Topics

• Overview of ISA
  – Tanenbaum 5.1

• Data types
  – Tanenbaum 5.2

• Instruction formats
  – Tanenbaum 5.3

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
Designing the ISA

**In theory…**
- Optimized for the compiler writers
  - Compilers can generate effective object code

**And**
- Optimized for the hardware designers
  - H/W can be implemented to deliver good performance and good performance/price

---

**In practice…**
- Is the ISA backward compatible?
  - You can add instructions and features
  - Customers do not want to throw away legacy code

- Examples
  - Intel processors still execute 8086/8088 code
  - Playstation and Playstation 2
Why Design a Good ISA?

• Can last for a long time
  – Customers will want backward compatibility

• Important for performance considerations
  – Execute programs quickly

• Important for hardware implementations
  – Manufacturing cost

• Important for emerging domains/markets
  – Embedded systems, multimedia processors

Factors for a Good ISA

Define a set of instructions that can be implemented efficiently in current and future technologies

• Technology trends
  – IC logic technology
    • Improves transistor density, die size, device speed
  – Semiconductor DRAM
    • Higher densities increases memory capacity
  – Magnetic disk technology
    • Larger hard drives and improved access times
  – Network technology
    • Improved switches and transmission systems
Factors for a Good ISA

Provides a clean target for compiled code

• Regularity and completeness
  – Provide a range of alternatives
  – Easy to generate good code for high-level language
  – Compiler must make the best choice among alternatives

Properties of ISA

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

• The compiler must know
  – Memory model
  – Registers
  – Data types
  – Instructions
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

ISA Definition

- An ISA can be defined in a formal document
  - Version 9 SPARC
  - Java Virtual Machine (JVM)
- Allows different implementers to build the machine
  - Have them all run exactly the same S/W and get exactly the same results
  - Differ only in performance and price
- Two types of sections
  - **Normative**: impose requirements
  - **Informative**: helps the reader
Modes of Operation of ISA

- **Kernel**
  - Run operating system
  - Allows all instructions to be executed

- **User**
  - Runs applications
  - Prevents certain sensitive instructions from being executed

Memory Models

- **Use byte as the basic cell**
  - One byte contains 8 bits
  - Convenient for ASCII which uses 7 bits
    - One ASCII character plus a parity bit per byte

- **Potential for a 16-bit cell**
  - If UNICODE becomes the dominant character code

- **Generally group bytes into 32-bit or 64-bit words**
  - Manipulate the entire word
Word Alignment

- Many architectures require word alignment for efficient memory access
- Having the capability to read words at arbitrary addresses requires extra H/W
  - Increases cost
  - Still must support some legacy code

![Figure 5-2. An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.](image)

Byte Ordering within Words

- Big Endian – numbering begins at higher-order end
- Little Endian – numbering begins at lower-order end

Problems occur when one system sends data to another over a network
Memory Address Space

• Most machines have linear address space
  – From 0 to some maximum # of bytes

• Some have separate address space for instructions and data
  – More complex
  – Not the same as a separate I-cache and D-cache
  – Possible for $2^n$ bytes for both program and data with only $n$ address lines
  – Impossible to accidentally overwrite program

Memory Semantics

• Ensuring memory behaves as expected

• One extreme – all requests are serialized
  – Hurts performance

• Other extreme – no guarantees
  – Requires SYNC instructions from compiler

• Intermediate – block RAW or WAR dependencies
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
  - General purpose
    • Hold key local variables
    • Provide rapid access to data

- Operating system or compiler will typically adopt conventions for using general purpose registers
Program Status Word (PSW)

ISA register containing status flags (p. 310)

- N – Set when the result is Negative
- Z – Set when the result is Zero
- V – Set when the result caused an overflow
- C – Set when the result caused a Carry out of MSB
- P – Set when the result had even Parity

Instructions

- Control what the machine can do
- Always have LOAD and STORE to move data between memory and registers
- Always have MOVE instructions to copy data among registers
- Also include arithmetic, Boolean, comparison and branching on results
IJVM Instruction Set

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>BIPUSH byte</td>
<td>Push byte onto stack</td>
</tr>
<tr>
<td>0x59</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0xA7</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0xB0</td>
<td>IADD</td>
<td>Pop two words from stack; push their sum</td>
</tr>
<tr>
<td>0x7E</td>
<td>IAND</td>
<td>Pop two words from stack; push Boolean AND</td>
</tr>
<tr>
<td>0x99</td>
<td>IFEQ offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x98</td>
<td>IFLT offset</td>
<td>Pop word from stack and branch if less than zero</td>
</tr>
<tr>
<td>0x9F</td>
<td>IF_ICMPGE offset</td>
<td>Pop two words from stack; branch if equal</td>
</tr>
<tr>
<td>0x84</td>
<td>INVC varnum const</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x10</td>
<td>ILADD varnum</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x86</td>
<td>INVOKEVIRTUAL disp</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x80</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0xAC</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x36</td>
<td>ISTORE varnum</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0x44</td>
<td>ISUB</td>
<td>Pop two words from stack; push their difference</td>
</tr>
<tr>
<td>0x13</td>
<td>LDC_W index</td>
<td>Push constant from constant pool onto stack</td>
</tr>
<tr>
<td>0x00</td>
<td>NOP</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0x67</td>
<td>POP</td>
<td>Delete word on top of stack</td>
</tr>
<tr>
<td>0x6F</td>
<td>SWAP</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0xC4</td>
<td>WIDE</td>
<td>Prefix instruction; next instruction has a 16-bit index</td>
</tr>
</tbody>
</table>

Figure 4-11. The IJVM instruction set. The operands byte, const, and varnum are 1 byte. The operands disp, index, and offset are 2 bytes.

ISA Examples

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

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

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

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 – R7</td>
<td>G0 – G7</td>
<td>Hardwired to 0. Stores into it are just ignored.</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>10 – 15</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

Data Types

- Some data types are not supported by H/W
  - Double precision uses two words to represent one number
- Two categories of H/W supported data types
  - Numeric
  - Non-numeric
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>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**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>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
<td>×</td>
<td>×</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>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-8. The JVM numeric data types.

Non-numeric

- Supports UNICODE characters

Instruction Formats

(a) OPCODE

(b) OPCODE ADDRESS

(c) OPCODE ADDRESS1 ADDRESS2

(d) OPCODE ADDR1 ADDR2 ADDR3

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
Instruction Formats

- Fixed length
  - Faster decoding
  - Wastes space
- Variable length
  - Slower decoding
  - More efficient code size

Figure 5-10. Some possible relationships between instruction and word length.

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

Figure 5-13. The Pentium II instruction formats.

UltraSPARC II

Figure 5-14. The original SPARC instruction formats.
Figure 5-15. The JVM instruction formats.