Computer Organization
CS 231-01

ISA Examples

Dr. William H. Robinson
December 3, 2004

http://eecs.vanderbilt.edu/courses/cs231/

Topics

“Are you mad 'cause I'm asking you 21 questions?”

– 50 Cent

• Administrative stuff
  – Programming assignment #3

• Instruction Set Architecture implementations
  – Pentium II
  – UltraSPARC II
  – JVM

Program 3

• Extend the IJVM instruction set
  – INEG
  – ISHL (Value 1 shifted by low 5 bits of Value 2)
  – ISHR (Value 1 shifted by low 5 bits of Value 2)
  – IUSHR (Value 1 shifted by low 5 bits of Value 2)
  – IXOR

• Must program in microcode
  – Remember that the Mic-1 *interprets* instructions

IJVM Examples

• Data movement
  – Ex: ILOAD, ISTORE

• Dyadic (two operands)
  – Ex: IADD, ISUB, IMUL

• Monadic (one operand)
  – Ex: DUP, ISHR, ISHL

• Comparisons and conditional branches
  – Ex: IFEQ, IFLT

• Procedure calls
  – Ex: INVOKEVIRTUAL
Instruction Set Architecture Level

- Interface between software and hardware
- Both compilers and hardware must understand ISA
  - Compilers translate high-level language into object code
  - Hardware must directly execute or interpret ISA

Properties of ISA

- ISA level defined by how the machine appears to the machine language programmer (compiler)

  - The compiler must know
    - Data types
    - Instructions
    - Memory model
    - Registers

Properties of ISA

- Technically, the implementation of the ISA (microarchitecture level) should not be visible

- Techniques like pipelining or superscalar can affect performance which is visible to the compiler
  - Sometimes the compiler can generate code that takes advantage of the underlying H/W implementation

Registers

- Some registers in microarchitecture are visible at the ISA
- Some registers in microarchitecture are not visible at the ISA
- Registers visible in ISA are always visible at the microarchitecture level
Registers

- **Two types**
  - **Special purpose**
    - Have defined function
    - Ex: Program Status Word from JVM
  - **General purpose**
    - Hold key local variables
    - Provide rapid access to data

- Operating system or compiler will typically adopt conventions for using general purpose registers

ISA Examples

- **IA-32**
  - Pentium II
  - CISC

- **Version 9 SPARC**
  - UltraSPARC II
  - RISC

- **Java Virtual Machine**
  - picoJava II
  - Stack machine

Pentium II (IA-32)

- **Testament to backward compatibility**

- **Three operating modes**
  - Real
  - Virtual 8086
  - Protected

- **Memory model**
  - Little endian
  - Effective linear address space of \(2^{32}\) bytes

Pentium II Registers

- **General purpose**
  - EAX, EBX, ECX, EDX

- **Pointers**
  - ESI, EDI, EBP, ESP

- **8088 segmentation**
  - CS, SS, DS, ES, FS, GS

- **Program counter**
  - EIP

- **Program Status Word**
  - EFLAGS
RISC Design Principles

- All instructions directly executed by H/W
  - Eliminate overhead of interpretation
- Maximize rate instructions are issued
  - Utilize parallelism within program
- Instructions should be easy to decode
  - Quickly determine resources required
- Only loads and stores should reference memory
  - Memory access is longer and unpredictable
- Provide plenty of registers
  - Avoid memory access penalty for flushing data set

UltraSPARC II (Version 9 SPARC)

- Example of a clean RISC design
- Load/Store architecture
  - All operands for instructions are located in registers
  - Only LOAD and STORE access memory
- Memory model
  - Big endian by default, but can switch to little endian by setting a bit in the PSW
  - Linear array of $2^{64}$ bytes
    - Built in future memory expansion unlike Intel

UltraSPARC II Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Alt. name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>G0</td>
<td>Hardwired to 0. Stores into it are just ignored.</td>
</tr>
<tr>
<td>R1 – R7</td>
<td>G1 – G7</td>
<td>Holds global variables</td>
</tr>
<tr>
<td>R8 – R13</td>
<td>O0 – O5</td>
<td>Holds parameters to the procedure being called</td>
</tr>
<tr>
<td>R14</td>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>R15</td>
<td>O7</td>
<td>Scratch register</td>
</tr>
<tr>
<td>R16 – R23</td>
<td>L0 – L7</td>
<td>Holds local variables for the current procedure</td>
</tr>
<tr>
<td>R24 – R29</td>
<td>I0 – I5</td>
<td>Holds incoming parameters</td>
</tr>
<tr>
<td>R30</td>
<td>FP</td>
<td>Pointer to the base of the current stack frame</td>
</tr>
<tr>
<td>R31</td>
<td>I7</td>
<td>Holds return address for the current procedure</td>
</tr>
</tbody>
</table>

- Some complexity of register organization
  - For passing parameters in procedure calls
- Two groups of registers
  - 32 64-bit general purpose registers
  - 32 floating-point registers

Java Virtual Machine

- Pure stack architecture
- No general-purpose registers
- Large number of memory references
  - Many are eliminated by folding multiple JVM instructions together
JVM Memory Model

- Same as IJVM but also includes the **heap**
  - Garbage collector looks for objects on heap no longer used
- Big endian

Numeric Data Types

- **Integers**
  - Signed and unsigned
  - Lengths of 8, 16, 32, and 64 bits

- **Floating point**
  - Follow IEEE Standard 754
  - Lengths of 32, 64, and 128 bits

- **Decimal**
  - Binary coded decimal for COBOL
  - Pack 2 decimal values per byte

Non-Numeric Data Types

- **Characters**
  - ASCII
  - UNICODE

- **Boolean**
  - Zero means FALSE
  - Everything else means TRUE

- **Pointers**
  - Machine address

Pentium II Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>8 Bits</th>
<th>16 Bits</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integer</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 5-6. The Pentium II numeric data types. Supported types are marked with x.*

**Non-numeric**

- Special instructions for copying and searching character strings
  - ASCII characters
UltraSPARC II Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>8 Bits</th>
<th>16 Bits</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integer</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-7. The UltraSPARC II numeric data types.

Non-numeric
- Character and string data types are not supported by special H/W instructions

JVM Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>8 Bits</th>
<th>16 Bits</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integer</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-8. The JVM numeric data types.

Non-numeric
- Supports UNICODE characters

Generic Examples: Instruction Format Widths

Variable:

- ...
- ...

Fixed:

- ...

Hybrid:

- ...

Instruction Formats

- (a) Zero-address instruction
- (b) One-address instruction
- (c) Two-address instruction
- (d) Three-address instruction

Figure 5-9. Four common instruction formats: (a) Zero-address instruction, (b) One-address instruction, (c) Two-address instruction, (d) Three-address instruction.

- Consists of OPCODE and additional information like operand locations and destination
- An expanding opcode can provide a trade-off between opcode bits and address bits (Section 5.3.2)
Instruction Formats

- **Fixed length**
  - Faster decoding
  - Wastes space

- **Variable length**
  - Slower decoding
  - More efficient code size

Instruction Format Design Criteria

- Short instructions are better than long ones
- Sufficient room to express all desired operations, including future expansion/addition
- Number of bits in address field

Pentium II

UltraSPARC II

Figure 5-13. The Pentium II instruction formats.

Figure 5-14. The original SPARC instruction formats.
Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 &lt;- R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 &lt;- R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 &lt;- R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 &lt;- R4+Mem[R1]</td>
</tr>
<tr>
<td>Base-Indexed</td>
<td>Add R3,(R1+R2)</td>
<td>R3 &lt;- R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 &lt;- R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 &lt;- R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Post-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 &lt;- R1+Mem[R2]; R2 &lt;- R2+d</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>Add R1,–(R2)</td>
<td>R2 &lt;- R2–d; R1 &lt;- R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 &lt;- R1+Mem[100+R2+R3*d]</td>
</tr>
</tbody>
</table>

Adapted from John Kubiatowicz’s CS 152 lecture notes. Copyright © 2003 UCB.

Addressing Mode Usage? (ignore register mode)

3 programs measured on machine with all address modes (VAX)

- Displacement: 42% avg, 32% to 55% 75%
- Immediate: 33% avg, 17% to 43% 85%
- Register deferred (indirect): 13% avg, 3% to 24%
- Scaled: 7% avg, 0% to 16%
- Memory indirect: 3% avg, 1% to 6%
- Misc: 2% avg, 0% to 3%

On average
- 75% displacement & immediate
- 85% displacement, immediate & register indirect

Adapted from John Kubiatowicz’s CS 152 lecture notes. Copyright © 2003 UCB.
Addressing Modes

- **Based-indexed**
  - Adds two registers plus an optional offset

- **Direct or absolute**
  - Give full memory address of operand

- **Stack**
  - Memory referenced with a Last-In First-Out (LIFO) queue

---

Three Bears Theory of Computer Science

- **The core of a modern computer**
  - Deeply-pipelined
  - Three-register load/store RISC engine

- **Some computer instructions are too big**
  - Pentium II (IA-32 ISA)

- **Some computer instructions are too small**
  - picoJava II (JVM ISA)

- **Some computer instructions are just right**
  - UltraSPARC II (Version 9 SPARC)

---

### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Pentium II</th>
<th>UltraSPARC II</th>
<th>JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Direct</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Register indirect</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Based-indexed</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 5.28. A comparison of addressing modes.*