G22.2243-001
High Performance Computer Architecture

Lecture 2
Instruction Set Architecture

Outline

• Administrative stuff
  – Accounts on department Suns?

• Fundamentals of computer design
  – Measuring and reporting performance (cont’d)
  – Quantitative principles of computer design

• Instruction set principles
  – What is an instruction set?
  – What is a good instruction set?
  – Instruction set aspects
  – RISC vs. CISC

• Instruction set examples: See Appendices C, D, E, and F (online)

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 2]
(Review) Measurement and Evaluation

- Architecture is an iterative process:
  - Search the possible design space
  - Make selections
  - Evaluate the selections made

- Good measurement tools are required to accurately evaluate the selection

- The book argues for what has now become widely accepted: A **quantitative approach** for evaluating selections
  - most accurate measure of performance is the **execution time** of representative **real programs** (benchmarks)

- Last lecture: CPU time equation

\[
\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

Programs to Evaluate Processor Performance

- **(Toy) Benchmarks**
  - 10-100 line program
  - e.g.: sieve, puzzle, quicksort

- **Synthetic Benchmarks**
  - Attempt to match average frequencies of real workloads
  - e.g., Whetstone, dhrystone

- **Kernels**
  - Time critical excerpts

- **Real Benchmarks**
  - Ideal: exactly the programs one would like to run on the architecture
  - More generally: a collection of programs that possess the characteristics of the real workload
SPEC: System Performance Evaluation Cooperative

- First Round SPEC CPU89
  - 10 programs yielding a single number
- Second Round SPEC CPU92
  - SPEC CINT92 (6 integer programs) and SPEC CFP92 (14 floating point programs)
  - Compiler flags can be set differently for different programs
- Third Round SPEC CPU95
  - new set of programs: SPEC CINT95 (8 integer programs) and SPEC CFP95 (10 floating point)
  - Single flag setting for all programs
- Fourth Round SPEC CPU2000
  - new set of programs: SPEC CINT2000 (12 integer programs) and SPEC CFP2000 (14 floating point)
  - Single flag setting for all programs
  - Programs in C, C++, Fortran 77, and Fortran 90
  - Report both baseline and best performance

SPEC 2000

- 12 integer programs:
  - 2 Compression
  - 2 Circuit Placement and Routing
  - C Programming Language Compiler
  - Combinatorial Optimization
  - Chess, Word Processing
  - Computer Visualization
  - PERL Programming Language
  - Group Theory Interpreter
  - Object-oriented Database.
  - Written in C (11) and C++ (1)

- 14 floating point programs:
  - Quantum Physics
  - Shallow Water Modeling
  - Multi-grid Solver
  - 3D Potential Field
  - Parabolic / Elliptic PDEs
  - 3-D Graphics Library
  - Computational Fluid Dynamics
  - Image Recognition
  - Seismic Wave Simulation
  - Image Processing
  - Computational Chemistry
  - Number Theory / Primality Testing
  - Finite-element Crash Simulation
  - High Energy Nuclear Physics
  - Pollutant Distribution
  - Written in Fortran (10) and C (4)
Other SPEC Benchmarks

- JVM98:
  - Measures performance of Java Virtual Machines
- SFS97:
  - Measures performance of network file server (NFS) protocols
- Web99:
  - Measures performance of World Wide Web applications
- HPC96:
  - Measures performance of large, industrial applications
- APC, MEDIA, OPC
  - Measures performance of graphics applications
- For more information about the SPEC benchmarks see: http://www.spec.org.

Implication of the Quantitative Approach

- Make the common case fast
  - One of the principles behind RISC: Reduced Instruction Set Computers
    - Identify most frequently-used instructions
      - Implement them in hardware
    - Emulate other instructions (slowly) in software
      - Pretty much every technique used in current-day microprocessors
  - Is there a way of quantifying the gains one is likely to see by improving some portion of the design?
    - What is the best one can hope to do?

- General principle for the above: Amdahl’s Law
  (sometimes also called Amdahl’s curse)
Amdahl's Law

- Speedup due to enhancement E:

\[
\text{Speedup (E)} = \frac{\text{Execution time without E}}{\text{Execution time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}
\]

- Suppose that enhancement E accelerates a fraction \( f \) of the task by a factor \( s \), and the remainder of the task is unaffected.

- New execution time and the overall speedup?

\[
\text{Exec. time}_{\text{new}} = \text{Exec. time}_{\text{old}} \ast \left[ (1 - f) + \frac{f}{s} \right]
\]

\[
\text{Speedup (E)} = \frac{\text{Exec. Time}_{\text{old}}}{\text{Exec. Time}_{\text{new}}} = \frac{1}{1 - f + \frac{f}{s}} \leq \frac{1}{1 - f}
\]

Example of Amdahl’s Law

- Floating point instructions improved to run *2x*; but only 10% of the time was spent on these instructions.

- How much improvement in performance should one expect?

\[
\text{Exec. time}_{\text{new}} = \text{Exec. time}_{\text{old}} \ast \left[ (1 - 0.1) + \frac{0.1}{2} \right] = \text{Exec. time}_{\text{old}} \ast 0.95
\]

\[
\text{Speedup (E)} = \frac{\text{Exec. Time}_{\text{old}}}{\text{Exec. Time}_{\text{new}}} = \frac{1}{0.95} = 1.053
\]

- The new machine is 5.3% faster for this mix of instructions.
Summary: Fundamentals of Computer Design

- A fundamental rule in computer design is to make the common case fast.
- The most accurate measure of performance is the execution time of representative real programs (benchmarks).
- Execution time is dependent on the number of instructions per program, the number of cycles per instruction, and the clock rate.
- When designing computer systems, both cost and performance need to be taken into account.

- See Section 1.7 of the textbook for several examples of performance and price-performance in desktop, server, and embedded computers.

Instruction Set Principles
Instruction Set Architecture (ISA)

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

Source: IBM in 1964 when introducing the IBM 360 architecture

- An instruction set is a functional description of the processor
  - What operations can it do
  - What storage mechanisms does it support

- ISA defines the hardware/software interface
  - Implementability: supports a (performance/cost) range
  - Programmability: easy to express programs
    - Simple (understandable) model of cost per instruction
  - Backward/forward/upward compatibility
    - To permit execution of legacy (and future) applications

Instruction Set Design Issues

- What operations are supported?
  - add, sub, mul, move, compare . . .

- Where are operands stored?
  - registers, memory, stack, accumulator

- How many explicit operands are there?
  - 0, 1, 2, or 3

- How is the operand location specified?
  - register, immediate, indirect, . . .

- What type and size of operands are supported?
  - byte, int, float, double, string, vector, . . .
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit general-purpose registers
  - R0 contains zero
  - Double precision operations take pair
- 3-address (src1, src2, dst), register-register arithmetic instructions
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

- Examples:
  SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>Opx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate (e.g., load/store)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>
A “Minimal” ISA

- Only one instruction, referred to by an address
  - CPU contains an accumulator, \( A \)
  - PC embedded in memory at location 0
  - PC incremented after each instruction

- Execution of instruction \( x \)
  - \( \text{Mem}[x] = A = \text{Mem}[x] - A \)

- Sometimes referred to as the “one address, no operand” machine
  - Attributed to Willem Van der Poel

Sample operations

- Clearing a memory location \( x \)
  - \( \text{Mem}[x] = A = \text{Mem}[x] - A \)
  - \( \text{Mem}[x] = A = 0 \)

- Unconditional branch to (PC + 20)
  - \( y \)
  - \( A = \text{Mem}[y] - A \)
  - \( y \)
  - \( A = 0 \)
  - \( T_{20} \)
  - \( A = 0 - 20 \)
  - \( 0 \)
  - \( PC = A = PC - (-20) \)

- Can also do move, add, subtract, conditional branch, …

ISA Metrics

- 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

- Ease of compilation (or assembly language programming)

- Ease of implementation
Evolution of Instruction Sets

- **Single Accumulator** (EDSAC 1950)
- **Accumulator + Index Registers** (Manchester Mark I, IBM 700 series 1953)
- **Separation of Programming Model from Implementation**
- **High-level Language Based Concept of a Family** (B5000 1963)(IBM 360 1964)
- **General Purpose Register Machines**
- **Complex Instruction Sets** (Vax, Intel 432 1977-80)
- **Load/Store Architecture** (CDC 6600, Cray 1 1963-76)
- **RISC** (MIPS, Sparc, HP-PA, IBM RS6000, . . . 1987)

ISA Aspect (1): Operand Location

- **Accumulator** (before 1960):
  - 1 address: add A  \( \text{acc } \leftarrow \text{acc + mem}[A] \)
- **Stack** (1960s to 1970s):
  - 0 address: push  \( \text{tos} \leftarrow \text{tos + next} \)
- **Memory-Memory** (1970s to 1980s):
  - 2 address: add A, B  \( \text{mem}[A] \leftarrow \text{mem}[A] + \text{mem}[B] \)
  - 3 address: add A, B, C  \( \text{mem}[A] \leftarrow \text{mem}[B] + \text{mem}[C] \)
- **Register-Memory** (1970s to present):
  - 2 address: add R1, A  \( \text{R1} \leftarrow \text{R1 + mem}[A] \)
  - load R1, A  \( \text{R1} \leftarrow \text{mem}[A] \)
- **Register-Register, also called Load/Store** (1960s to present):
  - 3 address: add R1, R2, R3  \( \text{R1} \leftarrow \text{R2 + R3} \)
  - load R1, R2  \( \text{R1} \leftarrow \text{mem}[R2] \)
  - store R1, R2  \( \text{mem}[R1] \leftarrow \text{R2} \)
Choices for Operand Location

- Running example: \( C := A + B \)

- **Accumulator**
  
  ```
  load A  accum = M[A];
  add B   accum += M[B];
  store C M[C] = accum;
  ```

  + Less hardware, code density
  - Memory bottleneck

- **Stack**
  
  ```
  push A  S[++tos] = M[A];
  push B  S[++tos] = M[B];
  add      t1= S[tos--]; t2= S[tos--]; S[++tos]= t1 + t2;
  pop C   M[C] = S[tos--];
  ```

  + Less hardware, code density
  - Memory, pipelining bottlenecks
  - x86 uses stack model for floating point computations

---

Choices for Operand Location (cont’d)

- **Memory-Memory**
  
  ```
  ```

  + Code density (most compact)
  - Large variations in instruction lengths (number of addresses)
  - Large variations in work per-instruction
  - Memory bottleneck
  - No current machines support memory-memory (VAX did)

- **Memory-Register**
  
  ```
  load R1, A  R1 = M[A];
  add R1, B   R1 += M[B];
  store C, R1 M[C] = R1;
  ```

  + Like several explicit (extended) accumulators
  + Code density, easy to decode
  - Asymmetric operands, different amount of work per instruction
  - Examples: IBM 360/370, x86, Motorola 68K
Choices for Operand Location (cont’d)

- **Register-Register (Load-Store)**
  
  ```
  load R1, A      R1 = M[A];
  load R2, B      R2 = M[B];
  add R3, R1, R2  R3 = R1 + R2;
  store C, R3     M[C] = R3;
  ```

  + Easy decoding, operand symmetry
  + Deterministic cost for ALU operations (simple cost model)
  + Scheduling opportunities, register-level CSE
  - Code density

Operand Location: Registers vs. Memory

- **Pros and cons of registers**
  + Faster, direct access
  + Simple cost model (fixed latency, no misses)
  + Short identifier
    - Must save/restore on procedure calls, context switches
    - Fixed size (larger-sized structures must live in memory)
    - Can’t take address of a register (pointed-to variables must be in memory)

- **Pros and cons of more registers**
  + Possible to keep more operands for longer in faster memory
    - Shorter operand access time, lower memory traffic
    - Longer specifiers
    - Slower access to registers
    - Larger cost for saving CPU state
    - Trend towards more registers
      - 8 (x86) -> 32 (MIPS/Alpha/PPC) -> 128 (IA-64)
      - Driven by increasing compiler involvement in scheduling
ISA Aspect (2): Addressing

- **Endian-ness**: Order of bytes in words
  - **Big**: byte address `xx xx xx 00` is in the most-significant position (big end)
    - E.g., IBM, Sun SPARC
  - **Little**: byte address `xx xx xx 00` is in the least-significant position (little end)
    - E.g., x86
  - Some processors allow mode to be selectable
    - E.g., PowerPC, MIPS (new implementations of the ISA)

- **Alignment**: Natural boundaries defined by architecture
  - Aligned address: `(address % size) equals 0`
- Different ISAs support different restrictions on alignment
  - **None**: (all alignments supported by hardware): expensive/exception handling
  - **Restricted**: misaligned access traps to software
  - **Middle ground**: misaligned data okay, but requires multiple instructions
    - E.g., MIPS: `lwl/lwr` (Load Word Left/Right: only modify part of register)

Types of Addressing Modes (VAX)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Register direct</td>
<td><code>add R4, R3</code></td>
<td><code>R4 ← R4 + R3</code></td>
</tr>
<tr>
<td>2. Immediate</td>
<td><code>add R4, #3</code></td>
<td><code>R4 ← R4 + 3</code></td>
</tr>
<tr>
<td>3. Displacement</td>
<td><code>add R4, 100(R1)</code></td>
<td><code>R4 ← R4 + M[100 + R1]</code></td>
</tr>
<tr>
<td>4. Register indirect</td>
<td><code>add R4, (R1)</code></td>
<td><code>R4 ← R4 + M[R1]</code></td>
</tr>
<tr>
<td>5. Indexed</td>
<td><code>add R4, (R1 + R2)</code></td>
<td><code>R4 ← R4 + M[R1 + R2]</code></td>
</tr>
<tr>
<td>6. Direct</td>
<td><code>add R4, (1000)</code></td>
<td><code>R4 ← R4 + M[1000]</code></td>
</tr>
<tr>
<td>7. Memory Indirect</td>
<td><code>add R4, @(R3)</code></td>
<td><code>R4 ← R4 + M[M[R3]]</code></td>
</tr>
<tr>
<td>8. Autoincrement</td>
<td><code>add R4, (R2)+</code></td>
<td><code>R4 ← R4 + M[R2]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R2 ← R2 + d</code></td>
</tr>
<tr>
<td>9. Autodecrement</td>
<td><code>add R4, (R2)−</code></td>
<td><code>R4 ← R4 + M[R2]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R2 ← R2 − d</code></td>
</tr>
<tr>
<td>10. Scaled</td>
<td><code>add R4, 100(R2)[R3]</code></td>
<td><code>R4 ← R4 + M[100 + R2 + R3*d]</code></td>
</tr>
</tbody>
</table>
Which modes are actually used?

- Study by Clark and Emer
  - Modes 1–4 account for 93% of all VAX operands
- Register mode responsible for roughly half
- Memory modes

An example of making the common case fast
- RISC machines typically implement register, immediate, and displacement
- Synthesize all other modes in software

Addressing Modes for Signal Processing

Two additional modes, motivated by DSP applications
- **Modulo** or circular addressing
  - DSPs deal with large (infinite, continuous) streams of data
  - Typically encoded as a circular buffer (mode allows auto-reset)
- **Bit reverse** addressing
  - Permit permutations on address (to support kernels such as FFT)
  - E.g., address/displ. (100) results in access to address/displ. (001)
- Renewed importance of autoincrement and autodecrement modes
  - Study on TI TMS320C54x DSP finds autoincrement mode use of ~18.8%
ISA Aspect (3): Types of Operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control: BRANCH, JUMP, CALL
- System: OS CALL, VM
- Floating Point: ADDF, MULF, DIVF
- Decimal: ADDD, CONVERT
- String: MOVE, COMPARE
- Graphics: (DE)COMPRESS

Relative Frequency of Instructions

- For the 80x86, averaged over five SPECin92 programs

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

- Simple instructions dominate
Operations for Media and Signal Processing

- Results of media processing gauged in terms of human perception
  - Narrow data items (8-16 bits) as opposed to 32 or 64-bit words
  - Lower precision requirements
  - Real-time requirements → cannot cause overflow traps

- Several ISAs have been extended to support graphics and multimedia
  - Intel: MMX, Internet Streaming SIMD Extensions (and ISSE2)
  - AMD: 3DNow!
  - Sun: Visual Instruction Set (VIS)
  - Motorola and IBM: AltiVec

- Common theme
  - Partitioned operations (SIMD), pack/unpack operations
  - Saturating arithmetic
  - Multiply-accumulate (MAC) instructions (dot products and vector multiplies)

---

**MMX**

- **Packed** data types
  - Packed byte
  - Packed word
  - Double word
  - Quad word
  - Stored in 64-bit FP registers
    - MMX does not add any new state to the processor
    - MMX and FP don’t work well together

- **SIMD** instructions work on the packed types (single cycle)
  - Minimal hardware to isolate sub-operations (5% increase in chip area)
  - E.g., $\text{PADD}[W]$

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  a_3 + b_3 & a_2 + b_2 & a_1 + b_1 & a_0 + b_0 \\
\end{array}
\]
MMX (cont’d)

- **Saturating arithmetic**
  - Both wrap-around and saturating adds are supported
  - Saturating mode: results that overflow are set to the largest value

```markdown
<table>
<thead>
<tr>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>FFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>8000</td>
</tr>
</tbody>
</table>
```

- **Multiply-accumulate (MAC) operations**
  - E.g., **PMADDWD**: Packed multiply-add word to double

```markdown
<table>
<thead>
<tr>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
</tr>
</tbody>
</table>
```

```markdown
| a3*b3 + a2*b2 | a1*b1 + a0*b0 |
```

MMX: Example

- Vector dot product on an 8-element vector
  - 9 MMX instructions (as compared to 40 without MMX)

```
x = \sum a(i) \times b(i)
```

```
PMADDWD (2)

<table>
<thead>
<tr>
<th>a7</th>
<th>a6</th>
<th>a5</th>
<th>a4</th>
</tr>
</thead>
<tbody>
<tr>
<td>b7</td>
<td>b6</td>
<td>b5</td>
<td>b4</td>
</tr>
</tbody>
</table>
```

```
| a3*b3 + a2*b2 | a1*b1 + a0*b0 |
```

```
copy and shift (4)

| a7*b7 + a6*b6 | a5*b5 + a4*b4 |
```

```
| a3*b3 + a2*b2 |
```

```
PADDD (2)

| a7*b7 +...+ a4*b4 |
```

```
PADDD (1)

| a7*b7 +...+ a0*b0 |
```
Performance Impact of MMX

- Pentium processors (200 MHz, 512 KB L2) with and without MMX
  (“MMX Technology Architecture Overview” – Mittal, Peleg, Weiser
  Intel Technology Journal, 3Q 1997)

<table>
<thead>
<tr>
<th>Application</th>
<th>Without MMX</th>
<th>With MMX</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video</td>
<td>155.52</td>
<td>268.70</td>
<td>1.72</td>
</tr>
<tr>
<td>Image Processing</td>
<td>159.03</td>
<td>743.90</td>
<td>4.67</td>
</tr>
<tr>
<td>3D geometry</td>
<td>161.52</td>
<td>166.44</td>
<td>1.03</td>
</tr>
<tr>
<td>Audio</td>
<td>149.80</td>
<td>318.90</td>
<td>2.13</td>
</tr>
<tr>
<td>Overall</td>
<td>156.00</td>
<td>255.43</td>
<td>1.64</td>
</tr>
</tbody>
</table>

- More recent Intel processors (Pentium III and 4) incorporate additional
  instructions: Internet Streaming SIMD Extensions (ISSE and ISSE-2)
  - New 128-bit registers, data prefetching
  - SIMD floating point operations (reciprocal, averaging …)
  - Performance gain of 1.5 to 2.0 on video and 3D applications

Control Instructions

- Instructions that change the value of the PC

- Three kinds of instructions
  - (conditional) branches, (unconditional) jumps
  - Function calls, function returns
  - System calls, system returns

- Questions
  - What kinds of conditions are supported? How are the conditions set?
  - How is the target address specified?
  - For function and system calls, how is the return address specified?
    - Most recent processors use an implicit register
      - Simple scheme
      - Software needs to save/restore register contents
    - Which instructions save/restore CPU state?
Control Instructions (1): Conditions

Three choices

- **“compare and branch”**
  + Single instruction branches
  - Needs ALU stage for branch instructions as

- **Separate “compare” and “branch”**
  + More scheduling opportunities, possible to reuse value of comparison
  - Uses up a register, separates condition from branch logic

- **Implicit or explicit condition codes** (e.g., zero, negative, overflow)
  + Set by the ALU
  - Part of the CPU state, needs to be saved and restored
  - Since state is implicit, difficult to schedule instructions

- **Recent processors (e.g., Alpha, IA-64) offer predicated instructions**
  - combine comparison, branch, and ALU operations
  - more about these later in the course

MIPS instructions

- Branch on == 0, <> 0
  (corresponds to ~50% of branches)
  - For other conditions, separate “compare and set” and “branch” instructions

Control Instructions (2): Target Address

Four choices

- **PC-relative with immediate**
  + Position independent, all info present for computing target
  + Short immediate sufficient (compact instructions)
  - Target must be known statically
  
  Uses: branches/jumps within function

- **Arbitrary immediate**
  
  Uses: function calls

- **Register**
  + Short specifier, target can be dynamic
  - Extra instruction to load register
  
  Uses: indirect calls (e.g., DLLs, virtual functions), returns, switches

- ** Vectored traps**
  + Protection (hence, heavyweight)
  
  Uses: system calls
Control Instructions (3): Save and Restore State

- Only call (function and system) instructions need to save state
  - Function calls: save registers
  - System calls: save registers, flags, PC, PSW, …

Two choices
- **Software** saving/restoring
  - Calling convention distinguishes between caller- and callee-save registers
- **Hardware** saving/restoring
  - Explicit instructions: VAX
  - Implicit: SPARC register windows
    - 32 registers: 8 input, 8 output, 8 local, 8 global
    - On call: 8 output of caller become 8 input of callee
      - Locals/output are new set (no need to save/restore), global is unchanged
    - On return: opposite, callee needs to restore 8 output of caller before returning
      + No save/restore on shallow call graphs
      - Makes advanced architectural techniques (e.g., register renaming) hard

The RISC vs. CISC Debate

- Early 80s: Several projects challenged how processors were being built
  - Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801
  - Contrasted their design, RISC (Reduced Instruction Set Computer) with what began to be called CISC (Complex Instruction Set Computer)

- **RISC** argument
  - CISC is too complex to ever be implemented well
    - too many addressing modes, variable format instructions, many multi-cycle operations, microcoding, hand-assembled programs
  - RISC characterized by
    - Single-cycle operation
    - Hardwired control
    - Load/store organization
    - Fixed instruction format
    - Few modes
    - Reliance on compiler optimization
    Motivated by quantitative studies of program behavior
    Focus on optimizing the common case
The Reality of RISC vs. CISC

- RISC does help compiler optimizations
  - Load/store architecture supports register allocation
    - Explicit choices of what values reside where in the memory hierarchy
  - Simple instructions make instruction selection, optimization easier

- However, CISC does not have any fundamental implementation flaws
  - Fixable with more transistors
    - Good CISC pipeline can be constructed with modest increase in transistors
    - Not an issue given Moore’s law
  - Further, irrespective of CISC or RISC, software costs will dominate

- Most commercially successful ISA is x86 (CISC)
  - Current-day Pentium processors translate CISC instructions into sequences of RISC micro-ops
    - Internal microarchitecture is actually RISC
  - Better substrate for implementing advanced architectural techniques