; more on this below.

1.4 Introduction to the MIPS Architecture

MIPS is one of the most popular RISC chips, used for example in SGI workstations and in a number of embedded applications.[9]

[8]Again, we will not discuss microcode here.

[9]MIPS is also the subject of a simulator, SPIM, which is used in many university computer architecture courses. See `http://www.cs.wisc.edu/~larus/spim.html`.

Every MIPS instruction is 32 bits, i.e. one word, in length, and every instruction which does not access memory executes in one cycle. Those that do access memory are assumed to "usually" take two cycles, meaning that this will be the case if the desired memory access results in a cache hit; otherwise, the CPU enters a **stall** mode until the memory access is satisfied.

1.4.1 Register Set

There are 32 general registers, named $0, $1, ..., $31. The assembler also has an alternate set of names for the registers, which show their suggested uses; here is a partial list:

```
$0       zero      contains the hard-wired constant 0
$1-$2    v0-v1     for expression evaluation and function return values
$4-$7    a0-a3     for passing arguments
$8-$15   t0-t7     for temporary results
...
$28      gp        pointer to global area
$29      sp        stack pointer
$30      fp        frame pointer (not used)
$31      ra        return address
```

The usages listed above, e.g. $29 as the stack pointer, are merely suggested; in keeping with the RISC philosophy of orthogonality, the registers are treated uniformly by the architecture.

1.4.2 Example Code

To introduce the MIPS architecture, here is a C program which calls a MIPS assembly language subroutine, **addone()**:

TryAddOne.c:

```
int x;

main()

{  x = 7;
   addone(&x);
   printf("%d\n",x);  // should print out 8
   exit(1);
}
```

AddOneMIPS.s:

```
.text
        .globl  addone
```

```
addone:
        lw       $2,0($4)
        addu     $2,$2,1
        sw       $2,0($4)
        j        $31
```

To begin, let's look at the compiled code we obtain by applying **gcc -S** to TryAddOne.c. The source lines

```
   x = 7;
   addone(&x);
```

translate to:

```
        # x = 7;
        li      $2,7
        sw      $2,x

        # addone(&x);
        la      $4,x
        la      $25,addone
        jal     $31,$25
```

The **li** (Load Immediate) instruction puts 7 into the register $2.[10]

The **sw** (Store Word) instruction then copies what is in $2 to the memory location x.

Now, what about the call? Many RISC architectures are less stack-oriented than are the classical CISC ones. This is because stacks are in memory, and memory access is slow.[11] MIPS is an example of this philosophy. Both the argument to **addone()**, as well as the return address, will be put in registers, rather than on the stack.

The **la** (Load Address) instruction here puts the argument, the address of **x**, into register $4.[12] We then use this same instruction to put the jump target, i.e. the address of **addone()**, into $25. We then call **addone()** with the instruction

```
        jal     $31,$25
```

which says to "jump and link" (i.e. do a subroutine call) to the subroutine pointed to by $25, saving the return address (the address of the instruction following the **jal**) in $31.

[10]Actually, **li** is not a real MIPS instruction, but is instead a macro, from which the assembler will generate an **ori** instruction. More on this later.

[11]Some architectures, such as SPARC, try to solve this problem by actually having space on the CPU chip for the top few elements of the stack.

[12]Recall that registers $4-$7 are used for passing arguments. If there are more arguments, we can store them in memory, and one of the arguments can be a pointer to them.

The return from the subroutine thus makes use of $31:

```
        j        $31
```

This is an unconditional jump, to the location pointed to by $31.

Note that operands have different meanings depending on the operation. In **sw $2, x**, the **x** meant the memory location named **x**, whereas in **la $4, x**, **x** meant the address of **x**.

Recall that **main()** had placed the argument in $4. The subroutine then uses it:

```
        lw       $2,0($4)
        addu     $2,$2,1
        sw       $2,0($4)
```

The **lw** (Load Word) instruction is the opposite of **sw**, fetching a word from memory rather than storing it. Both the **lw** and **sw** here use the more general format, which is based addressing. For example,

```
        lw       $2,0($4)
```

adds $4 to 0, and then treats that as an address, fetching the contents of that address and putting it in register $2.

The **addu** (Add Unsigned) instruction has three operands; it adds the second and third, and places the sum in the first.

1.4.3 MIPS Assembler Pseudoinstructions

Recall that most assembly languages allow **macros**. MIPS assemblers go a step further, offering the programmer use of **pseudoinstructions**; these appear to be machine instructions, but are actually like macros, which the assembler translates to one or more real MIPS instructions.

Several "instructions" which we saw in the example above are actually pseudoinstructions. The first of those was **li**, allegedly the Load Immediate instruction. There actually is no such instruction, and our line containing **li** was actually treated by the assembler as

```
        ori $2, $0, 7
```

where **ori** is the MIPS' OR Immediate instruction. Remember, the register $0 contained the hardwired value 0, so the instruction above applies the logical OR operation to 0 and 7, yielding 7, and thus placing 7 into the register $2, just as desired. So, why couldn't the designers of the MIPS architecture include **li** Load Immediate instruction?

To see the answer, suppose we wish to load a large number like 0xabcd1234, which is 32 bits in length. Remember, all MIPS instructions are 32 bits long, so this operand would occupy the entire instruction, with no room for an op code!

Indeed, if our assembly language source code contains a line

```
li $2,0xabcd1234
```

the assembler will actually treat it as if it were two instructions:

```
lui $1,0xabcd
ori $2, $1, 0x1234
```

The MIPS instruction **lui** places the given immediate constant, 0xabcd in this case, and placing it in upper 16 bits of the destination, the register $1 here, and places 0s in the lower 16 bits. The MIPS instruction **ori** then puts 0x1234 in the lower 16 bits of a copy of $1, then placing the result in $2. This gets 0xabcd1234 into $2, as desired.

So, the use of pseudoinstructions here frees the programmer from having to think about the size of the operand she is using.

Similarly, in the line

```
la $4, x
```

from our example code above, **la** is just a pseudoinstruction, not a real MIPS machine instruction. One of the issues here is like the one with **li** above; the address of x is a 32-bit quantity, and thus could not be fit into one instruction. The other issue is that we need a way of letting the assembler know that "x" here means the address of x, and our use of the pseudoinstruction **la** tells the assembler that.

Finally, there is even some chicanery in the line

```
sw $2, x
```

Actually, **sw** IS a real MIPS instruction. But the real instruction has three operands, as in our other line,

```
sw $2, 0($4)
```

(the three operands are $2, 0 and $4), whereas

```
        sw $2, x
```

has only two operands. The fact that there are only two operands here is a signal to the assembler that this too is a pseudoinstruction, in spite of the fact that there is a real MIPS instruction **sw**. So, the assembler will convert this to an **lui** and **ori** and then a (real) **sw**.

1.4.4 Programs Tend to Be Longer on RISC Machines

Note in our example above that it took two instructions

```
        li      $2,7
        sw      $2,x
```

to do what, for example, on Intel would take only one:

```
movl $7,x
```

In fact, we later found that those two instructions were actually pseudoinstructions, so here MIPS is taking four instructions to do what Intel does in one.

This is typical of RISC machines, and means that RISC programs tend to be longer than CISC ones. For example, the compiled version of TryAddOne.c above[13] is 1276 bytes long on an SGI machine but only 1048 on an Intel platform.

1.4.5 Instruction Formats

One very “RISC-y” feature of MIPS is that it has only three instruction formats.[14] To present them, let us number the most- and least-significant bits in a word as 31 and 0, respectively. Also, we will use the following abbreviations:

```
op         op code
--         -------
rs         register operand
rt         register operand
rd         register operand
imm        immediate constant
shamt      shift amount
funct      special function (like an additional op code)
dist       distance factor to branch (i.e. jump) target
```

[13]This is TryAddOne.o, generated from applying **gcc -c** to TryAddOne.c.

[14]By contrast, the Intel chip, a CISC, has a large number of instruction formats, and even the RISC SPARC chip has six.

the formats are as follows.

- I ("immediate") format:
  - op: bits 31-26
  - rs (source): bits 25-21
  - rt (destination): bits 20-16
  - imm: bits 15-0
- J ("jump") format:
  - op: bits 31-26
  - dist: bits 25-0
- R ("register") format:
  - op: bits 31-26
  - rs (source): bits 25-21
  - rt (source): bits 20-16
  - rd (destination): bits 15-11
  - shamt: bits 10-6
  - funct: bits 5-0

The **shamt** field is used if the instruction is to perform a right- or left-shift; the shift will be **shamt** bits in the specified direction.

The **dist** field in J instructions is somewhat unusual, but actually clever: It is equal to 1/4 of the distance to the branch target.[15] Since all instructions are 4 bytes long, then every instruction address is a multiple of 4, and thus the distance from any branch instruction to its target is a multiple of 4, i.e. the distance has two 0 bits at the right end. That last point means that there is no point in storing the last two bits of the distance; they are simply tacked on by the CPU when the instruction is executed. This effectively multiplies the jump range by 4.

The J format is used only for unconditional branches, including **call**. Conditional branches use the I or R format.

Note that the **imm** field can have two interpretations, depending on **op**. The **addi** instruction, for example, treats the immediate constant as a signed number, while **addiu** treats it as unsigned. Since MIPS has word

[15]Recall that *branch* is a synonym for *jump*.

size 32 bits, its ALU deals with 32-bit quantities, not the 16-bit quantity in an **imm** field. Thus the latter must be extended to 32 bits before entering the ALU, by copying the leftmost bit to the leading 16 bits. Thus for instance in the instructions

```
addi $6, $7, 0xb123
addiu $9, $10, 0xb123
```

in the first case the 0xb123 becomes 0xffffb123 while in the second case it becomes 0x0000b123.

1.4.6 Arithmetic and Logic Instruction Set

Here is a partial listing of the MIPS instruction set for arithmetic and logic operations:

```
add rd, rs, rt              # rd <-- rs + rt
addu rd, rs, rt             # rd <-- rs + rt (without overflow)
addi rd, rs, imm            # rd <-- rs + imm

and rd, rs, rt              # rd <-- rs AND rt
ani rd, rs, imm             # rd <-- rs AND imm

not rd, rs                  # rs <-- NOT rd

or rd, rs, rt               # rd <-- rs OR rt
ori rd, rs, imm             # rd <-- rs OR imm

sll rd, rt, shamt           # rd <-- rt << shamt
srl rd, rt, shamt           # rd <-- rt >> shamt

sub rd, rs, rt              # rd <-- rs - rt
subu rd, rs, rt             # rd <-- rs - rt (without overflow)
```

1.4.7 Conditional Branches in MIPS

On a MIPS machine, applying **gcc -S** to

```
   if (i == 20) j = 2;
```

produces

```
        lw      $2,i
        li      $3,20
        bne     $2,$3,.L4
        li      $2,2
        sw      $2,j
```

where .L4 is a label of the branch target. The **bne** instruction compares the two registers and then branches to the target if they are not equal. (There is nothing like Intel's Z and S Flags in MIPS.)

1.5 Some MIPS Op Codes

```
ORI    001101
LUI    001111
SW     101011
JAL    000011
J      000010
BNE    000101
ADDI   001000
ADDU   001001
```

1.6 Dealing with Branch Delays

RISC architectures are especially good at having smooth pipeline operation, due to the uniformity of instruction execution time: Every instruction executes in one clock cycle, except for branches and load/store instructions, which access memory and thus take longer. Even to go to the cache takes up an extra clock cycle, and going to full memory takes even longer. Thus the pipeline is delayed.

For example, consider this MIPS code:

```
        subiu $2, $2, $1
        bne    $4, $3, yyy
        addiu $5, $5 12
```

While the fetch/decode/execute cycle for the **bne** is taking place, the CPU is prefetching the **addiu**. But if the branch is taken, the instruction at yyy will be the one we need, and it will not have been prefetched.

1.6.1 Branch Prediction

One possible way to deal with the branch-delay problem is to design the hardware to try to predict whether a conditional branch will be taken or not. The hardware will prefetch from the instruction which it predicts will be executed next. If it is correct, there is no delay; if the prediction is false, we must fetch the other instruction, incurring the delay.

Even non-RISC architectures do this. Intel CPUs, for example do the following when a given conditional branch instruction is executed for the first time: If it is a branch backward, the prediction is that the branch will be taken; this design is motivated by the assumption that backward branches will typically be the last instruction within a loop, and usually such branches will be taken. If the branch is forward, the prediction is that it will not be taken. In subsequent times this particular branch instruction is executed, a complicated prediction algorithm is applied, based on previous history.

1.6.2 Delayed Branch

MIPS—as well as most RISC chips—solves this problem by using something called **delayed branch**. The CPU actually executes the instruction which sequentially follows the branch instruction in memory, in this case the **addiu**, before executing the instruction at the branch target. That way useful work is being done during the delay involving fetching of the branch target.

But wait! This can't work as it stands. After all, if the branch is taken, we do not want to execute that **addiu**! So, the assembler (or compiler, in the case of C/C++) must put in a "do nothing" instruction: **nop**, pronounced "no-op" for "no operation."[16]

```
subiu $2, $2, $1
bne    $4, $3, yyy
nop
addiu $5, $5 12
```

But now we are back to the original problem — no useful work is being done while the instruction at yyy is being fetched. So, next an optimizer within the assembler (or compiler) will try to actually rearrange the instructions. It will look for some instruction elsewhere in the code which can be moved into the slot currently occupied by the **nop**. In this case, the **subiu** can be safely moved:

```
bne $4, $3, yyy
subiu $2, $2, $1
addiu $5, $5, 12
```

The reasoning is this: The **bne** is not affected by the outcome of the **subiu** which precedes it (this would not be true if the **bne** compared $4 with $2, for example), so it doesn't matter if we move the **subiu**. Remember, the latter will still be executed, because the CPU is designed to execute the instruction which immediately follows the branch in memory. ✓

The setup for load/store instructions is similar. The MIPS CPU will execute the instruction which immediately follows a load/store instruction in memory while performing the load/store. This is done because the latter requires a time-consuming memory access.

But as with branches, we need to watch out for dependencies. For example, consider

```
lw $2, 0($4)
addiu $2, $2, 1
```

The CPU would execute the add simultaneously with fetching register $2, with potentially disastrous consequences. Thus the assembler will at first change this to

[16]Almost every CPU architecture has this kind of instruction. On MIPS, its op code is all 0s.

```
lw $2, 0($4)
nop
addiu $2, $2, 1
```

and then the optimizer will try to find some instruction elsewhere to move to the position now occupied by the **nop**.

The instruction position following a branch or load/store is called a **delay slot**. Machines such as MIPS and SPARC have one delay slot, but some other machines have more than one. Clearly, the more delay slots there are, the harder it is to "fill" them, i.e. to find instructions to move to put in the slots, and thus avoid simply putting wasteful NOPs in them.

How hard is it to fill even one delay slot? The Katevenis dissertation on RISC says:

> Measurements have shown [citation given] that the [compiler] optimizer is able to remove about 90% of the no-ops following unconditional transfers and 40% to 60% of those following conditional branches. The unconditional and conditional transfer-instructions each represent approximately 10% of all executed instructions (20% total). Thus, while a conventional pipeline would lose $\approx$ 20% of the cycles, optimized RISC code only loses about 6% of them.

Chapter 2

The Java Virtual Machine

Note: Thomas Fifield, a student in ECS 50 a few years ago, wrote portions of Section 2.7 (excluding the source code for **NumNode.java**), and added the corresponding new instructions to Section 2.8.

2.1 Background Needed

In what follows, it is assumed that the reader has at least a rudimentary knowledge of Java. See the author's Java tutorial at `http://heather.cs.ucdavis.edu/~matloff/java.html` for a 30-minute introduction to the Java language. But really, for our purposes, anyone who understands C++ should have no trouble following the Java code here.

2.2 Goal

Keep in mind that our goal here is to study the Java (virtual) *machine*, not the Java language. We wish to see how the machine works, and—this is very important—see why its developers chose to design certain features of the machine the way they did.

2.3 Why Is It a "Virtual" Machine?

You may have heard about the **Java virtual machine** (JVM), associated with the Java language. What is really going on here?

The name of any Java source file has a **.java** suffix, just like C source file names end in **.c**. Suppose for example our Java program source code is all in one file, **x.java**, and that for instance we are doing our work on a PC running Linux.

We first compile, using the Java compiler, **javac**:

```
% javac -g x.java
```

(The **-g** option saves the symbol table for use by a debugger.)

This produces the executable Java file, **x.class**, which contains machine language, called **byte code**, to run on a "Java machine." But we don't have such a machine.

Instead, we have a program that emulates the operation of such a machine. This, of course, is the reason for the 'V' in "JVM." The emulator (**interpreter**) program is named **java** . Note that in our case here, **java** will be a program running the Intel machine language of our PC.[1]

We then run our program:

```
% java x
```

Note that Java not only runs on a virtual machine, it also is in some sense running under a virtual operating system. For example, the analog of C's **printf()** function in Java is **System.out.println()**. Recall that (if our real machine is running UNIX) a **printf()** call in a C program calls the **write()** function in the OS. But this does not happen in the Java case; our OS is running on our real machine, but our Java program is running on the JVM. What actually happens is that **System.out.println()** makes a call to what amounts to an OS in the JVM, and the latter calls **write()** on the real machine.

By the way, JVM chips—i.e. chips that run Java byte code—do exist, but they are not in common use. Also, note that newer versions of GCC include GCJ, which will compile Java to machine code for your machine, e.g. Intel machine language if you are on a PC. This is really nice, as it produces faster-running programs, and it is far more common than Java chips, but still, the vast majority of Java code is executed on emulators.

2.4 The JVM Architecture

The JVM is basically a stack-based machine, with a 32-bit word size, using 2s complement arithmetic.

The term *stack-based machine* means that almost all operands are on the stack, and they are implicit to the instruction. For example, contrast the Intel instruction

```
addl %eax, %ebx
```

[1]And, for that matter, **javac** would be an Intel machine-language program too.

to the JVM instruction

```
iadd
```

In the Intel case, the programmer has a choice of which two things to add together, and explicitly states them. In the JVM case, the programmer has no choice; the two operands must be the top two elements of the stack. And since those operands must be the top two elements of the stack, there is no need to specify them in the instruction, so the instruction consists only of an op code, no explicit operands.[2]

Before continuing, it's vital that we discuss what the term "the" stack means above. Unlike the case of C/C++/assembly language, where there is a stack for each *process*, here we will have a stack for each *method call*.[3] So, the term "the stack" in the material here must always be thought of as meaning "the stack for whatever method is currently running." This is referred to as the **operand stack**.

On the other hand, there *is* something analogous to the stack you have worked with before in C/C++/assembly language. Recall that in that situation, all the function calls use a common stack, and all the data for a given function call—its arguments, return value and local variables—comprise the **stack frame** for that function call. Java has this too. There is a stack frame for each method call, and they are placed in the overall stack in the same last-in, first-out fashion that use saw for function calls in C/C++/assembly language. Each stack frame in Java will in turn consist of the operand stack for that method call, a section for arguments and local variables, and some other data.

2.4.1 Registers

The JVM register set[4] is fairly small:

- **pc**: program counter
- **optop**: pointer to the top of the operand stack for the currently-active method
- **frame**: pointer to the stack frame of the currently-active method
- **vars**: pointer to the beginning of the local variables of the currently-active method

[2]Historical note: A number of famous machines popular in the past, such as the Burroughs mainframe and HP 3000 minicomputer series, were stack-based.

[3]Recall that in Java, we use the term *method* instead of *function*.

[4]Remember, in the case of a real Java chip, these would be real registers, but in the JVM setting, these registers, as well as the instruction set, are simulated by the **java** program.

2.4.2 Memory Areas

- the (general) Stack:

  A method call produces a new stack frame, which is pushed onto the Stack for that program,[5] and a return pops the frame from the program's Stack.

  A stack frame is subdivided into the following.

  - a Local Variables section:

    All the local variables and arguments for the method are stored here, one per **slot** (i.e. one variable per word), with the arguments stored first and then the locals. The arguments and locals are stored in order of their declaration. In the case in which the method is an instance method,[6] slot 0 will contain a pointer to the object on which this method is operating, i.e. the object referred to as **this** in the program's source code.
  - an Operand Stack section:

    This is the area on which the method's instructions operate. As noted, almost all JVM instructions are stack-based; e.g. an "add" instruction pops the top two elements of the stack, adds them, and pushes the sum back onto the stack. So the operand stack portion of a method's stack frame is what we are referring to when we refer to an instruction as operating on "the" stack.
  - a Frame Data section:

    We will not go into the details of this, but for example the Frame Data section of a called method's stack frame will include a pointer to the caller's stack frame. This enables return to the caller when the called method finishes, and enables the latter to put the return value, if any, into the caller's stack frame.

- the Method Area:

  The classes used by the executing program are stored here. This includes:

  - the bytecode and access types of the methods of the class (similar to the **.text** segment of a UNIX program, except for the access types)
  - the **static** variables of the class (their values and access types, i.e **public**, **private** etc.)

  The **pc** register points to the location within the Method Area of the JVM instruction to be executed next.

  The Method Area also includes the Constant Pool. It contains the string and numeric literals used by the program, e.g. the 1.2 in

[5]More precisely, for that thread, since many Java programs are threaded.

[6]In general object-oriented programming terminology, a function is called an **instance** function if it applies specifically to an instance of a class. In C++ and Java, this is signified by not being declared **static**. The alternate form is **class** methods, which apply to all objects of that class, and is signified in C++/Java by being declared **static**.

```
float W = 1.2;
```

and also contains information on where each method is stored in the Method Area, as well as the static variables for the various classes.

- the Heap:

  This is where Java objects exist. Whenever the Java **new** operation is invoked to create an object, the necessary memory for the object is allocated within the heap.[7] This space will hold the instance variables for the object, and a pointer to the location of the object's class in the Method Area.

2.5 First Example

Consider the following source code, **Minimum.java**:

```
public class Minimum {

   public static void main(String[] CLArgs)

   {  int X,Y,Z;

      X = Integer.parseInt(CLArgs[0]);
      Y = Integer.parseInt(CLArgs[1]);
      Z = Min(X,Y);
      System.out.println(Z);
   }

   public static int Min(int U, int V)

   {  int T;

      if (U < V) T = U;
      else T = V;
      return T;
   }
}
```

2.5.1 Java Considerations

In Java, there are no global quantities of any kind. That not only means no global variables, but also no free-standing, functions—every function must be part of some class.[8] So for example we see here that even **main()** is part of a class, the class **Minimum**. (That class in turn must be named after the name of the source file, **Minimum.java**.)

[7]Just as in C, where a call to **malloc()** results in memory space being allocated from the heap, and just as in C++, where a call to **new** results in space being taken from the heap.

[8]Java threads from the same class will share the variables of that class.

The argument for **main()**, **CLArgs**, is the set of command-line arguments, i.e. what we normally name **argv** in C/C++.[9] There is nothing like **argc**, though. The reason is that **CLArgs** is of type **String []**, i.e. an array of strings, and in Java even arrays are objects. Array objects include a member variable showing the length of the array, so information analogous to **argc** is incorporated in **CLArgs**, specifically in **CLArgs.length**—the value of the **length** member variable of the array class, for the instance **CLArgs** of this class.

The function **Integer.parseInt()** is similar to **atoi()** in C/C++. Note, though, since Java does not allow free-standing functions, **parseInt()** must be part of a class, and it is—the **Integer** class, which has lots of other functions as well.

We use the compiler, **javac**, to produce the class file, **Minimum.class**. The latter is what is executed, when we run the Java interpreter, **java**.

2.5.2 How to Inspect the JVM Code

We can use another program, **javap**, to disassemble the contents of **Minimum.class**:[10]

```
% javap -c Minimum
Compiled from Minimum.java
public class Minimum extends java.lang.Object {
    public Minimum();
    public static void main(java.lang.String[]);
    public static int Min(int, int);
}

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

Method void main(java.lang.String[])
   0 aload_0
   1 iconst_0
   2 aaload
   3 invokestatic #2 <Method int parseInt(java.lang.String)>
   6 istore_1
   7 aload_0
   8 iconst_1
   9 aaload
  10 invokestatic #2 <Method int parseInt(java.lang.String)>
  13 istore_2
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Min(int, int)>
  19 istore_3
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
```

[9]Note that in C/C++ we can name that parameter whatever we want, though it is customary to call it **argv**.

[10]So, this listing here is similar to the output of **gcc -S** in a C/C++ context.

```
  24 invokevirtual #5 <Method void println(int)>
  27 return

Method int Min(int, int)
   0 iload_0
   1 iload_1
   2 if_icmpge 10
   5 iload_0
   6 istore_2
   7 goto 12
  10 iload_1
  11 istore_2
  12 iload_2
  13 ireturn
```

Note that each "line number" is actually an offset, i.e. the distance in bytes of the given instruction from the beginning of the given method.

DON'T TRY TO UNDERSTAND THE ABOVE ASSEMBLY CODE YET. We'll do this one part at a time, starting with the call to **Min()** in the next section.

2.5.3 The Local Variables Section of main()

Here is what the Local Variables Section of **main()**'s stack frame looks like:

slot	variable
0	pointer to CLArgs
1	X
2	Y
3	Z

2.5.4 The Call of Min() from main()

Now consider the call to **Min()**. The code

```
      Z = Min(X,Y);
```

gets compiled to

```
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Min(int, int)>
  19 istore_3
```

As in a classical architecture, the arguments for a call are pushed onto **main()**'s operand stack, as follows. The **iload_1** ("integer load") instruction in offset 14 of **main()** pushes slot 1 to the operand stack. Since slot 1 in **main()** contains X, this means that X will be pushed onto **main()**'s operand stack. The instruction in offset 15 will then push Y.[11]

The **invokestatic** instruction is the function call. We see here the **Min()** is function number 3. By looking up information which is stored regarding **Min()**, this instruction knows that **Min()** has two arguments.[12] Thus the instruction knows that it must pop two values from **main()**'s operand stack, which are the arguments (placed there earlier by **iload_1** and **iload_2**), and it places them in the first two slots of **Min()**'s Local Variables Area.

Note that in the instruction **iload_1**, for example, the '1' is a part of the instruction name, not an operand. In fact, none of **iload_0**, **iload_1**, **iload_2** and **iload_3** has an operand; there is a separate instruction, i.e. separate op code, for each slot.

But there is no **iload_5**, **iload_6** and so on. What if you need to load from some other slot, say slot 12? Then you would use the **iload** instruction, which is fully general, i.e. can load from any slot, including but not limited to slots 0, 1, 2 and 3. Note that **iload_0** etc. are one-byte instructions, while **iload** is a two-byte instruction, with the second byte being the slot number, so we should use those special instructions for the cases of slots 0, 1, 2 and 3. The designers of the JVM knew that most accesses to local variables would be to slots 0-3, so they decided to include special one-byte instructions to load from those slots, thus making for smaller code. Similar statements hold for **istore**, etc.

2.5.5 Accessing Arguments from within Min()

Now, upon entry to **Min()**, here is what that function's Local Variables Section will look like:

slot	variable
0	U
1	V
2	T

Since the arguments **U** and **V** are in slots 0 and 1, they will be accessed with instructions like **iload_0** and **iload_1**.

The main new instruction in **Min()** is **if_icmpge** ("if integer compare greater-than-or-equal") in offset 2. Let's refer to the top element of the current (i.e. **Min()**'s) operand stack as *op2* and the next-to-top element as *op1*. The instruction pops these two elements off the operand stack, compares them, and then jumps to the branch target if $op1 \geq op2$. Again, keep in mind that these items on **Min()**'s operand stack were placed there by the **iload_0** and **iload_1** instructions.

[11] So, the arguments must be pushed in the order in which they appear in the call, unlike C/C++.

[12] Again unlike C/C++, where the number of arguments in the call is unknown to the compiler.

Min()'s **ireturn** instruction then pops the current (i.e. **Min()**'s) operand stack and pushes the popped value on top of the caller's (i.e. **main**()'s operand stack. In the case of methods which do not have return values, we use the **return** instruction instead of **ireturn**.

2.5.6 Details on the Action of Jumps

For jump instructions, the branch target is specified as the distance from the current instruction to the target. As can be seen in the JVM assembler code above, the target for the **if_icmpge** instruction in offset 2 of **Min()** is offset 10 (an **iload_1** instruction). Since our **if_icmpge** instruction is in offset 2, the distance will be 8, i.e. 0x0008.[13] Those latter two bytes comprise the second and third bytes of the instruction. Note that the branch target must be within the current method; JVM will announce a runtime error if not.

From the Sun JVM specifications (see below), we know that the op code for the **if_icmpge** instruction is 0xa2. Thus the entire instruction should be a2 00 08, and this string of three bytes should appear in the file **Minimum.class**. Running the command

```
% od -t x1 Minimum.class
```

on a UNIX machine, we see that a2 00 08 is indeed in that file, in bytes 1255-1257 octal (685-687 decimal).

2.5.7 Multibyte Numbers Embedded in Instructions

Note in the jump example above, the instruction was a0 00 08, with a0 being the op code and 00 08 being the distance to the jump target. Note that the order of bytes in memory will be that seen above, i.e. a0 00 08, so 00 will be in a lower-address byte than 08. So, the question arises as to whether the embedded constant is intended to be 0008, i.e. 8, or 8000. The answer is the former.

In other words, numbers embedded within instructions are meant to be interpreted as big-endian.

2.5.8 Calling Instance Methods

When an instance method is called, we use the **invokevirtual** instruction instead of **invokestatic**. To understand how that works. recall that in general OOP programming, an instance method includes an extra "unofficial" argument in the form of a pointer to the instance on which the method is invoked, i.e. the famous **this** pointer in C++ (and Java).

Concretely, suppose a class **C** includes an instance member method **M()**, say with two arguments. Then the call

[13]Unlike the Intel (and typical) case, the jump distance in JVM is calculated from the jump instruction, not from the one following it.

```
M(A,B)
```

from within **C** would in essence have three arguments—the two explicit ones, **A** and **B**, and also implicitly, a pointer to the current instance of **C**. As with any stack-based calling system, e.g. C++ running on an Intel machine, that extra parameter must be pushed onto the stack too, just like the explicit ones. This is also done in the case of the JVM.

Thus we can't use **invokestatic** in the JVM case, because we must account for that "extra" argument. The **invokestatic** instruction, as explained earlier, pops the arguments off the caller's stack and places them into the callee's Local Variables Section. That's not good enough in the case of an instance method, because that "extra" argument must be popped and placed into the callee's Local Variables Section too. So, the designers of the JVM also included an **invokevirtual** instruction to do this; it transfers *all* the arguments from the caller's stack to the callee's Local Variables Section, both the explicit arguments and the "extra" one.

So, from the callee's point of view, the item in its slot 0 will be **this**, the pointer to the current object, and in fact the compiler will translate references to **this** to accesses to slot 0.

The only example we have so far of a call made via **invokevirtual** is in the compiled code for our source line in **main()**,

```
System.out.println(Z);
```

The compiled code is

```
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
  24 invokevirtual #5 <Method void println(int)>
```

That 3 in offset 23 refers to slot 3, the location of our local variable **Z**, our argument to **println()**.

The code

```
System.out.println(Z);
```

calls the **println()** method on the object **System.out**. The official Java documentation indicates that the latter is an instance of the class **java.io.PrintStream**, which in turn is a subclass of the class **java.io.FilterOutputStream**. That latter class includes a member variable **out**, which is basically a pointer to the file we are writing to. In this case, we are writing to the screen (in UNIX parlance, **stdout**).

With that in mind, look at the instruction in offset 20. From our discussion above, you can see that this instruction must be placing a pointer to the object **System.out** on the stack, and that is exactly what the **getstatic** instruction is doing. It is getting a class (i.e. **static**) variable from the given object, and pushing it onto the stack.

2.5.9 Creating and Accessing Arrays

Arguments and local variables which are arrays are maintained as addresses within the Local Variables Area. Array read accesses work by pushing the address of the array and the desired index onto the operand stack, and then executing one of the special array instructions. Array write accesses are done the same way, but with an additional push for the value to be stored.

There are also special array instructions which create the array storage itself when **new** is invoked.

Consider the following example:

```
public int gy(int x)
{  int y[],z;
   y = new int[5];
   z = x + 2;
   y[3] = x;
   return 0;
}
```

(It does nothing of interest, but will serve as a convenient simple example.)

That code is translated to:

```
Method int gy(int)
   0 iconst_5
   1 newarray int
   3 astore_2
   4 iload_1
   5 iconst_2
   6 iadd
   7 istore_3
   8 aload_2
   9 iconst_3
  10 iload_1
  11 iastore
  12 iconst_0
  13 ireturn
```

Here you see the **newarray** instruction is used to create space for the array, specifically enough for 5 **int**s; the 5 was pushed onto the operand stack by the **iconst_5** ("integer constant") instruction.

The assignment to **y[3]** is done by the code in offsets 7-11. Make sure you see how this works. In particular, the **iastore** ("integer array store") instruction first pops the stack to determine which value to store, which is slot 1, i.e. **x**. (Note that lacking the **static** keyword, this is an instance function.) It then pops again to determine which array element to store to (index 3), and finally pops once more to determine which array to use (slot 2, i.e. **y**).

In the code for **main()** in our **Min** example above, **CLArgs** is an array of strings, thus an array of arrays, just as in C/C++. The **aload_0** ("address load") instruction pushes the address in slot 0 i.e. the address of **CLArgs**, onto the current operand stack. Similarly, **iconst_0** ("integer constant") pushes the constant 0. All this is preparation for accessing the array element **CLArgs[0]**. However, since that element is a string, thus an address again, we do not use **iastore** as before, instead using the **aastore** instruction ("address array store").

2.5.10 Constructors

When the constructor for a new object is called, we need to execute a **invokespecial** instruction.

2.5.11 Philosophy Behind the Design of the JVM

Remember, we are here to learn about the Java machine, not the Java language. One aspect of that concerns why the designers of the JVM made certain choices as to the structure of the machine.

Before we continue, note that the JVM *is* a machine, just like Intel, MIPS or the others. While it's true that we most often just use an emulator program, JVM chips have indeed been built. So for example one could certainly write a C compiler for the machine, i.e. write a compiler that translates C source code to JVM machine code.

2.5.11.1 Instruction Structure

Most instructions are a single byte in length, but there are a few multi-byte instructions as well. In any case, the first byte is the op code, and any operands are in the bytes that follow. As mentioned earlier, almost all JVM instructions involve stack operations, so there are no explicit operands, which is why most instructions are a single byte long. This has other implications, explained below.

2.5.11.2 Stack Architecture

You may wonder why the developers of the JVM chose the stack-based architecture. Actually, the literature is not clear about this.

One claim is that the choice of this architecture was made in order to make it easy to write JVM interpreters for native machines that had few or no registers. If for example the developers had chosen to have the JVM have 32 registers, then on machines with fewer registers the authors of Java interpreters would not be able to model the JVM virtual registers by the real registers of the native machines. Instead, the JVM registers would have to be modeled by variables in memory. This does not sound like a very convincing argument. For one thing, it makes it more difficult for authors of interpreters on machines that do have a lot of registers to write fast interpreters.

Another claim is that by having most instructions only a single byte in length, the overall program size is smaller. This would be an important consideration if it were true. Java code is often transported over the network. This happens behind the scenes when you access a Java-powered Web page. The Web site will download Java code for some operation to your machine (the one where you are running the browser), and that code will be executed by the Java interpreter on your machine.

So smaller code would mean less network download delay. But is Java code smaller? Recall that RISC instructions are also very simple, but that that actually tends to make code size larger than on CISC machines, since each instruction does less work. Since the stack architecture means a lot of pushes and pops in order to manoevre data items for arithmetic, it may well be the case that code size is larger for the JVM.

However,[14] the JVM situation is very different from the RISC one, because with the former we are worried about conserving network bandwidth. JVM byte code is likely to be highly compressable—certain instructions like, say, **iload_0** are probably quite frequent—so that download times could be made quite low.

2.5.11.3 Safety

Note that there are separate instructions **istore_0** and **astore_0**. Each pops the operand stack and places the popped value into slot 0. But the first assumes the popped value is an integer, while the second assumes an address.[15] In other words, unlike "normal" machines, in which the hardware is unaware of types, the JVM is quite aware and proactive. The JVM will check types (which are stored with the value), and a runtime error will occur if, for example, one tries to execute **astore_0** when the top of the operand stack contains an integer type.

Java does allow **type coercion** as in C/C++. Consider for instance the code

```
int I = 12; float X;
...
X = I + (float) 3.8;
```

Here the variable **I** is converted to type **float** in the compiled code, using the JVM's **i2f** instruction.

Getting back to the safety issue, one concern is that Java code might be "hacked" during its transport over the network. There are various mechanisms within the JVM to check that the code is the same as when it was sent; none of these is perfect, of course, but it does provide some protection.

2.6 Another Example

This one finds the row in a matrix which has minimum sum.

[14]As pointed out by student Daniel Wolfe.

[15]Formally called a **reference**.

```
public class Min2 {

   public static int ABC[][];  // elements will be within [0,1000000]

   public static void main(String[] CLArgs)

   {  int I,J;

      ABC = new int[10][10];
      // fill it with something, just an example
      for (I = 0; I < 10; I++)
         for (J = 0; J < 10; J++)
            ABC[I][J] = 2 * I + J;
      System.out.println(Min());
   }

   public static int Min()

   {  int Row,Col,MinSoFar,Sum;

      MinSoFar = 1000000;  // note restrictions on ABC above
      for (Row = 0; Row < 10; Row++)  {
         Sum = 0;
         for (Col = 0; Col < 10; Col++)  {
            Sum += ABC[Row][Col];
            if (Sum > MinSoFar) break;
         }
         if (Sum < MinSoFar)
            MinSoFar = Sum;
      }
      return MinSoFar;
   }
}
```

```
Compiled from Min2.java
public class Min2 extends java.lang.Object {
    public static int ABC[][];
    public Min2();
    public static void main(java.lang.String[]);
    public static int Min();
}

Method Min2()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 bipush 10
   2 bipush 10
   4 multianewarray #2 dim #2 <Class [[I>
   8 putstatic #3 <Field int ABC[][]>
  11 iconst_0
  12 istore_1
  13 goto 45
  16 iconst_0
  17 istore_2
  18 goto 36
  21 getstatic #3 <Field int ABC[][]>
  24 iload_1
  25 aaload
  26 iload_2
  27 iconst_2
  28 iload_1
  29 imul
  30 iload_2
  31 iadd
  32 iastore
  33 iinc 2 1
  36 iload_2
  37 bipush 10
  39 if_icmplt 21
  42 iinc 1 1
  45 iload_1
  46 bipush 10
  48 if_icmplt 16
  51 getstatic #4 <Field java.io.PrintStream out>
  54 invokestatic #5 <Method int Min()>
  57 invokevirtual #6 <Method void println(int)>
  60 return

Method int Min()
   0 ldc #7 <Integer 1000000>
   2 istore_2
   3 iconst_0
   4 istore_0
   5 goto 52
   8 iconst_0
   9 istore_3
  10 iconst_0
  11 istore_1
  12 goto 36
  15 iload_3
  16 getstatic #3 <Field int ABC[][]>
  19 iload_0
  20 aaload
  21 iload_1
  22 iaload
  23 iadd
  24 istore_3
  25 iload_3
  26 iload_2
  27 if_icmple 33
  30 goto 42
  33 iinc 1 1
  36 iload_1
  37 bipush 10
  39 if_icmplt 15
  42 iload_3
  43 iload_2
  44 if_icmpge 49
  47 iload_3
  48 istore_2
  49 iinc 0 1
  52 iload_0
  53 bipush 10
  55 if_icmplt 8
  58 iload_2
  59 ireturn
```

So, for example, look at 13 of the source code,

```
ABC[I][J] = 2 * I + J;
```

This single Java statement compiles to a remarkable 10 JVM machine instructions! They are in offsets 21-32 of **main()**. Below is an overview of what they do:

```
21   push address of ABC
24   push I
25   pop I, address of ABC; push address of ABC[I]
26   push J
27   push 2
28   push I
29   pop I, 2; push 2*I
30   push J
31   pop J, 2*I; push 2*I+J
32   pop 2*I+J, J, address of ABC[I]; do ABC[I][J]=2*I+J
```

As you read this, recall that a two-dimensional array is considered an array of arrays. For example, row **I** of **ABC**, i.e. **ABC[I]**, is an array. Recall also that an array name, when used without a subscript, is the address of the beginning of that array. Here **ABC[I]**, considered as an array, has no subscript, while for instance **ABC[I][J]** has the subscript **J**. So, **ABC[I]** is an address. Thus **ABC** is an array of addresses! Hence the use of **aaload** in offset 25.

The **multianewarray** instruction sets up space for the array, pushing the address on the stack. The **putstatic** instruction then pops that address off the stack, and places it in the entry for this **static** variable in the Method Area.

2.7 Yet Another Example

This example builds a linked list of integers:

```
// this class is in the file NumNode.java

public class NumNode

{  private static NumNode Nodes = null;
   // valued stored in this node
   int Value;
   // "pointer" to next item in list
   NumNode Next;

   public NumNode(int V)  {
      Value = V;
      Next = null;
   }

   public static NumNode Head()  {
         return Nodes;
   }

   public void Insert()  {
      if (Nodes == null)  {
         Nodes = this;
         return;
      }
      if (Value < Nodes.Value)  {
         Next = Nodes;
         Nodes = this;
         return;
      }
      else if (Nodes.Next == null)  {
         Nodes.Next = this;
         return;
      }
      for (NumNode N = Nodes; N.Next != null; N = N.Next)  {
         if (Value < N.Next.Value)  {
            Next = N.Next;
            N.Next = this;
            return;
         }
         else if (N.Next.Next == null)  {
            N.Next.Next = this;
            return;
         }
      }
   }

   public static void PrintList()  {
      if (Nodes == null) return;
      for (NumNode N = Nodes; N != null; N = N.Next)
         System.out.println(N.Value);
   }

}
```

```
Compiled from "NumNode.java"
public class NumNode extends java.lang.Object{
int Value;

NumNode Next;

public NumNode(int);
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   iload_1
   6:   putfield        #2; //Field Value:I
   9:   aload_0
   10:  aconst_null
   11:  putfield        #3; //Field Next:LNumNode;
   14:  return

public static NumNode Head();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   areturn

public void Insert();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       11
   6:   aload_0
   7:   putstatic       #4; //Field Nodes:LNumNode;
   10:  return
   11:  aload_0
   12:  getfield        #2; //Field Value:I
   15:  getstatic       #4; //Field Nodes:LNumNode;
   18:  getfield        #2; //Field Value:I
   21:  if_icmpge       36
   24:  aload_0
   25:  getstatic       #4; //Field Nodes:LNumNode;
   28:  putfield        #3; //Field Next:LNumNode;
   31:  aload_0
   32:  putstatic       #4; //Field Nodes:LNumNode;
   35:  return
   36:  getstatic       #4; //Field Nodes:LNumNode;
   39:  getfield        #3; //Field Next:LNumNode;
   42:  ifnonnull       53
   45:  getstatic       #4; //Field Nodes:LNumNode;
   48:  aload_0
   49:  putfield        #3; //Field Next:LNumNode;
   52:  return
   53:  getstatic       #4; //Field Nodes:LNumNode;
   56:  astore_1
   57:  aload_1
   58:  getfield        #3; //Field Next:LNumNode;
   61:  ifnull  119
   64:  aload_0
   65:  getfield        #2; //Field Value:I
   68:  aload_1
   69:  getfield        #3; //Field Next:LNumNode;
   72:  getfield        #2; //Field Value:I
   75:  if_icmpge       92
   78:  aload_0
   79:  aload_1
   80:  getfield        #3; //Field Next:LNumNode;
   83:  putfield        #3; //Field Next:LNumNode;
   86:  aload_1
   87:  aload_0
   88:  putfield        #3; //Field Next:LNumNode;
   91:  return
   92:  aload_1
   93:  getfield        #3; //Field Next:LNumNode;
   96:  getfield        #3; //Field Next:LNumNode;
   99:  ifnonnull       111
   102: aload_1
   103: getfield        #3; //Field Next:LNumNode;
   106: aload_0
   107: putfield        #3; //Field Next:LNumNode;
   110: return
   111: aload_1
   112: getfield        #3; //Field Next:LNumNode;
   115: astore_1
   116: goto    57
   119: return

public static void PrintList();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       7
   6:   return
   7:   getstatic       #4; //Field Nodes:LNumNode;
   10:  astore_0
   11:  aload_0
   12:  ifnull  33
   15:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   18:  aload_0
   19:  getfield        #2; //Field Value:I
   22:  invokevirtual   #6; //Method java/io/PrintStream.println:(I)V
   25:  aload_0
   26:  getfield        #3; //Field Next:LNumNode;
   29:  astore_0
   30:  goto    11
   33:  return

static {};
  Code:
   0:   aconst_null
   1:   putstatic       #4; //Field Nodes:LNumNode;
   4:   return

}
```

You may notice some new instructions in this example. There are actually only six of them—**aconst_null**, **ifnull**, **ifnonnull**, **getfield**, **putfield**, and **areturn**.

The first three of these all deal with the value `NULL`, the value for null pointers. The instruction **aconst_null** is very simple. All it does is push the value of `NULL` onto the stack, just like **iconst_4** would push the value of 4 onto the stack.[16]

The two instructions **ifnull** and **ifnonnull** are conditional jumps, just like **if_cmpeq**. However, instead of comparing the first item on the stack to the second one, they jump if the value on the top of the stack is a `NULL` value (for **ifnull**) or not `NULL` (for **ifnonull**).

[16]Recall that the ‘a’ means “address,” while ‘i’ means “integer.”

Let's look at one of these new conditional jumps in action. Here's the first line of the function **PrintList()**:

```
if (Nodes == null) return;
```

This line compiles to the following three instructions:

```
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       7
   6:   return
```

It's interesting to note that an "If Nodes **is** `NULL`..." source code line generates a "If Nodes is **not** `NULL`..." instruction (**ifnonnull**) in the compiled code! The reason? It takes fewer instructions to do it that way. Consider the alternative:

```
public static void PrintList();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnull          9
   6:   goto            10
   9:   return
```

The instruction **areturn**[17] does basically the same thing as **ireturn**, except that it doesn't return an integer as a return value. Instead, it returns an object referance—a pointer to something other than your usual types like **int** and **float**.[18] For example, the statement

```
return Nodes;
```

in **Head()** compiles to

```
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   areturn
```

The **getstatic** instruction pushes the value of **Nodes** onto the stack, and then the instruction **areturn** pops that value, and pushes it onto the stack of the function which called **Head()**.

The last two new instructions, **getfield** and **putfield** work basically the same as **getstatic** and **putstatic**, except that they load and save values to and from non-static member variables ("fields") in a class. For instance, in the line

[17] That's *a return*, not *are turn*.

[18] Again, the 'a' stands for "address."

```
if (Value < Nodes.Value)  {
```

the fetching of **Value**, whose “full name” is **this.Value**, compiles to

```
11:  aload_0
12:  getfield        #2; //Field Value:I
```

The **aload_0** instruction pushes the address in Slot 0, which is **this**, onto the stack. The **getfield** instruction—yes, keep in mind that this is a JVM machine instruction—then gets the **Value** field from within the class instance pointed to by **this**.

The file **Intro.java** illustrates the usage of the **NumNode** class:

```
// usage:  java Intro nums

// reads integers from the command line, storing them in a linear linked
// list, maintaining ascending order, and then prints out the final list
// to the screen

public class Intro

{  public static void main(String[] Args) {
      int NumElements = Args.length;
      for (int I = 1; I <= NumElements; I++)  {
         int Num;
         Num = Integer.parseInt(Args[I-1]);
         NumNode NN = new NumNode(Num);
         NN.Insert();
      }
      System.out.println("final sorted list:");
      NumNode.PrintList();
   }
}
```

It compiles to

```
Compiled from Intro.java
public class Intro extends java.lang.Object {
    public Intro();
    public static void main(java.lang.String[]);
}

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

Method void main(java.lang.String[])
   0 aload_0
   1 arraylength
```

```
 2 istore_1
 3 iconst_1
 4 istore_2
 5 goto 35
 8 aload_0
 9 iload_2
10 iconst_1
11 isub
12 aaload
13 invokestatic #2 <Method int parseInt(java.lang.String)>
16 istore_3
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
27 aload 4
29 invokevirtual #5 <Method void Insert()>
32 iinc 2 1
35 iload_2
36 iload_1
37 if_icmple 8
40 getstatic #6 <Field java.io.PrintStream out>
43 ldc #7 <String "final sorted list:">
45 invokevirtual #8 <Method void println(java.lang.String)>
48 invokestatic #9 <Method void PrintList()>
51 return
```

There are again some new instructions to discuss here. First, consider the Java statement

```
int NumElements = Args.length;
```

in **Intro.java**. Java is a more purely object-oriented language than C++, and one illustration of that is that in Java arrays are objects. One of the member variables in the **Array** class is **length**, the number of elements in the array. Thus the compiler translates the fetch of **Args.length** to

```
0 aload_0
1 arraylength
```

Note again that **arraylength** is a JVM machine instruction. This is not a subroutine call.

Now consider the statement

```
NumNode NN = new NumNode(Num);
```

It compiles to

```
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
```

The JVM instruction set includes an instruction **new**, which allocates space from the heap for the object to be created of class **NumNode**. The instruction pushes a pointer to that space. Next, the **dup** (“duplicate”) instruction pushes a second copy of that pointer, to be used later in the compiled code for

```
NN.Insert();
```

But before then we still need to call the constructor for the **NumNode** class, which is done in offset 22, after pushing the parameter in offset 21. The assignment of the pointer to **NN** then is done in offset 25.

2.8 Overview of JVM Instructions

In the following, *top* will refer to the element at the top of the operand stack, and *nexttop* will refer to the element next to it.

- **aaload**:

  Format: 0x32

  Loads an element of an array of addresses (i.e. from a multidimensional array). Treats *nexttop* as a pointer to an array (i.e. a variable declared of array type), and *top* is an index into the array. The instruction loads the array element and pushes it onto the operand stack. In other words, *nexttop* and *top* are popped, and *nexttop*[*top*] is fetched and pushed onto the operand stack.

- **arraylength**:

  Format: 0xbe

  Pops the operand stack to get the array address, and then pushes the length of the array.

- **aastore**:

  Format: 0x53

  Does the opposite of **aaload**, popping the operand stack first to get *value*, then to get *index*, the finally to get *address*. It then stores *value* into element *index* of the array starting at *address*.

- **aconst_null**:

  Format: 0x01

  Pushes the value `NULL` onto the stack.

- **aload**:

  Format: 0x19 *8bitindex*

  Treats the value in slot *8bitindex* as an address, and pushes it onto the stack.

- **aload_0**, **aload_1**, **aload_2**, **aload_3**:

  Formats: 0x2a, 0x2b, 0x2c

  The instruction **aload_0**, is the same as **aload**, but specifically for slot 0. The others are analogous.

- **areturn**:

  Format: 0xb0

  Does exactly the same thing as **ireturn**, but instead of returning an integer, it returns an object referance.

- **astore**:

  Format: 0x3a, *8bitindex*

  Opposite of **aload**, popping the operand stack and placing the popped value (presumed to be an address) into the specified slot.

- **astore_0**, **astore_1**, **astore_2**, **astore_3**:

  Formats: 0x4b, 0x4c, 0x4d, 0x4e

  Same as **astore**, but specifically for slot 0, slot 1 etc..

- **bipush**:

  Format: 0x10 *8bitinteger*

  Pushes the given 8-bit integer onto the operand stack.

- **dup**:

  Format: 0x59

  Duplicates the top word on the operand stack, so the operand stack now has two copies of that word instead of one.

- **dup_x2**:

  Format: 0x5b

  Like **dup**, but places the duplicate copy of the top of the stack just past the third word in the stack. For example, (12,5,13,8) becomes (12,5,13,12,8).

- **getfield**:

  Format: 0xb5 *16bitindex*

  Opposite of **putfield**. Gets value of the given non-static item, then pushes it onto the operand stack.

- **getstatic**:

  Format: 0xb2 *16bitindex*

  Opposite of **putstatic**. Gets value of the given static item, then pushes it on the operand stack.

- **goto**:

  Format: 0xa7 *16bitjumpdistance*

  Unconditional jump. See **if_icmpeq** for explanation of *16bitjumpdistance*.

- **i2f**:

  Format: 0x86

  Pops the top word on the stack, converts it from **int** to **float**, and pushes the new value back onto the stack.

- **iadd**:

  Format: 0x60

  Pops *nexttop* and *top*, and pushes the sum *nexttop* + *top*.

- **iaload**:

  Format: 0x2e

  Load an element of an integer array. See **aaload** above.

- **iastore**:

  Format: 0x4f

  Store to an element of an integer array. See **aastore** above.

- **iconst_0**, **iconst_1**, **iconst_2**, **iconst_3**, **iconst_4**, **iconst_5**:

  Format: 0x3, 0x4, 0x5, 0x6, 0x7, 0x8

  Pushes the integer constant 0, 1, 2 etc. onto the operand stack.

- **idiv**:

  Format: 0x6c

  Pops *nexttop* and *top*, and pushes the quotient *nexttop* / *top*.

- **if_icmpeq**:

  Format: 0x9f *16bitjumpdistance*

  If *top* = *nexttop*, jumps the given distance to the branch target. The quantity *16bitjumpdistance* is a 2-byte, 2s complement signed number, measured from the jump instruction. Both *top* and *nexttop* must be integers; a runtime error occurs if no.

- **if_icmpge**:

  Format: 0xa2 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if $nexttop \geq top$.

- **if_icmple**:

  Format: 0xa4 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if $nexttop \leq top$.

- **if_icmplt**:

  Format: 0xa1 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if $nexttop < top$.

- **if_icmpne**:

  Format: 0xa0 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if $top \neq nexttop$.

- **ifnonnull**:

  Format: 0xc7 *16bitjumpdistance*

  Same as the previous jump conditions, but jumps if $top \neq$ `NULL`.

- **ifnull**:

  Format: 0xc6 *16bitjumpdistance*

  Same as the previous jump conditions, but jumps if top = `NULL`.

- **iinc**:

  Format: 0x84 *8bitindex 8bitinteger*

  Increments slot *8bitindex* by the amount *8bitinteger*.

- **iload**:

  Format: 0x15 *8bitindex*

  Same as **aload** but for integers instead of addresses.

- **iload_0**, **iload_1**, **iload_2**, **iload_3**:

  Formats: 0x1a, 0x1b, 0x1c, 0x1d

  Same as **aload_0** etc. but for integers instead of addresses.

- **imul**:

  Format: 0x68

  Pops *nexttop* and *top*, and pushes the product *nexttop* $\times$ *top*.

- **invokespecial**:

  Format: 0xb7 *16bitindex*

  Like **invokevirtual**, but for constructors, superclass and other special situations.

- **invokestatic**:

  Format: 0xb8 *16bitindex*

  Method call. The quantity *16bitindex* serves as an index into the Constant Pool, pointing to the given method. Creates a new stack frame for the method. Pops the method's arguments from the caller's operand stack and places them into the method's Local Variables Section. Points the **frame** register to the method's stack frame, and jumps to the method.

- **invokevirtual**:

  Format: 0xb6 *16bitindex*

  Same as **invokestatic**, but the arguments include the "hidden" argument **this**, i.e. a pointer to the object this method is being invoked on.

- **ireturn**:

  Format: 0xac

  Return from method with return value. Pops integer from current operand stack, places it on the caller's operand stack, restores the **frame** register to point to the caller's stack frame, and jumps back to the caller.

- **istore**:

  Format: 0x36 *8bitindex*

  Same as **astore** but for integers instead of addresses.

- **istore_0**, **istore_1**, **istore_2**, **istore_3**:

  Formats: 0x3b, 0x3c, 0x3d, 0x3e

  Same as **astore_0** etc. but for integers instead of addresses.

- **isub**:

  Format: 0x64

  Pops *nexttop* and *top*, and pushes the difference *nexttop* - *top*.

- **ldc**:

  Format: 0x12 *8bitindex*

  Gets an item from the Constant Pool and pushes it onto the operand stack.

- **new**:

  Format: 0xbb *16bitindex*

  Performs the Java **new** operation, with *16bitindex* being the index into the Constant Pool, pointing to the given class. Creates the given object using memory from the Heap, and then pushes the address of the new object on the operand stack.

- **putfield**:

  Format: 0xb4 *16bitindex*

  Pops the operand stack, and assigns the popped value to the non-static item given by *16bitindex*.

- **putstatic**:

  Format: 0xb3 *16bitindex*

  Pops the operand stack, and assigns the popped value to the static item given by *16bitindex*.

- **return**:

  Format: 0xb1

  Same as **ireturn**, but for methods without a return value.

- **swap**:

  Format: 0x5f

  Swaps the top two words of the operand stack.

2.9 References

- Bill Venners' book, *Inside the Java Virtual Machine*. Available on the Web (with many helpful URL links) at `www.artima.com`.

- Sun's "official" definition of the JVM: *The Java Virtual Machine Specification*, by Lindholm and Yellin, also available on the Web, at `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`. Chapter 6 gives specs (mnemonics, op codes, actions, etc.) on the entire JVM instruction set.