The Java Virtual Machine (JVM)

- Well-defined stack-oriented bytecode
- Special instructions to support OO, arrays, ...
- Metadata describing classes, fields, methods
**Example Java program**

```java
class Node extends Object {
    Node next;
    Node prev;
    int item;
}
class LinkedList extends Object {
    Node first, last;
    void addLast(int item) {
        Node node = new Node();
        node.item = item;
        if (this.last == null) {
            this.first = node;
            this.last = node;
        } else {
            this.last.next = node;
            node.prev = this.last;
            this.last = node;
        }
    }
    void printForwards() { ... }
    void printBackwards() { ... }
}
```

**The JVM class file format**

```
#1 Object.<init>()
#2 class Node
#3 Node.<init>()
#4 int Node.item
#5 Node LinkedList.last
#6 Node LinkedList.first
#7 Node Node.next
#8 Node Node.prev
#9 void InOut.print(int)
```

```
header
    LinkedList extends Object

pool
    #1 Object.<init>()
    #2 class Node
    #3 Node.<init>()
    #4 int Node.item
    #5 Node LinkedList.last
    #6 Node LinkedList.first
    #7 Node Node.next
    #8 Node Node.prev
    #9 void InOut.print(int)

fields
    first (#6)
    last (#5)

methods
    <init>()
    void addLast(int)
    void printForwards()
    void printBackwards()

class attributes
    source "ex6java.java"
```
The JVM bytecode instructions

- push constant onto local stack: `iconst, ldc, aconst_null, ...`
- arithmetic: `iadd, isub, imul, idiv, irem, ineg, iinc, fadd, ...`
- load local variable onto local stack: `iload, aload, fload, ...`
- store local variable from local stack: `istore, astore, fstore, ...`
- load array element onto local stack: `iaload, baload, aaload, ...`
- stack manipulation: `swap, pop, dup, dup_x1, dup_x2, ...`
- load field onto local stack: `getfield, getstatic`
- method call: `invokestatic, invokevirtual, invokespecial`
- method return: `return, ireturn, areturn, freturn, ...`
- unconditional jumps: `goto`
- conditional jumps (compare to 0): `ifeq, ifne, iflt, ifle, ifgt, ifge`
- conditional jumps (compare values): `if_icmpeq, if_icmpne, ...`
- object-related: `new, instanceof, checkcast`

Type prefixes: `i=int, a=object, f=float, d=double, s=short, b=byte, ...`

JVM bytecode verification

The JVM bytecode is **statically verified** before execution:

- All instructions must work on stack operands and local variables of the correct type
- A method must use no more local variables and no more local stack positions than it claims to
- For every point in the bytecode, the local stack has the same depth whenever that point is reached
- A method must throw no more exceptions than it claims to
- The execution of a method must end with a return or throw instruction, not ‘fall off the end’
- Execution must not use one half of a two-word value (e.g. a long) as a one-word value (int)
Additional JVM runtime checks

- Array-bounds checks
- Array assignment checks: Can store only subtypes of A into A-array
- Null-reference check (a reference is null or points to an object or array, because no pointer arithmetics)
- Checked casts: Cannot make arbitrary conversions between object classes
- Memory allocation succeeds or throws exception
- No manual memory deallocation or reuse

Bottom line: A JVM program cannot read or overwrite arbitrary memory
- No buffer overflow attacks, worms, etc

Example JVM runtime state

```java
void m() {
    LinkedList lst = new LinkedList();
    lst.addLast(5);
    lst.addLast(7);
    Node node = lst.first;
}
```
The .NET Common Language Runtime (CLR, CLI)

• Much the same philosophy and design as JVM, but with some improvements:
  – Standardized bytecode assembly format
  – Better versioning, strongnames, ...
  – Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, ...)
  – Tail calls to support functional languages
  – User-defined value types
  – True generic types in bytecode, safer, more efficient and more complex

Canonical compilation?

• Consider the Java/C#/C program:

```csharp
static void Main(string[] args) {
    int n = int.Parse(args[0]);
    int y;
    y = 1889;
    while (y < n) {
        y = y + 1;
        if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
            InOut.PrintI(y);
    }
    InOut.PrintC(10);
}
```

• Let us compile it twice:
  – as Java, to JVM bytecode
  – as C#, to CLR bytecode
Ten-minute exercise

- On a printout of the preceding slide
  - Draw arrows to indicate where jumps go
  - Draw blocks around the bytecode segments corresponding to fragments of the Java program
**Just-in-time (JIT) compilation**

- Originally, JVM bytecode was interpreted.
- Now bytecode is compiled to real (e.g. x86) machine code at runtime, for speed.
- .NET bytecode has always been JIT compiled.
- Java/C# speed is comparable to C/C++.

---

**Alternative: JVM in silicon**

- There have been many attempts to make JVMs in silicon:
  - picoJava (Sun 1996?)
  - aJile
  - ARM Jazelle (a few instructions JVM support)
- Mostly useful in embedded devices:
  - Writable memory is scare (uses energy)
  - Hence JIT compilation is a bad idea.
“Real” machines: The x86

- The x86 is the dominant desktop and laptop (and server?) CPU architecture
- Other architectures (Hitachi H8, ARM, TI) dominate embedded systems, PDAs, mobile phones, ...
- 8086 → 80286 → 386 → 486 → Pentium → Pentium Pro → Pentium II → Pentium III → Pentium 4 (dying out)
  → Pentium III → Pentium M → Core → Core 2 → Core 2 Duo

The x86 architecture

- A register machine with few registers:
  - esp: stack pointer
  - ebp: base pointer
  - eax, ebx, ecx, edx: int arithmetic registers
  - edi, esi: indexing registers
  - st0-st7: floating-point registers (32-80 bit)
  - mm0-mm7: MMX multimedia registers
- Powerful and irregular instruction set:
  - Many instructions work only on particular registers
  - Multimedia instructions: SSE, MMX1, ...
  - Instruction length: 1 byte to 15 bytes
x86 registers

Figure A.24 IA-32 Register Set. Registers shown are for 32-bit protected mode operation.

Register machines, principle

IT University of Copenhagen

www.itu.dk
Register machines, reality: Pentium M incarnation of x86

- Instruction pipeline:
  - An instruction is decoded in multiple steps
  - Many instr. being decoded simultaneously

- Register renaming
  - Approx. 40 internal (shadow) registers
  - Dynamically mapped to eax, ebx, ...

- Out-of-order execution
  - Late instructions may execute before early
  - Using different shadow registers

- Multiple integer arithmetic/logic units
A machine code program
(x86, tryadd.asm)

- Compute 2+3 and print the result

```assembly
main:
    mov eax, 2 ; put 2 in eax
    mov ebx, 3 ; put 3 in ebx
    add eax, ebx ; add ebx to eax
    push dword eax ; push eax on stack
    push dword mystring ; push pointer
    call printf ; call C library fcn.
    add esp, byte 8 ; pop 8 bytes from stack
    ret ; return

segment DATA
myint     dd 1234
mystring  db      'The result is ->%d<-', 10, 0
```

The factorial function (tryfac.asm)

```assembly
fac:
    push    ebp ; Save ebp on stack
    mov     ebp, esp ; Save stack pointer in ebp
    mov     eax, [ebp+8] ; Get argument n
    cmp     eax, 0
    je      .false ; If n==0 goto .false
    dec     eax ; else compute n-1 ...
    push    eax ; and push as argument
    call    fac ; and compute fac(n-1)
    add     esp, byte 4 ; pop argument
    mul     dword [ebp+8] ; then multiply eax by n
    jmp     .end
    .false: mov     eax, 1 ;
    .end:   mov     esp, ebp ; Restore esp from ebp
             pop     ebp ; Restore ebp from stack
             ret ; Return
```
How fast are Java and C#?
How predictable are the x86 processors?

- Case studies
  - Matrix multiplication
  - A division intensive series
  - Polynomial evaluation
- Languages
  - C
  - C#
  - Java
- Execution platforms:
  - C: gcc, Linux
  - C#: Microsoft .NET and Mono
  - Java: Sun Hotspot client and server, and IBM JVM
- Hardware:
  - Intel Pentium M, 1600 MHz

Naïve matrix multiplication

- Computing \( R = A \times B \) for matrices \( A, B, R \)
  ```java
  for (r=0; r<rRows; r++) {
    for (c=0; c<rCols; c++) {
      double sum = 0.0;
      for (k=0; k<aCols; k++)
        sum += A.data[r*aCols+k] * B.data[k*bCols+c];
      R.data[r*rCols+c] = sum;
    }
  }
  ```
- Pretty much the same code in Java, C#
- For 80x80, performs \( 80^3 = 512,000 \) mults
Matrix multiplication benchmark results (us/mult)

<table>
<thead>
<tr>
<th>Language/CMS</th>
<th>Time (us/iter)</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>2.8</td>
<td>4.4</td>
</tr>
<tr>
<td>C# matmult1 Microsoft</td>
<td>3.2</td>
<td>5.0</td>
</tr>
<tr>
<td>C# matmult1 Microsoft, ngen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult1 Mono</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult2 Microsoft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult2 Microsoft, ngen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult2 Mono</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult3 Microsoft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult3 Microsoft, ngen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# matmult3 Mono</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java, Sun Hotspot -client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java, IBM JVM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Computing a series (division intensive)

- Given \( M \) find \( n \) such that

\[
\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \geq M
\]

```java
double sum = 0.0;
int n = 0;
while (sum < M) {
    n++;
    sum += 1.0/n;
}
```

A division takes 32 cpu cycles, dominates all other costs.
Division intensive code, benchmark results (ns/iter)

<table>
<thead>
<tr>
<th>Language</th>
<th>Benchmark (ns/iter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>25.6</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>21.3</td>
</tr>
<tr>
<td>C# Microsoft, ngen</td>
<td>21.3</td>
</tr>
<tr>
<td>C# Mono</td>
<td>21.4</td>
</tr>
<tr>
<td>Java, Sun Hotspot -client</td>
<td>21.0</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>20.2</td>
</tr>
<tr>
<td>Java, IBM JVM</td>
<td>21.2</td>
</tr>
</tbody>
</table>

32 cycles at 1.6 GHz takes 20 ns
Division, increment, addition and test done in parallel

Evaluating a polynomial

- Evaluate for given $x$:

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

```csharp
double res = 0.0;
for (int i=0; i<cs.Length; i++)
    res = cs[i] + x * res;
```
Benchmark results, polynomial order 1000 (us/eval)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>5.2</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>5.1</td>
</tr>
<tr>
<td>C# Microsoft, ngen</td>
<td>5.1</td>
</tr>
<tr>
<td>C# Mono</td>
<td>8.3</td>
</tr>
<tr>
<td>Java, Sun Hotspot -client</td>
<td>5.9</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>5.1</td>
</tr>
<tr>
<td>Java, IBM JVM</td>
<td>5.3</td>
</tr>
</tbody>
</table>

5.1 ns (8 cycles) for:
- index increment
- test
- array bounds check
- memory access
- multiplication
- addition

The effect of machine architecture

- Is a 2.8 MGHz Pentium 4 faster or slower than 1.6 GHz Pentium M?
- Answer: Both

<table>
<thead>
<tr>
<th>Language</th>
<th>Pentium M 1.6</th>
<th>Pentium 4 2.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>5.3</td>
<td>4.7</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>C# Microsoft, ngen</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>C# Mono</td>
<td>8.4</td>
<td>13.0</td>
</tr>
<tr>
<td>Java, Sun Hotspot -client</td>
<td>5.9</td>
<td></td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>Java, IBM JVM</td>
<td>5.2</td>
<td></td>
</tr>
</tbody>
</table>

11% faster

55% slower
Compiling Java

- To compile source file Source.java:
  ```bash
  javac Source.java
  ```
- The result is a set of `.class` files containing bytecode and metadata
- To disassemble file MyClass.class:
  ```bash
  javap -c MyClass
  ```
- The result is text
- How inspect the JITted machine code?
- Allegedly, Sun HotSpot JVM 7 beta can display the x86 machine code

Compiling C

- Here we use Linux, e.g. on ssh.itu.dk
- To compile file `program.c`:
  ```bash
  gcc -O3 program.c -o program
  ```
- The result is an executable file `program`
- Get the machine code for `program.c`:
  ```bash
  gcc -S -O3 program.c -o program.s
  ```
- The result is a text file `program.s`
Compiling C# (Microsoft)

- To compile source file Source.cs:
  `csc /o Source.cs`
- The result is an assembly `Source.exe` containing metadata and bytecode
- To disassemble `Source.exe`:
  `ildasm /text Source.exe`
- The result is text
- To inspect the JITted machine code:
  `cordbg Source.exe`
  `(cordbg) b Class::Method`
  `(cordbg) g`
  `(cordbg) dis 1000`

Compiling C# (Mono)

- To compile source file Source.cs:
  `gmcs Source.cs`
- The result is an assembly `Source.exe` containing metadata and bytecode
- To disassemble `Source.exe`:
  `monodis /text Source.exe`
  Or use `ildasm` from Microsoft
- The result is text
- To inspect the JITted machine code:
  `mono –v –v Source.exe`
- You get very verbose output. Search for `Class_Method` to locate the code
Decompiling Java and C#

• For Java, use Jdec or Jad or ...
• For C#, use Lutz Roeder’s Reflector
• These are pretty useful tools:
  – when the computer containing your source files are stolen; and
  – when the developers of a useful library forgot to document it.
• Also a bit scary: Your source code can be recovered quite well from Java *.class files and .NET assemblies.