Lecture 3: Instruction Set Architecture

ISA types, register usage, memory addressing, endian and alignment, quantitative evaluation

What Is ISA?

Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine.

For IBM System/360, 1964

- Class ISA types: Stack, Accumulator, and General-purpose register
- ISA is mature and stable
  - Why do we study it?

Stack

- Implicit operands on stack
- Ex. \( C = A + B \)
  - Push A
  - Push B
  - Add
  - Pop C
- Good code density; used in 60's-70's; now in Java VM

Accumulator

- The accumulator provides an implicit input, and is the implicit place to store the result.
- Ex. \( C = A + B \)
  - Load R1, A
  - Add R3, R1, B
  - Store R3, C
- Used before 1980

General-purpose Registers

- General-purpose registers are preferred by compilers
  - Reduce memory traffic
  - Improve program speed
  - Improve code density
- Usage of general-purpose registers
  - Holding temporal variables in expression evaluation
  - Passing parameters
  - Holding variables
- GPR and RISC and CISC
  - RISC ISA is extensively used for desktop, server, and embedded: MIPS, PowerPC, UltraSPARC, ARM, MIPS64, Thumb
  - CISC: IBM 360/370, VAX, and Intel 80x86

Variants of GRP Architecture

- Number of operands in ALU instructions: two or three
  - Add R1, R2, R3
  - Add R1, R2
- Maximal number of memory operands in ALU instructions: zero, one, two, or three
  - Load R1, A
  - Load R1, A
  - Load R2, B
  - Add R3, R1, B
  - Add R3, R1, R2
- Three popular combinations
  - register-register (bid-store): 0 memory, 3 operands
  - register-memory: 1 memory, 2 operands
  - memory-memory: 2 memories, 2 operands; or 3 memories, 3 operands
Register-memory

- There is no implicit operand
- One input operand is register, and one in memory
  Ex: C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C
- Processors include VAX, 80x86

Register-register (Load-store)

- Both operands are registers
- Values in memory must be loaded into a register and stored back
  Ex: C = A + B
  Load R1, A
  Load R2, B
  Add R3, R1, R2
  Store R3, C
- Processors: MIPS, SPARC

How Many Registers?

If the number of registers increases:
- Allocate more variables in registers (fast accesses)
- Reducing code spill
- Reducing memory traffic
- Longer register specifiers (difficult encoding)
- Increasing register access time (physical registers)
- More registers to save in context switch

MIPS64: 32 general-purpose registers

ISA and Performance

CPU time = #inst * CPI * cycle time

- RISC with Register-Register instructions
  - Simple, fixed-length instruction encoding
  - Simple code generation
  - Regularity in CPI
  - Higher instruction counts
  - Lower instruction density
- CISC with Register-memory instructions
  - More extra load in accessing data in memory
  - Easy encoding
  - Operands being not equivalent
  - Restricted registers due to encoding memory address
  - Irregularity in CPI

Memory Addressing

Instructions use registers, constant values, and memory

- Addressing mode decides how to specify an object to access
  - Object can be memory location, register, or a constant
  - Memory addressing is complicated
- Memory addressing involves many factors
  - Memory addressing mode
  - Object size
  - Byte ordering
  - Alignment

For a memory location, its effective address is calculated in a certain form, depending on the addressing mode.

Little or Big: Where to Start?

- Byte ordering:
  Where is the first byte?

- Big-endian: IBM, SPARC, Motorola
- Little-endian: Intel, DEC
- Supporting both: MIPS, PowerPC

<table>
<thead>
<tr>
<th>Number 0x5678</th>
<th>Little-endian</th>
<th>Big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>00000002</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>00000003</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>00000004</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Alignment
Align n-byte objects on n-byte boundaries (n = 1, 2, 4, 8)

- One align position, n-1 misaligned positions
- Misaligned access is undesirable
  - Expensive logic, slow references
- Aligning in registers may be necessary for bytes and half words

MIPS Data Addressing Modes
- Register
  ADD $16, $7, $8
- Immediate
  ADDI $17, $7, 100
- Displacement
  LW $18, 100($9)

Only the three are supported for data addressing

Storage Used by Compilers
Register storage
  - Holding temporaries in expression evaluation
  - Passing parameters
  - Holding variables

Memory storages consists of
  - Stack: to hold local variables
  - Global data area: to hold statically declared objects
  - Heap: to hold dynamic objects

Memory Addressing Seen in CISC
- Direct (absolute) ADD R1, (1001)
- Register indirect
  SUB R2, (R1)
- Indexed
  ADD R1, (R2 + R3)
- Scaled
  SUB R2, 100(R2)[R3]
- Autoincrement
  ADD R1, (R2)+
- Autodecrement
  SUB R2, -(R1)
- Memory indirect
  ADD R1, @(R3)

(see textbook p98, And more ...)

Choosing of Memory Addressing Modes
Choosing complex addressing modes
  - Close to addressing in high-level language
  - May reduce instruction counts (thus fast)
  - Increase implementation complexity (may increase cycle time)
  - Increase CPI

RISC ISA comes with simple memory addressing, and CISC ISA with complex ones

How Often Are Those Address Modes?

<table>
<thead>
<tr>
<th>Address Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>10%</td>
</tr>
<tr>
<td>Indirect</td>
<td>20%</td>
</tr>
<tr>
<td>Indexed</td>
<td>30%</td>
</tr>
<tr>
<td>Scaled</td>
<td>40%</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>5%</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>10%</td>
</tr>
</tbody>
</table>

Usage of address modes, VAX machine, SPEC99
Usage of Immediate Operands In RISC

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Integer average</th>
<th>Floatpoint average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>12%</td>
<td>22%</td>
</tr>
<tr>
<td>ALU operations</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>All instructions</td>
<td>10%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Alpha, SPEC CINT2000 & CFP2000

Immediate Size in RISC

Operands size, type and format
- In MIPS Opcode encodes operand size
  - E, ADD for signed integer, ADDU for unsigned integer, ADD D for double precision FP
- Most common types include
  - Integer: complement binary numbers
    - Character: ASCII
    - Floating point: IEEE standard 754, single-precision or double-precision
- Decimal format
  - 4 bits for one decimal digit (0-9), one byte for two decimal digits
  - Necessary for business applications
- Fixed Point format in DSP processors:
  - Representing fractions in [-1, 1]
  - 10000101_2 = 0.1000101_2

Displacement Size in RISC

Dynamic Instruction Mix (MIPS)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SPEC2K Int</th>
<th>SPEC2K FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>26%</td>
<td>15%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2%</td>
</tr>
<tr>
<td>Add</td>
<td>19%</td>
<td>23%</td>
</tr>
<tr>
<td>Compare</td>
<td>5%</td>
<td>2%</td>
</tr>
<tr>
<td>Cond br</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond mv</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>Jump</td>
<td>1%</td>
<td>0%</td>
</tr>
<tr>
<td>LOGIC</td>
<td>16%</td>
<td>4%</td>
</tr>
<tr>
<td>FP load</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>FP store</td>
<td>7%</td>
<td></td>
</tr>
<tr>
<td>FP others</td>
<td>19%</td>
<td></td>
</tr>
</tbody>
</table>

Compiler Effects

Architectures change for the needs of compilers
- How do compilers use registers? How many?
- How do compilers use addressing modes?
- Anything that compilers do not like?