JavaOne℠

Sun's Worldwide Java Developer Conference

JavaOne

Sun's Worldwide Java Developer Conference

picoJava™: The Java Virtual Machine in Hardware

Marc Tremblay
Chief Architect
Sun Microelectronics

The Java™ – picoJava Synergy

- **Java's origins lie in improving the consumer embedded market**
- **picoJava is a low cost microprocessor dedicated to executing Java™-based bytecodes**
  - Best system price/performance
- **It is a processor core for:**
  - Network computer
  - Internet chip for network appliances
  - Cellular phone & telco processors
  - Traditional embedded applications

Java in Embedded Devices

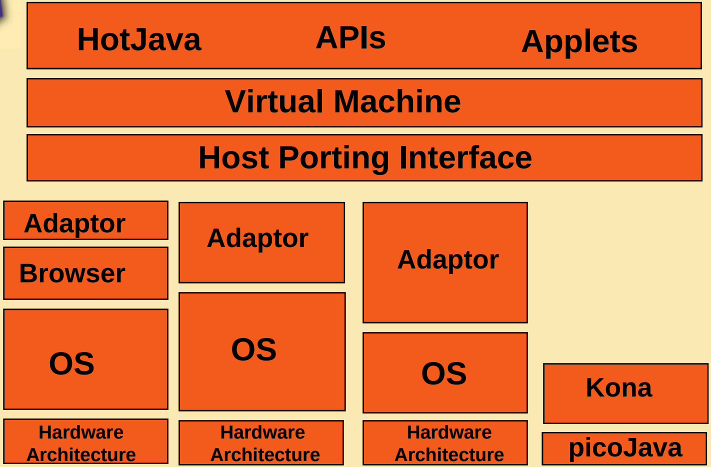
Products in the embedded market require:

- Robust programs
  - Graceful recovery vs. crash
- Increasingly complex programs with multiple programmers
  - Object-oriented language and development environment
- Re-using code from one product generation to the next
  - Portable code

Important Factors to Consider in the Embedded World

- Low system cost
  - Processor, ROM, DRAM, etc.
- Good performance
- Time-to-market
- Low power consumption

Various Ways of Implementing the Java Virtual Machine



picoJava

- Directly executes bytecodes
  - Excellent performance
  - Eliminates the need for an interpreter or a JIT compiler
  - Low memory footprint
- Simple core
  - Legacy blocks and circuits are not present
- Hardware support for the runtime
  - Addresses overall system performance

Java Virtual Machine

- What the virtual machine specifies:
  - Instruction set
  - Data types
  - Operand stack
  - Constant pool
  - Method area
  - Garbage collected heap for runtime data area

Java Virtual Machine Code Size

- Java™-based bytecodes are small
  - No register specifiers
  - Implicit "VARS" register for local variable accesses
- This results in very compact code
  - Average JVM instruction is 1.8 bytes
  - RISC instructions typically require 4 bytes

Java Virtual Machine Code Size (cont.)

- A large application (2500+lines) coded in both the C++ and Java languages:
  - Java bytecodes are 2-3x smaller than the RISC code from the C++ compiler

Virtual Machine — Instruction Set

- Data types: byte, short, int, long, float, double, char, object, returnAddress
- All opcodes have 8 bits, but are followed by a variable number of operands(0, 1, 2, 3, …)
- Opcodes
  - 200 assigned
  - 24 quick variations
  - 2 reserved

JVM – Instruction Set – RISCy

- Some instructions are “RISCy”:

```
bipush value    :push signed integer
iadd            :integer add
fadd            :single float add
ifeq            :branch if equal to 0
iload offset    :load integer from
                :local variable
```

JVM – Instruction Set – CISCy

- Some instructions are "CISCy":

`lookupswitch`: "traditional" switch statement

byte 1	byte 2	byte 3	byte 4
opcode (171)	0..3 byte padding		
default offset			
numbers of pairs that follow (N)			
match 1			
jump offset 1			
match 2			
jump offset 2			
...			
...			
match N			
jump offset N			

Interpreter Loop

```
loop: 1: fetch bytecodes
      2: indirect jump to
         emulation code
```

→

Emulation Code

```
1: get operands
2: perform operation
3: increment PC
4: go to loop
```

JVM: Stack-Based Architecture

- Operands typically accessed from the stack, put back on the stack
- Example — integer add:
  - Add top 2 entries in the stack and put the result on top of the stack
  - Typical emulation on a RISC processor

```
1: load tos
2: load tos-1
3: add
4: store tos-1
```

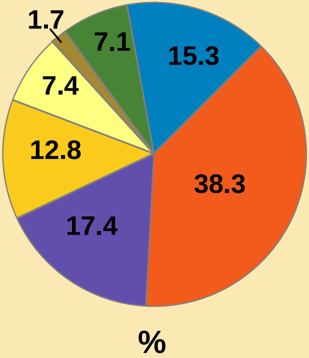
How to Best Execute Bytecodes?

- Leverage RISC techniques developed over the past 15 years
- Implement in hardware only those instructions that make a difference
  - Trap for costly instructions that do not occur often

How to Best Execute Bytecodes? (cont.)

- Base clock rate on fundamental 32-bit adder
  - Pipeline instructions
  - Single cycle execution for most instructions
- Stack architecture implemented as a RISC

Dynamic Instruction Mix



loads_loc	Loads from local variables
loads_mem	Loads from constant pool, objects' field, arrays, etc.
Stores	3% to memory, 9.8% to local variables
ALU	Add, subtract, booleans, shifts
FP	Mul, add, subtract, compare
Stack	Dup, constant push, swap
Branch	Invoke methods, branches, returns, jumps

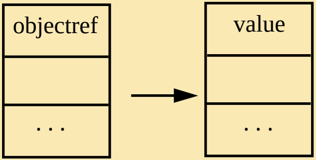
Implementation of Important Instructions

`getfield_quick offset`

– Fetch field from object

– Replaces getfield

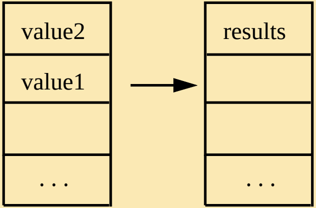
– Executes as a “load [object + offset]” on picoJava



`isub`

– Fully pipelined

– Executes in a single cycle



New Paradigm —> New Processors

- Early RISC processors were designed for C and Fortran; benchmarks were Dhrystone, Hanoi, SPEC89, etc.
- New applications may dictate new instructions or new hardware support
- For example: multimedia applications of the '90's led to the creation of new multimedia instructions (UltraSPARC's VIS and X86's MMX)

New Paradigm —> New Processors (cont.)

- The proliferation of the Java language in the embedded market
  - —> Lean processors dedicated to executing bytecodes
- Java Runtime (gc.c, monitor.c, threadruntime.c, etc.)
  - Significant time spent synchronizing threads
  - Significant time spent for memory management
    - —> On-chip support reduces overhead

picoJava:
A System Performance Approach

- Accelerates runtime
  - Support for threads
  - Support for garbage collection
- Simple but efficient, non-invasive, hardware support

picoJava

Best system price/performance for running Java™-powered applications in embedded markets

- **Embedded market very sensitive to system cost**
- **picoJava eliminates interpreter or JIT compiler**
- **Excellent system performance**
- **Efficient implementation through use of the same methodology, process and circuit techniques developed for our RISC processors**

picoJava

- Licensing now
- Stay tuned for more information
  - Hot Chips
  - MicroProcessor Forum