Topics

“Only those who will risk going too far can possibly find out how far one can go.”

– T. S. Eliot
(1888 - 1965)

• Administrative stuff
  – Return graded assignments
  – Blackboard
  – Programming assignment #3

• Instruction Set Architecture design

Announcing Program 3

• Extend the IJVM instruction set
  – INEG
  – ISHL
  – ISHR
  – IUSHR
  – IXOR

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

Designing the ISA

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

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**
  - Runs the operating system
  - Allows all instructions to be executed

- **User**
  - Runs applications
  - Prevents certain sensitive instructions from being executed
ISA: What Must be Specified?

- Instruction format or encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which operands can or cannot be in memory?
- Data type and size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - fetch-decode-execute is implicit!

Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>96%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simple instructions dominate instruction frequency

Typical Operations (little change since 1960)

Data Movement
- Load (from memory)
- Store (to memory)
- Memory-to-memory move
- Register-to-register move
- Input (from I/O device)
- Output (to I/O device)
- Push, pop (to/from stack)

Arithmetic
- Integer (binary + decimal) or FP
- Add, Subtract, Multiply, Divide

Shift
- Shift left/right, rotate left/right

Logical
- Not, and, or, set, clear

Control (Jump/Branch)
- Unconditional, conditional

Subroutine Linkage
- Call, return

Interrupt
- Trap, return

Synchronization
- Test & set (atomic r-m-w)

String
- Search, translate

Graphics (MMX)
- Parallel subword ops (4 16-bit add)

Operation Summary

Support these simple instructions, since they will dominate the number of instructions executed:

- Load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch, jump, call, return;
Compilers and ISA

• **Ease of compilation**
  --orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
  --completeness: support for a wide range of operations and target applications
  --regularity: no overloading for the meanings of instruction fields
  --streamlined: resource needs easily determined

• **Register assignment is critical too**
  --Easier if lots of registers

General Purpose Registers Dominate

* 1975-2000 all machines use general purpose registers

* Advantages of registers
  • Registers are faster than memory
  • Registers are easier for a compiler to use
    - e.g., \((A^*B) - (C^*D) - (E^*F)\) can do multiplies in any order vs. stack
  • Registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

Basic ISA Classes

Most real machines are hybrids of these:

Accumulator (1 register):
- 1 address: add A   acc ← acc + mem[A]
- 1+x address: addx A  acc ← acc + mem[A + x]

Stack:
- 0 address: add  tos ← tos + next

General Purpose Register (can be memory/memory):

Load/Store:
- 3 address: add Ra Rb Rc  Ra ← Rb + Rc
- load Ra Rb  Ra ← mem[Rb]
- store Ra Rb  mem[Rb] ← Ra

Comparison:
- Bytes per instruction?
- Number of Instructions?
- Cycles per instruction?
Memory Operands per Instruction

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands allowed</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia CPU64</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Intel 80x86, Motorola 68000, TI TMS320C54x</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>VAX (also has 3-operand formats)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>VAX (also has 2-operand formats)</td>
</tr>
</tbody>
</table>

- RISC machines tend to have only register operands + Load/Store

Memory Addressing

Since 1980 almost every machine uses addresses to level of 8-bits (byte)

2 questions for design of ISA:

- Since you could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto words?

- Can a word be placed on any byte boundary?

Addressing Objects: Endianness and Alignment

- **Big Endian**: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian**: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

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 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 100*(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4, (R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Base-Indexed</td>
<td>Add R3, (R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1, (1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1, @ (R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Post-increment</td>
<td>Add R1, (R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>Add R1, -(R2)</td>
<td>R2 ← R2-d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1, 100*(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2<em>R3</em>d]</td>
</tr>
</tbody>
</table>

**Why Post-increment/Pre-decrement? Scaled?**
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.

Summary

- ISA is the common language between the compiler and the hardware

- ISA must support/define
  - Instruction format or encoding
  - Location of operands and result
  - Data type and size
  - Operations
  - Successor instruction