Advanced Models and Programs

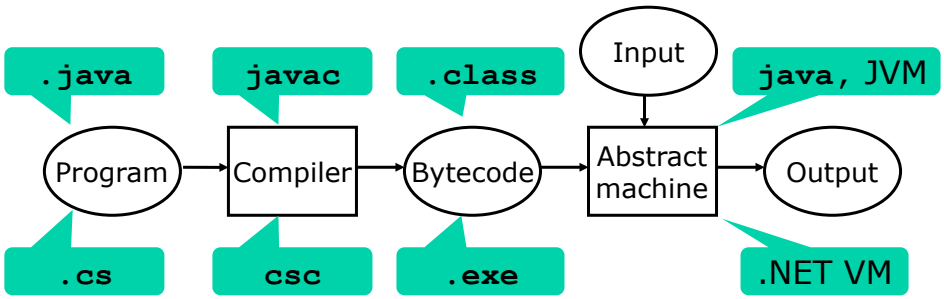
Bytecode and real machine code

Peter Sestoft

Wed 2011-02-23* and
Mon 2011-02-28

The Java Virtual Machine (JVM)

- Well-defined stack-oriented bytecode
- Special instructions to support OO, arrays, …
- Metadata describing classes, fields, methods



Example Java program

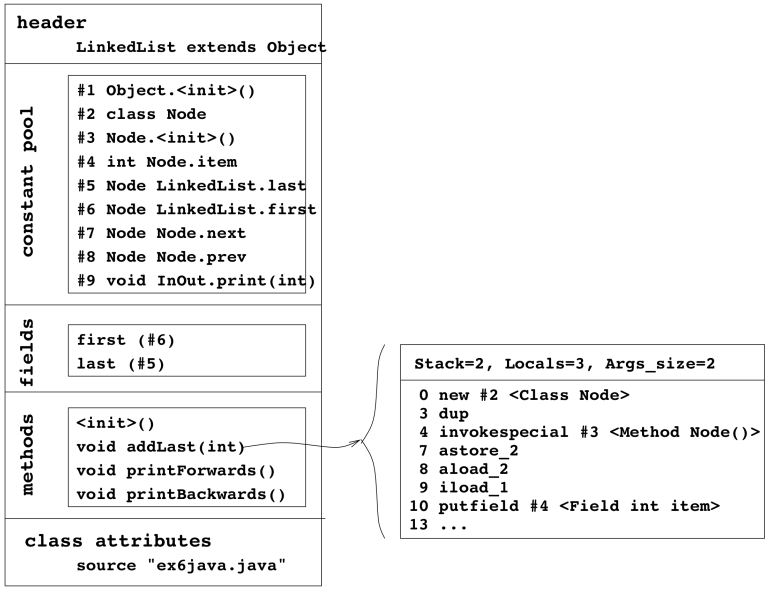
```
class Node extends Object {
  Node next;
  Node prev;
  int item;
}
```

```
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



Some JVM bytecode instructions

Kind	Example instructions
push constant	iconst, ldc, aconst_null, …
arithmetic	iadd, isub, imul, idiv, irem, ineg, iinc, fadd, …
load local variable	iload, aload, fload, …
store local variable	istore, astore, fstore, …
load array element	iaload, baload, aaload, …
stack manipulation	swap, pop, dup, dup_x1, dup_x2, …
load field	getfield, getstatic
method call	invokestatic, invokevirtual, invokespecial
method return	return, ireturn, areturn, freturn, …
unconditional jump	goto
conditional jump	ifeq, ifne, iflt, ifle, …; 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:

- An instruction 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 admits 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)

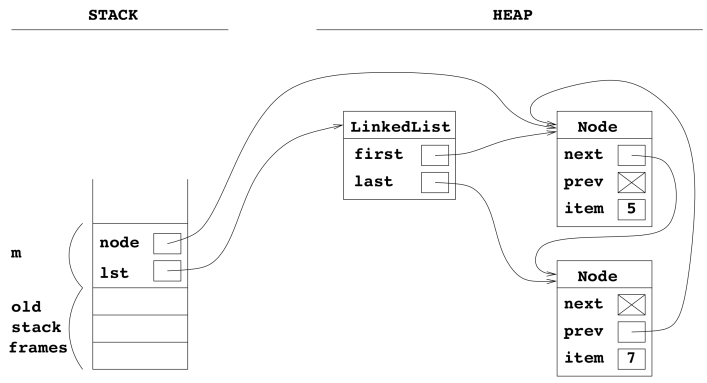
Additional JVM *runtime* checks

- Array-bounds checks
- Array assignment checks: Can store only subtypes of A into an 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
- Better debugging, better security
- No buffer overflow attacks, worms, etc as in C/C++

Example JVM runtime state

```
void m() {
  LinkedList lst = new LinkedList();
  lst.addLast(5);
  lst.addLast(7);
  Node node = lst.first;
}
```



The .NET Common Language Infrastructure (CLI, CLR)

- 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, …)
  - User-defined value types (structs)
  - Tail calls to support functional languages
  - True generic types in bytecode: safer, more efficient, and more complex…
- The .exe file = stub + bytecode
- Standardized as Ecma-335

Some .NET CLI bytecode instructions

Kind	Example instructions
push constant	ldc.i4, ldc.r8, ldnull, ldstr, ldtoken
arithmetic	add, sub, mul, div, rem, neg; add.ovf, sub.ovf, …
load local variable	ldloc, ldarg
store local variable	stloc, starg
load array element	ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8
stack manipulation	pop, dup
load field	ldfld, ldstfld
method call	call, calli, callvirt
method return	ret
unconditional jump	br
conditional jump	brfalse, brtrue; beq, bge, bgt, ble, blt, …; bge.un …
object-related	newobj, isinst, castclass

Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double,

Canonical compilation?

- Consider the Java/C#/C program:

```
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

JVM

```
 0 aload_0                  |
 1 iconst_0                 |
 2 aaload                   |
 3 invokestatic #2 (...)    |
 6 istore_1                 |
 7 sipush 1889              |
10 istore_2                 |
11 goto 43                  |
14 iload_2                  |
15 iconst_1                 |
16 iadd                     |
17 istore_2                 |
18 iload_2                  |
19 iconst_4                 |
20 irem                     |
21 ifne 31                  |
24 iload_2                  |
25 bipush 100               |
27 irem                     |
28 ifne 39                  |
31 iload_2                  |
32 sipush 400               |
35 irem                     |
36 ifne 43                  |
39 iload_2                  |
40 invokestatic #3 (...)    |
43 iload_2                  |
44 iload_1                  |
45 if_icmplt 14             |
48 bipush 10                |
50 invokestatic #4 (...)    |
53 return                   |
```

.NET

```
IL_0000:  ldarg.0                 |  args
IL_0001:  ldc.i4.0                |
IL_0002:  ldelem.ref              |  args[0]
IL_0003:  call        (...)       |  parse int
IL_0008:  stloc.0                 |  n = ...
IL_0009:  ldc.i4      0x761       |
IL_000e:  stloc.1                 |  y = 1889;
IL_000f:  br.s        IL_002f     |  while (...) {
IL_0011:  ldloc.1                 |
IL_0012:  ldc.i4.1                |
IL_0013:  add                     |
IL_0014:  stloc.1                 |  y = y + 1;
IL_0015:  ldloc.1                 |
IL_0016:  ldc.i4.4                |
IL_0017:  rem                     |
IL_0018:  brtrue.s    IL_0020     |  y % 4 == 0
IL_001a:  ldloc.1                 |
IL_001b:  ldc.i4.s    100         |
IL_001d:  rem                     |
IL_001e:  brtrue.s    IL_0029     |  y % 100 != 0
IL_0020:  ldloc.1                 |
IL_0021:  ldc.i4      0x190       |
IL_0026:  rem                     |
IL_0027:  brtrue.s    IL_002f     |  y % 400 == 0
IL_0029:  ldloc.1                 |
IL_002a:  call        (...)       |  print y
IL_002f:  ldloc.1                 |
IL_0030:  ldloc.0                 |
IL_0031:  blt.s       IL_0011     |  (y < n) }
IL_0033:  ldc.i4.s    10          |
IL_0035:  call        (...)       |  newline
IL_003a:  ret                     |  return
```

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

No generic types in JVM bytecode

- Type parameter T gets erased to Object
- No type at runtime to represent T

```
class C<T> {
  T f;
  void m(Object o) {
    T[] arr = new T[10];
    Class ty = T.class;
    if (o instanceof T) {
      T t = (T)o;
    }
  }
}
```

Field f has type Object

Illegal/doesn't work in Java

```
void mo(C<Integer> x) { }
void mo(C<String> x)  { }
```

Illegal in Java

Why cannot Java create array T[5]?

- Array assignment requires runtime check:

```
static void storePerson(Person[] arr, Person x) {
  arr[0] = x;
}
```

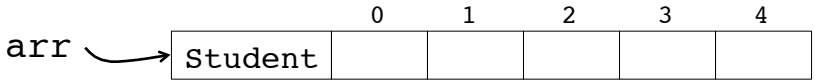
Runtime check here

```
Student[] sarr = new Student[5];
Person[] parr = sarr;
storePerson(parr, new Person());
```

Must fail ...

... else sarr[0] now contains a Person, not a Student!

- For the check, a Java/C# array must contain its *exact* element type:



But T in C<T> is erased to Object

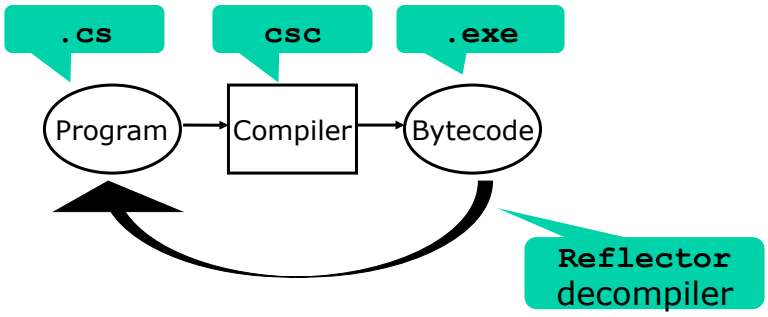
- So **`new T[5]`** would do **`new Object[5]`**
- That would disable array assignment checks
- Also, no runtime type for constructed types, such as **`Pair<String,Integer>`**
- Therefore, Java cannot create array of constructed type:

```
... new Pair<String,Integer>[5] ...
```

- Java workaround: Use **`new ArrayList<T>()`** instead of **`new T[5]`**. Why does this work?

Metadata and decompilers

- The .class and .exe files contains *metadata*: names and types of fields, methods, classes
- One can *decompile* bytecode into programs:

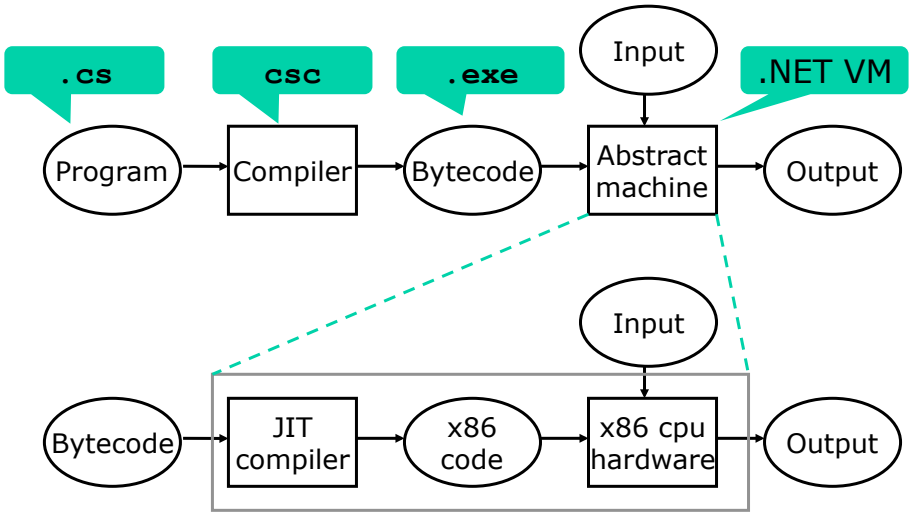


- For Java, use Jdec or Jad or …
- Bad for protecting intellectual property
- *Bytecode obfuscators* make decompilation harder

17

Just-in-time (JIT) compilation

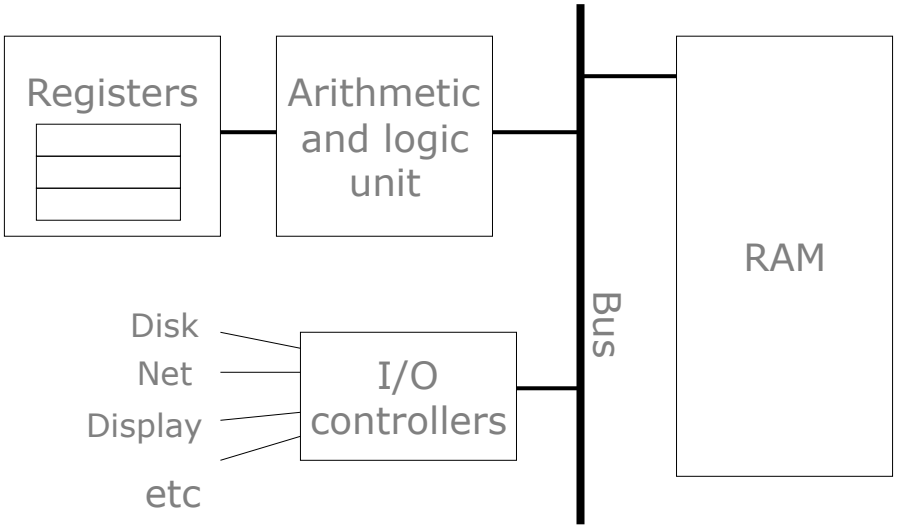
- Bytecode is compiled to real (e.g. x86) machine code at runtime to get speed comparable to C/C++



"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, … 95-98% of all CPUs
- IBM zSeries dominate mainframe market; complicated and expensive but very fast
- 8086 → 80286 → 386 → 486 → Pentium → Pentium Pro → Pentium II → Pentium III → Pentium 4 (dying out)
  → Pentium III → Pentium M → Core → Core 2 (x86-64) → Core 2 Duo → Intel i7

Register machines, principle



x86 (IA32, i386) registers

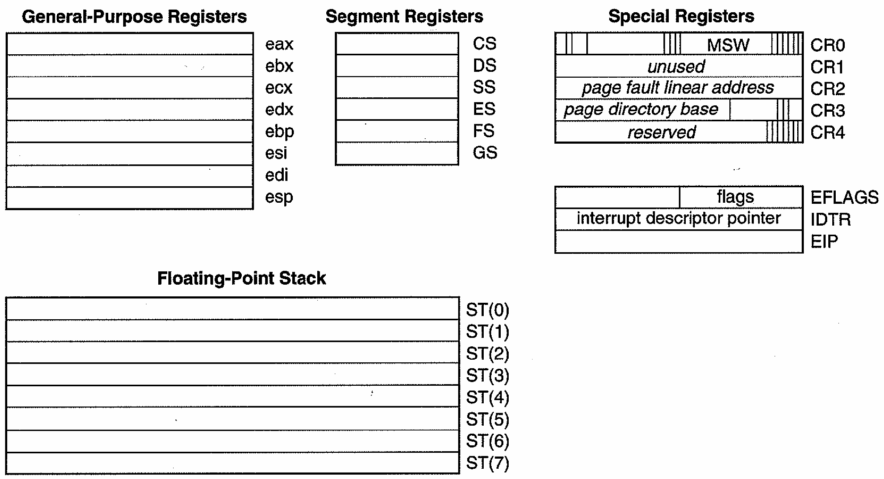
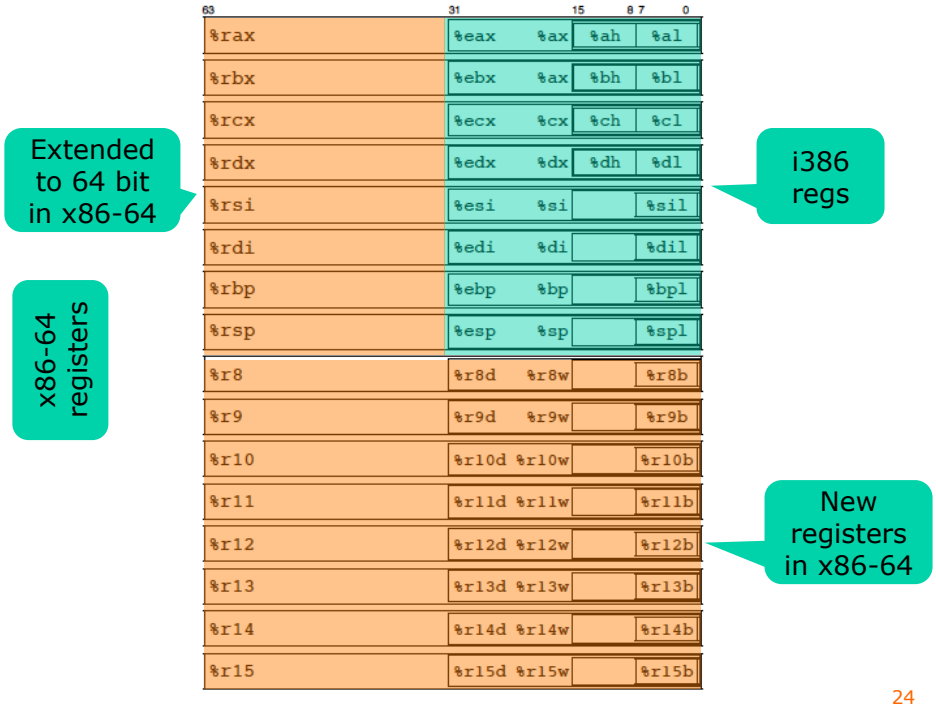


Figure A.24 IA-32 Register Set. *Registers shown are for 32-bit protected-mode operation.*

The x86 (IA32, i386) 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 very irregular instruction set:
  - Many instructions work only on particular registers
  - Multimedia instructions: SSE, MMX1, …
  - Instruction length: 1 byte to 15 bytes
  - Mixed-size registers: AH, AL (8 bit); AX (16); EAX (32); RAX (64)



24

A machine code program (Linux x86, tryadd.asm)

- Compute 2+3 and print the result

```
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
```

Newline

String terminator

27

The factorial function (tryfac.asm)

argument n
on stack

```
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
```

leave

How good are the JIT compilers?

```
private static int fac(int n) {
  if (n==0)
    return 1;
  else
    return n * fac(n-1);
}
```

C#

```
0000: ldarg.0
0001: brtrue 0008
0006: ldc.i4.1
0007: ret
0008: ldarg.0
0009: ldarg.0
000a: ldc.i4.1
000b: sub
000c: call fac(int32)
0011: mul
0012: ret
```

CIL bytecode

Code from Mono 2.10 JIT on MacOS

```
0000: ldarg.0
0001: brtrue 0008
0006: ldc.i4.1
0007: ret
0008: ldarg.0
0009: ldarg.0
000a: ldc.i4.1
000b: sub
000c: call fac(int32)
0011: mul
0012: ret
```

.NET/CLI bytecode

argument n on stack

```
00  pushq %rbp
01  movl  %esp,%ebp
03  subl  $0x08,%esp
06  cmpl  $0x00,0x08(%rbp)
0a  jne   0x00000013
0c  movl  $0x00000001,%eax
11  jmp   0x0000002b
13  movl  0x08(%rbp),%eax
16  subq  $0x0c,%rsp
1a  pushq %rax
1b  callq 0x10000000
20  addl  $0x10,%esp
23  movl  %eax,%ecx
25  movl  0x08(%rbp),%eax
28  imull %ecx,%eax
2b  leave
2c  ret
```

??

x86-64 code by Mono

What code from C (gcc 4.2.1)?

C

```
int fac(int n) {
  if (n==0)
    return 1;
  else
    return n*fac(n-1);
}
```

argument n in reg %edi

```
_fac:
  pushq  %rbp              ; save old base pointer
  movq   %rsp, %rbp        ; set new base pointer
  pushq  %rbx              ; save %rbx
  subq   $8, %rsp          ; increase stack
  movl   %edi, %ebx        ; move n into %ebx
  movl   $1, %eax          ; move 1 into %eax
  testl  %edi, %edi
  je     L4                ; if n==0 goto L4
  leal   -1(%rbx), %edi    ; subtract 1 from n
  call   _fac              ; call fac, result %eax
  imull  %ebx, %eax        ; n*fac(n-1) giving %eax
L4:
  addq   $8, %rsp          ; decrease stack
  popq   %rbx              ; restore %rbx
  leave                    ; restore old base ptr
  ret                      ; return
```

x86-64 for MacOS

How fast are C, C# and Java?

- Case studies
  - Matrix multiplication
  - A division intensive series
  - Polynomial evaluation
  - A statistical function (NORMDIST)
- Languages
  - C
  - C#
  - Java
- Execution platforms:
  - C: gcc 4.2.1, MacOS
  - C#: Microsoft .NET 4.0 and Mono 2.6
  - Java: Sun Hotspot server (unfortunately not IBM JVM)
- Hardware:
  - Intel Core 2 Duo, 2660 MHz

Naïve matrix multiplication

- Computing R = A * B
  for matrices A, B, R with **`double`** elements

```
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 fp. mul/add

Matrix multiplication benchmark results (us/mult)

C (gcc -O3)	702
C# matmult1 Microsoft	3218
C# matmult1 Mono	4627
C# matmult2 Microsoft	1165
C# matmult2 Mono	1943
C# matmult3 Microsoft	1575
C# matmult3 Mono	2888
Java, Sun Hotspot -server	1180

1.4 ns/iter = 3.7 cycles

C-style unsafe code

array of arrays

Computing a series (division intensive)

- Given M find n such that

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq M$$

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

A division takes 20 cpu cycles, dominates all other costs

Division intensive code, benchmark results (ns/iter)

C (`gcc -O3`)	7.9
C# Microsoft	7.7
C# Mono	7.6
Java, Sun Hotspot -server	10.6

20 cycles per division at 2.66 GHz takes 7.5 ns
Division, increment, addition and test done in parallel

Evaluating a polynomial

- Evaluate for given x:

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

```
double res = 0.0;
for (int i=0; i<cs.Length; i++)
  res = cs[i] + x * res;
```

Benchmark results, polynomial of order 1000 (us/eval)

C (gcc -O3)	3.0
C# Microsoft	3.1
C# Mono	5.3
Java, Sun Hotspot -server	3.0

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

Benchmark results Normal distribution CDF N(0,1)

```
double expntl = Math.Exp(zabs * zabs * -.5);
double pdf = expntl / root2pi;
if (zabs < cutoff)
  p = expntl * ((((((p6 * zabs + p5) * zabs + p4) * zabs + p3) * zabs
        + p2) * zabs + p1) * zabs + p0) / (((((((q7 * zabs + q6) *
        zabs + q5) * zabs + q4) * zabs + q3) * zabs + q2) * zabs + q1)
         * zabs + q0);
else
  p = pdf / (zabs + 1 / (zabs + 2 / (zabs + 3 / (zabs + 4 / (zabs + .65)))));
```

- Widely used in probability and statistics

C (gcc -O3)	54
C# Microsoft	64
C# Mono	146
Java, Sun Hotspot -server	69

ns per function call

What's next

- Wed 2 March: Scheme, program generation
- Mon 7 March: Program specialization, and runtime code generation in (Java and) C#

- Hand-in for exercises 04 and 05 have been postponed by one week
  - Exercise 04: 11 March
  - Exercise 05: 18 March