Disclaimer: Many slides of this lecture are based on the slides of authors of the textbook from Elsevier. All copyrights reserved.
What are we talking about?

• Traditional process VMs are afterthoughts
• How about if we design a special guest ISA/system interface:
  – With portability as the main goal
  – define an abstract interface that can be supported by all conventional OSes.
  – Reflects important features of specific HLL or class of HLLs.
  – Simplifies compilation

This is why we call them HLL VMs
HLL VM is similar to Process VM BUT ...

- ISA defined for user-mode programs only
- ISA not designed for real hardware
  - Only to be executed on virtual processor
  - Referred to as virtual-ISA or V-ISA
- System interface is a set of standardized APIs
HLL VMs from language/compiler perspective

- **Goal:** complete **platform independence** for applications
- **Virtual instruction set** + libraries
  - Instead of ISA and OS interface

![Diagram]

- **Traditional HLL VM**
- **HLL VM**
P-Code VM
P-Code VM

Pascal Program → Compiler → P-code prog. → Virt. Mem. Image → P-code VM

Only one compiler has to be developed
Easier to implement than a compiler

P-code emulator
+ Standard lib for I/O
P-Code VM

• Popularized HLL VMs
• Provided highly portable version of Pascal
• Consists of
  – Primitive libraries
  – Machine-independent object file format (P-code)
  – A set of byte-oriented “pseudo-codes”
Memory Architecture Components

- Program memory area
  - indexed with PC
- Constant area
  - Values stored in this area are generated by the compiler
- Stack
  - Procedure stack
  - Operands for instruction execution
- Heap
  
All data areas are divided into cells
  - Each cell holds a single value
  - Size of a cell is implementation dependent
• Instruction set
  – Stack oriented
  – Stack “Frame” is part of VM definition

MP: Mark Pointer
EP: Extreme Pointer
NP: New Pointer
SP: Stack Pointer

lodi 0 3  // load variable from current frame (nest 0 depth),
// offset 3 from top of mark stack.
ldci 1  // push constant 1
addi  // add
stri 0 3  // store variable back to location 3 of current frame
A Closer Look at Stack Frame

- **Function value**: returns result value from the function, if any.
- **Static link**: linking statically linked procedures (Pascal allows nested procedures)
- **Dynamic link**: MP value of the previous frame
- **Operand stack**: holding operands and intermediate values when instructions are executed.
Basic ISA

- push/pop
- arithmetic
- logic
- shift

- instructions are typed:
  - example: add integer (adi),

Operate on the stack
What features do current HLL VMs share with P-Code VM?

• Stack instruction set → minimum number of registers
• Memory divided into implementation dependent cells
• Interface to OS through standard libraries
From P-code VM to modern HLL VMs

• Need to support networked computing environments
• Object-oriented programming paradigms
Modern HLL VM
Modern HLL VMs

• Superficially similar to P-code scheme
  – Stack-oriented ISA
  – Standard libraries

• Network Computing Environment
  – Security (this is the internet, after all)
  – Robustness (generally a good idea)
    • ⇒ object-oriented programming
  – Bandwidth is a consideration
  – Good performance must be maintained

• Two major examples
  – Java VM
  – Microsoft Common Language Infrastructure (CLI)
Modern HLL VMs

- Compiler forms program files (e.g. class files)
- Program files contain both code and metadata
Terminology

• **Java Virtual Machine Architecture ↔ CLI**
  – Analogous to an ISA
• **Java Virtual Machine Implementation ↔ CLR (Common Language Runtime)**
  – Analogous to a computer implementation
• **Java bytecodes ↔ Microsoft Intermediate Language (MSIL), CIL, IL**
  – The instruction part of the ISA
• **Java Platform ↔ .NET framework**
  – ISA + Libraries; a higher level ABI
5 Features of HLL VMs

- Platform-independence
- Security
- Robustness
- Networking
- Performance
Platform-Independence

• HLL VM is developed for each platform.
• Programs are compiled to meet the specifications for the VM.
  – Platform-independent code and metadata.
→ Programs can now run on all platforms that do have a VM.
Security

- A key aspect of modern network-oriented VMs
- Must protect:
  - Local files and resources
  - The VM itself
- The program runs in a sandbox at the host machine. It is managed by the VM runtime.
- The ability to load an untrusted application and run it in a managed secure fashion is a very big challenge!
Protection Sandbox

- **Remote resources**
  - Protected by remote system

- **Local resources**
  - Protected by security manager

- **VM software**
  - Protected via static/dynamic checking done by VM itself.
Robustness: Object-Oriention

- **Objects**
  - Data carrying entities
  - Dynamically allocated
  - Must be accessed via pointers or references

- **Methods**
  - Procedures that operate on objects

- **Class**
  - A type of object and its associated methods
  - Object created at runtime is an *instance* of the class
  - Data associated with a class may be *dynamic* or *static*

*OO programming paradigm has become the model of choice for modern HLL VMs.*
*Both Java and CLI are designed to support OO software.*
Robustness: More characteristics

- Strong-type checking: Relies to a high degree on static (checking) for protection, with minimal runtime checking.
- Garbage collection
Networking

• The application must use the available bandwidth (scarce) efficiently
  – Application loaded incrementally → dynamic linking
  – Improves program startup-time
Performance

• Of course we sacrifice some performance for the sake of portability
• Yet, we can use the techniques we learned so far (and some more as we proceed) to ensure good performance.
Case Study:
Java Virtual Machine (JVM)
Data items

- Types are defined, but no implementation details
  - Reference types (pointers): number of bits needed is not part of Java ISA.
  - Primitive types, e.g. int, char, byte, short, long, float, double,
  - Another primitive type: ReturnAddress (not in Java HLL but in Java ISA)

- Exact sizes of data types are not given
  - Only the range of values that can be held
    - e.g. byte is between -128 and +127
Objects and Arrays

• **Objects:**
  – Logical structure, defined by programmer, to carry data
  – Composed of primitive data types and references

• **Array**
  – Fixed number of elements
  – All elements must be of the same type
  – If the elements are references then they must all point to objects of the same type
Data Storage Types

• **Global**
  – the main memory
  – where globally declared variables reside

• **Local**
  – temporary storage
  – for variables local to a method

• **Operand**
  – holds variables while they are being operated on by functional instructions
Data Storage Types

• All storage is divided into cells or slots
• A cell/slot usually holds a single data item
• Actual amount of bits needed for cell/slot is implementation dependent
Stack

- Arguments
- Locals
- Operands

As each method is called, a stack frame is allocated.

In that order

Of fixed size determined at compile time
Global Memory

• Method area
  – for holding code

• Global storage area
  – for holding arrays and objects
  – managed as a heap
  – of unspecified size with respect to JVM architecture
  – Can contain both static and dynamic objects
Heap

- Objects are created on heap
- Each application has its own
- JVM instructions allocate objects on heap
  - No instruction to release memory
  - Garbage collection is part of implementation
- Object representation is implementation dependent
Constant Pool

- ISA allows constants to be expressed in the instruction as immediate operands
- Constant data associated with a program is placed in a block called constant pool
- Instructions access them by indexing constant pool
- Constant pool:
  - defined as part of the ISA
  - Exact size of constants is specified
  - Does not change with program execution
Putting it All Together: Memory Hierarchy in JVM

Instruction stream

Opcode operand operand
Opcode operand
Opcode operand operand
Opcode
Opcode operand
Opcode operand
Opcode operand

Index

Stack Frame

Locals

Operands

Constant Pool

Heap

Array

Object

Object

Object
Network Friendliness

• Support dynamic class file loading on demand
  – Load only classes that are needed
  – Spread loading out over time

• Compact instruction encoding
  – Use stack-oriented ISA (as in Pascal)
Garbage Collected Heap

• Objects are created and “float” in memory space
  – Tethered by references
  – In architecture, memory is unbounded in size
  – In reality it is limited

• Garbage creation
  – During program execution, many objects are created then abandoned (become garbage)

• Collection
  – Due to limited memory space, Garbage should be collected so memory can be re-used
  – Forcing programmer to explicitly free objects places more burden on programmer
    • Can lead to memory leaks, reducing robustness
  – To improve robustness, have VM collect garbage automatically
Instruction Set

- Stack based
- Defined for class file, not memory image
- Bytecodes
  - One byte opcode
  - Zero or more operands
    - Opcode indicates how many
- Can take operands from
  - Instruction
  - Current constant pool
  - Current frame local variables
  - Values on operand stack
    - Distinguish storage types and computation types
Implied Registers

• Program Counter
• Local variable pointer
• Operand stack pointer
• Current frame pointer
• Constant pool base
Instruction Types

• Pushing constants onto the stack
• Moving local variable contents to and from the stack
• Managing arrays
• Generic stack instructions (dup, swap, pop & nop)
• Arithmetic and logical instructions
• Conversion instructions
• Control transfer and function return
• Manipulating object fields
• Method invocation
• Miscellaneous operations
• Monitors
Data Movement

- All data movement takes place through stack
### Bytecode Example

<table>
<thead>
<tr>
<th>PC</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>iconst_2 //pushes constant 2 onto operand stack</td>
</tr>
<tr>
<td>1</td>
<td>aload_0 //pushes local variable 0 onto the stack</td>
</tr>
<tr>
<td>2</td>
<td>getfield #2; //object ref on the stack, entry 2 on constant pool gives descr</td>
</tr>
<tr>
<td>5</td>
<td>iconst_0</td>
</tr>
<tr>
<td>6</td>
<td>iaload</td>
</tr>
<tr>
<td>7</td>
<td>aload_0</td>
</tr>
<tr>
<td>8</td>
<td>getfield #2;</td>
</tr>
<tr>
<td>11</td>
<td>iconst_1</td>
</tr>
<tr>
<td>12</td>
<td>iaload</td>
</tr>
<tr>
<td>13</td>
<td>iadd</td>
</tr>
<tr>
<td>14</td>
<td>imul</td>
</tr>
<tr>
<td>15</td>
<td>ireturn</td>
</tr>
</tbody>
</table>
public int Abs(int i) {
    if (i < 0)
        return(i * -1);
    else
        return(i);
}

.method public Abs(I)I  // int argument, int result
    .limit stack 2       // stack with 2 locations
    .limit locals 2      // space for 2 locals

// --locals--  --stack---

    iload_1      // [ x -3 ]   [ -3 * ]
    ifge Labell  // [ x -3 ]   [ * * ]
    iload_1      // [ x -3 ]   [ -3 * ]
    iconst_m1    // [ x -3 ]   [ -3 -1 ]
    imul         // [ x -3 ]   [ 3 * ]
    ireturn      // [ x -3 ]   [ * * ]

Labell:
    iload_1
    ireturn
.end method

Comments show the execution  Abs(-3)

Stack Tracking

- Operand stack at any point in program must have:
  - Same number of operands
  - Of same types
  - In same order

  Regardless of control flow path getting there

- Helps with static type checking by the loader
Binary Classes: Code + metadata

- Magic number and header
- Major regions preceded by counts
  - Constant pool
  - Interfaces
  - Field information
  - Methods
  - Attributes
Binary Classes

- Holds all constant values and references used by the methods that are to follow.
- Provides access information, example:
  - whether public
  - whether interface
  - ...
- Given as indices in the constant pool
**Binary Classes**

- **Contains a number of references to the superinterfaces to this class**
  - Given as indices in the constant pool
  - The constant pool entries are references to the interfaces

- **Contains the specifications of the fields declared in this class**

- **The information regarding each method, as well as the methods themselves (encoded as bytecode)**

- **Contains detailed information regarding the previous sections**

<table>
<thead>
<tr>
<th>Magic Number</th>
<th>Version Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access Flags</th>
<th>This Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Super Class</th>
<th>Interface Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
A Note About: Garbage Collection

- Garbage: objects that are no longer accessible
- Examples: D, G, H
Garbage Collection

- A large topic on its own
- Mark and sweep
  - Start with root set of references
    - On stack, static objects, constant pool
  - Trace and mark all reachable objects
- Sweep through heap, collecting unmarked objects
  - Keep free space in linked list
    Advantage: Fast
  - Does not require moving object/pointers
- Disadvantage:
  - Discontiguous free space, fragmentation
  - Allocate new objects from best-fit free list
Compacting Collector

- Make free space contiguous
- Involves multiple passes through heap
- A lot of object movement => many pointer updates
Copying Collector

- Divide heap into halves
- Collect when one half full
- Copy into unused half during sweep phase
  + Reduces passes through heap
  - “Wastes” half of heap
Generational Collector

• Divide heap into halves
  – “tenured” and “nursery”
• Collect nursery more frequently
• Move long-lived objects into tenured half
• Objects have either very very long or very short lives
JVM Bytecode Emulation

• Interpretation
  – Simple, fast startup, but slow

• Just-In-Time (JIT) Compilation
  – Compile each method when first touched
  – Simple, static optimizations

• Hot-Spot Compilation
  – Find frequently executed code
  – Apply more aggressive optimizations on that code
  – Typically phased with interpretation or JIT

• Dynamic Compilation
  – Based on Hot-Spot compilation
  – Use runtime information to optimize
So JVM is:

• An abstract entity that gives meaning to class files

• Has many concrete implementations
  – Hardware
  – Interpreter
  – JIT compiler

• Persistence
  – An instance is created when an application starts
  – Terminates when the application finishes
Putting It All Together

- Class Loader Subsystem
- Memory
  - method area
  - heap
  - Java stacks
  - native method stacks
- Execution Engine
- Garbage Collector
  - native method libraries

- class files
- addresses
- data & instructions
- PCs & implied regs
Conclusions

• HLL VM is built with portability as main goal:
  – Building a loader and HLL VM is easier than building a full-fledged compiler
  – API and not ABI