# Computer Architecture

Week 2
CS 212 – Spring 2008

## Announcements

- Watch the website for announcements about section scheduling
- Part 1 (for both Compiler Project and GBA Project)
  - Will appear on the website later this week
  - Will be due on or about Friday, Feb 8

## Machine Language vs. Assembly Language

- Machine Language
  - Instructions and coding scheme used internally by computer
  - Humans do not usually write machine language
  - Typical machine language instructions have two parts
    - Op-code (operation code)
    - Operand
- Assembly Language
  - Symbolic representation of machine language
  - Use mnemonic words for op-codes
    - Example: PUSHIMM 5
  - Typically provide additional features to help make code readable for humans
    - Example: names as labels instead of numbers

## High-Level Language

- Idea: Use a program (a *compiler* or an *interpreter*) to convert high-level code into machine code
- Pro
  - Easier for humans to write, read, and maintain code
- Con
  - The resulting program will never be as efficient as good assembly-code
    - Waste of memory
    - Waste of time
- The whole concept was initially controversial
  - Thus, FORTRAN (mathematical FORmula TRANslating system) was designed with efficiency very-much in mind

## FORTRAN

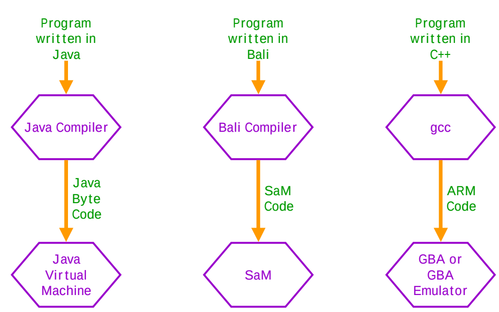
- Initial version developed in 1957 by IBM

- Example code

```
C     SUM OF SQUARES
      ISUM = 0
      DO 100 I=1,10
      ISUM = ISUM + I*I
  100 CONTINUE
```

- FORTRAN introduced many of the ideas typical of programming languages
  - Assignment
  - Loops
  - Conditionals
  - Subroutines

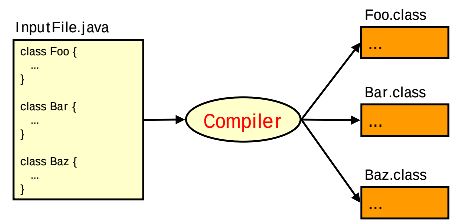
## Compiling & Running



## Java Byte Code (JBC)

- A Java compiler creates Java Byte Code (JBC)
  - A sequence of bytes
  - Platform-independent
  - Compact
    - Suitable for mobile code, applets
- JBC is machine code for a *virtual* (pretend) computer called the *Java Virtual Machine* (JVM)
  - A *byte code interpreter* reads and executes each instruction
- JBC is designed to be easy to interpret
  - Java virtual machine (JVM) in your browser
  - Simple stack-based semantics
  - Support for objects
- javap –c classfile
  - Can use this to see JBC

## Class Files




## What's in a Class File?

- Magic number, version info
- Constant pool
- Super class
- Access flags (public, private, ...)
- Interfaces
- Fields
  - Name and type
  - Access flags (public, private, static, ...)
- Methods
  - Name and signature (argument and return types)
  - Access flags (public, private, static, ...)
  - Bytecode
  - Exception tables
- Other stuff (source file, line number table, ...)

## Class File Format

| | | |
|---|---|---|
| magic number | 4 bytes | **0xCAFEBABE** |
| major version | 2 bytes | **0x0021** |
| minor version | 2 bytes | **0x0000** |

- Magic number identifies the file as a Java class file
- Version numbers inform the JVM whether it is able to execute the code in the file

## Constant Pool

| | |
|---|---|
| CP length | 2 bytes |
| CP entry 1 | (variable) |
| CP entry 2 | (variable) |
| ... | ... |

- Constant pool consists of up to 65536 = $2^{16}$ entries
- Entries can be of various types, thus of variable length

## Constant Pool Entries

| | |
|---|---|
| Utf8 (unicode) | literal string (2 bytes for length, characters) |
| Integer | Java int (4 bytes) |
| Float | Java float (4 bytes) |
| Long | Java long (8 bytes) |
| Double | Java double (8 bytes) |
| Class | class name |
| String | String constant -- index of a Utf8 entry |
| Fieldref | field reference -- name and type, class |
| Methodref | method reference -- name and type, class |
| InterfaceMethodref | interface method reference |
| NameAndType | Name and Type of a field or method |

# Example

```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many entries in the constant pool?

A) 33

```
1)CONSTANT_Methodref[10](class_index = 6, name_and_type_index = 20)
2)CONSTANT_Fieldref[9](class_index = 21, name_and_type_index = 22)
3)CONSTANT_String[8](string_index = 23)
4)CONSTANT_Methodref[10](class_index = 24, name_and_type_index = 25)
5)CONSTANT_Class[7](name_index = 26)
6)CONSTANT_Class[7](name_index = 27)
7)CONSTANT_Utf8[1]("<init>")
8)CONSTANT_Utf8[1]("()V")
9)CONSTANT_Utf8[1]("Code")
10)CONSTANT_Utf8[1]("LineNumberTable")
11)CONSTANT_Utf8[1]("LocalVariableTable")
12)CONSTANT_Utf8[1]("this")
13)CONSTANT_Utf8[1]("LFoo;")
14)CONSTANT_Utf8[1]("main")
15)CONSTANT_Utf8[1]("([Ljava/lang/String;)V")
16)CONSTANT_Utf8[1]("args")
17)CONSTANT_Utf8[1]("[Ljava/lang/String;")
18)CONSTANT_Utf8[1]("SourceFile")
19)CONSTANT_Utf8[1]("Foo.java")
20)CONSTANT_NameAndType[12](name_index = 7, signature_index = 8)
21)CONSTANT_Class[7](name_index = 28)
22)CONSTANT_NameAndType[12](name_index = 29, signature_index = 30)
23)CONSTANT_Utf8[1]("Hello world")
24)CONSTANT_Class[7](name_index = 31)
25)CONSTANT_NameAndType[12](name_index = 32, signature_index = 33)
26)CONSTANT_Utf8[1]("Foo")
27)CONSTANT_Utf8[1]("java/lang/Object")
28)CONSTANT_Utf8[1]("java/lang/System")
29)CONSTANT_Utf8[1]("out")
30)CONSTANT_Utf8[1]("Ljava/io/PrintStream;")
31)CONSTANT_Utf8[1]("java/io/PrintStream")
32)CONSTANT_Utf8[1]("println")
33)CONSTANT_Utf8[1]("(Ljava/lang/String;)V")
```

# Code Attribute of a Method

| | | |
|---|---|---|
| maxStack | 2 bytes | max operand stack depth |
| maxLocals | 2 bytes | number of local variables |
| codeLength | 2 bytes | length of bytecode array |
| code | codeLength | the executable bytecode |
| excTableLength | 2 bytes | number of exception handlers |
| exceptionTable | excTableLength | exception handler info |
| attributesCount | 2 bytes | number of attributes |
| attributes | variable | e.g. LineNumberTable |

# Example Bytecode

```
if (b) x = y + 1;
else x = z;
```

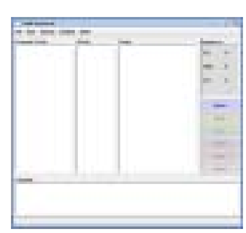
```
5:  iload_1        //load b
6:  ifeq 16        //if false, goto else
9:  iload_3        //load y        }
10: iconst_1       //load 1        }
11: iadd           //y+1           } then clause
12: istore_2       //save x        }
13: goto 19        //skip else     }
16: iload_4        //load z        } else clause
18: istore_2       //save x        }
19: ...
```

# Java Virtual Machine (JVM)

- JBC is code for the JVM
  - No such machine really exists
  - A *JVM interpreter* must be created for each machine architecture on which JBC is to run
- The JVM is designed as an "average" computer
  - Uses features that are widely available (e.g., a stack)
- Design goals
  - Should be easy to convert Java code into JBC
  - Should be reasonably easy to create a JVM interpreter for most computer architectures

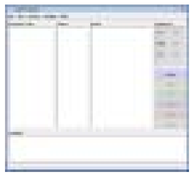
# SaM (Stack Machine)

- Goals
  - Approximate the JVM
  - But simpler
- We produce sam-code, assembly language for SaM, our own virtual machine
- We have a SaM Simulator (thanks to David Levitan) that we can use to execute sam-code
- In place of
  - JBC for the JVM
- We produce
  - sam-code for SaM



# SaM Design

- Three memory areas
  - Program Code
    - Holds your sam-code
  - Stack
    - Work area while program runs
    - Contains local variables and return-information for all currently active functions
  - Heap
    - Used to store arrays & objects
- Three registers
  - PC (Program Counter)
    - Location of current instruction
  - FBR (Frame Base Register)
    - Location (on Stack) of data for currently running function
  - SP (Stack Pointer)
    - Current top of Stack



# Some Sam-Code Instructions

- SaM's main memory is maintained as a Stack
- The SP (stack pointer) register points at the next empty position on the stack
  - The first position has address 0
  - Addresses increase as more items are pushed onto the Stack
- PUSHIMM c
  - (push immediate)
  - Push integer c onto Stack
- ADD
  - Add top two Stack items, removing those items, and pushing result onto Stack
- SUB
  - Subtract top two Stack items, removing those items, and pushing result onto Stack
  - Order is important
    - stack[top-1] – stack[top]

# More Sam-Code Instructions

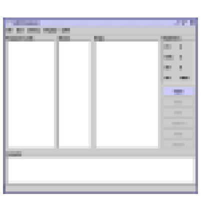
- ALU Instructions
  - ADD, SUB, TIMES, DIV
  - NOT, OR, AND
  - GREATER, LESS, EQUAL
- Stack Manipulation Instructions
  - PUSHIMM c
  - DUP, SWAP
  - PUSHIND
    - (push indirect)
    - Push stack[stack[top]] onto Stack
  - STOREIND
    - (store indirect)
    - Store stack[top] into stack[stack[top-1]]

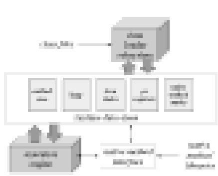
# Machine Instruction Categories

- Data transfer
  - Copy data from one memory location to another
    - LOAD: copy data from a memory cell to a register
    - STORE: copy data from a register to a memory cell
    - I/O instructions
- Arithmetic / Logic
  - Request activity in ALU
    - Arithmetic (ADD, SUB, TIMES, ...)
    - Logic (AND, OR, NOT, XOR)
    - SHIFT, ROTATE
- Control
  - Direct the execution of the program
    - JUMP, JUMPC (conditional jump)

# Runtime Data Areas

- For SaM
  - Code
  - Stack
  - Heap
  - Registers
- For Java
  - Method area
  - Java stacks
  - Heap
  - PC registers
  - Native method stacks





from: http://www.artima.com/insidejvm/ed2/jvm2.html

# JVM Runtime Data Areas

- Method area (stores data for each type)
  - Information about the type (e.g., name, modifiers, superclass, etc.)
  - *Constant pool* for the type
    - Any constant used in the type's code (e.g., 5 or 'x' or 1.414)
  - Field & method information for the type (including the *code* for each method)
  - *Class variables* (i.e., *static* fields)
- Java stacks
  - Stores *stack frames*
  - But keeps *multiple* stacks because Java is *multithreaded*
- Heap
  - Stores objects (including *instance variables*)
- PC registers
  - One PC register for each *thread*
- Native method stacks
  - A work area for methods written in a language other than Java

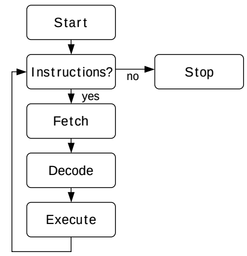
# GBA Runtime Data Areas

Memory Map (Simplified)

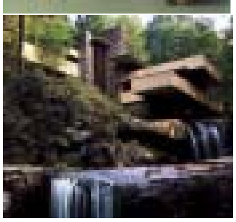
| |
|---|
| BIOS |
| Work RAM |
| Control Registers |
| Palettes |
| Video Display |
| Sprites |
| Game Code |

- BIOS (Basic Input/Output System)
  - System stuff; normally inaccessible
- Work RAM
  - Workspace; variables are stored here
- Control Registers
  - Setting these alters the way the game is displayed
- Palettes
  - Used to compactly represent colors
- Video display
  - Data here is displayed on the game-screen
- Sprites
  - These are small images that can be layered on top of the video display
- Game code

# Fetch and Decode Cycle




- Control Unit (CU) fetches next instruction from memory at address specified by Program Counter (PC)
- CU places instruction into the instruction register (IR)
- CU increments PC to prepare for next cycle
- CU decodes instruction to see what to do
- CU activates correct circuits to execute the instruction (e.g., ALU performs an addition)



# Computer Architecture

Week 2
CS 212 – Spring 2008