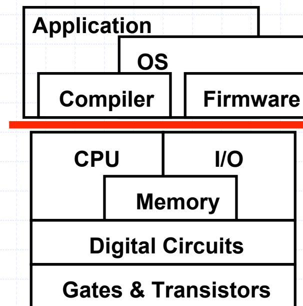
CIS 501
Introduction to Computer Architecture

Unit 2: Instruction Set Architecture

Instruction Set Architecture (ISA)



- What is a good ISA?
- Aspects of ISAs
- RISC vs. CISC
- Implementing CISC: μISA

Readings

- H+P
  - Chapter 2
  - Further reading:
    - Appendix C (RISC) and Appendix D (x86)
    - Available from web page
- Paper
  - The Evolution of RISC Technology at IBM by John Cocke
- Much of this chapter will be "on your own reading"
  - Hard to talk about ISA features without knowing what they do
  - We will revisit many of these issues in context

What Is An ISA?

- ISA (instruction set architecture)
  - A well-define hardware/software interface
  - The **"contract"** between software and hardware
    - **Functional definition** of operations, modes, and storage locations supported by hardware
    - **Precise description** of how to invoke, and access them
  - No guarantees regarding
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less

A Language Analogy for ISAs

- A ISA is analogous to a human language
  - Allows communication
    - Language: person to person
    - ISA: hardware to software
  - Need to speak the same language/ISA
  - Many common aspects
    - Part of speech: verbs, nouns, adjectives, adverbs, etc.
    - Common operations: calculation, control/branch, memory
  - Many different languages/ISAs, many similarities, many differences
    - Different structure
  - Both evolve over time
- Key differences: ISAs must be unambiguous
  - ISAs are explicitly engineered and extended

RISC vs CISC Foreshadowing

- Recall performance equation:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC (Complex Instruction Set Computing)
  - Improve "instructions/program" with "complex" instructions
  - Easy for assembly-level programmers, good code density
- RISC (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - Help from smart compiler
  - Perhaps improve clock cycle time (seconds/cycle)
    - via aggressive implementation allowed by simpler instructions

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4,...

Programmability

- Easy to express programs efficiently?
  - For whom?
- Before 1985: **human**
  - Compilers were terrible, most code was hand-assembled
  - Want high-level coarse-grain instructions
    - As similar to high-level language as possible
- After 1985: **compiler**
  - Optimizing compilers generate much better code that you or I
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly or not

Human Programmability

- What makes an ISA easy for a human to program in?
  - Proximity to a high-level language (HLL)
    - Closing the "**semantic gap**"
  - Semantically heavy (CISC-like) insns that capture complete idioms
    - "Access array element", "loop", "procedure call"
    - Example: SPARC `save/restore`
    - Bad example: x86 `rep movsb` (copy string)
    - Ridiculous example: VAX `insque` (insert-into-queue)
  - "**Semantic clash**": what if you have many high-level languages?
- Stranger than fiction
  - People once thought computers would execute language directly
  - Fortunately, never materialized (but keeps coming back around)

Compiler Programmability

- What makes an ISA easy for a compiler to program in?
  - Low level primitives from which solutions can be synthesized
    - Wulf: "**primitives not solutions**"
    - Computers good at breaking complex structures to simple ones
      - Requires traversal
    - Not so good at combining simple structures into complex ones
      - Requires search, pattern matching (why AI is hard)
    - Easier to synthesize complex insns than to compare them
  - Rules of thumb
    - Regularity: "**principle of least astonishment**"
    - Orthogonality & composability
    - One-vs.-all

Today's Semantic Gap

- Popular argument
  - Today's ISAs are targeted to one language…
  - Just so happens that this language is very low level
    - The C programming language
- Will ISAs be different when Java/C# become dominant?
  - Object-oriented? Probably not
  - Support for garbage collection? Maybe
  - Support for bounds-checking? Maybe
  - Why?
    - Smart compilers transform high-level languages to simple instructions
    - Any benefit of tailored ISA is likely small

Implementability

- Every ISA can be implemented
  - Not every ISA can be implemented efficiently
- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
  - – Variable instruction lengths/formats: complicate decoding
  - – Implicit state: complicates dynamic scheduling
  - – Variable latencies: complicates scheduling
  - – Difficult to interrupt instructions: complicate many things

Compatibility

- No-one buys new hardware… if it requires new software
  - Intel was the first company to realize this
  - ISA must remain compatible, no matter what
    - x86 one of the worst designed ISAs EVER, but survives
    - As does IBM's 360/370 (the *first* "ISA family")
- **Backward compatibility**
  - New processors must support old programs (can't drop features)
  - Very important
- **Forward (upward) compatibility**
  - Old processors must support new programs (with software help)
    - New processors redefine only previously-illegal opcodes
    - Allow software to detect support for specific new instructions
    - Old processors emulate new instructions in low-level software

The Compatibility Trap

- Easy compatibility requires forethought
  - Temptation: use some ISA extension for 5% performance gain
  - Frequent outcome: gain diminishes, disappears, or turns to loss
    - Must continue to support gadget for eternity
  - Example: register windows (SPARC)
    - Adds difficulty to out-of-order implementations of SPARC
    - Details shortly

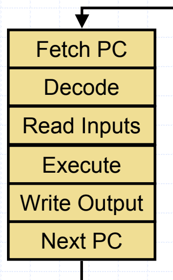
The Compatibility Trap Door

- Compatibility's friends
  - **Trap**: instruction makes low-level "function call" to OS handler
  - **Nop**: "no operation" - instructions with no functional semantics
- Backward compatibility
  - Handle rarely used but hard to implement "legacy" opcodes
  - Define to trap in new implementation and emulate in software
    - Rid yourself of some ISA mistakes of the past
    - Problem: performance suffers
- Forward compatibility
  - Reserve sets of trap & nop opcodes (don't define uses)
  - Add ISA functionality by overloading traps
    - Release firmware patch to "add" to old implementation
  - Add ISA hints by overloading nops

Aspects of ISAs

- **VonNeumann model**
  - Implicit structure of all modern ISAs
- Format
  - Length and encoding
- **Operand model**
  - Where (other than memory) are operands stored?
- Datatypes and operations
- Control
- Overview only
  - Read about the rest in the book and appendices

The Sequential Model



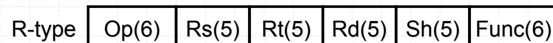
- Implicit model of all modern ISAs
  - Often called VonNeuman, but in ENIAC before
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instruction
    - Next PC is PC++ unless insn says otherwise
  - Order and **named storage** define computation
    - Value flows from insn X to Y via storage A iff…
    - X names A as output, Y names A as input…
    - And Y after X in total order
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts
- Alternatives have been proposed…

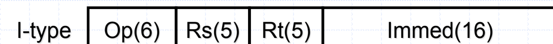
Format

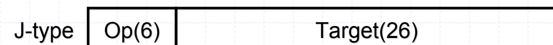
- **Length**
  - Fixed length
    - Most common is 32 bits
    - + Simple implementation: compute next PC using only PC
    - – Code density: 32 bits to increment a register by 1?
      - – x86 can do this in one 8-bit instruction
  - Variable length
    - – Complex implementation
    - + Code density
  - Compromise: two lengths
    - MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify decoder implementation

Example: MIPS Format

- Length
  - 32-bits
- Encoding
  - 3 formats, simple encoding
  - Q: how many instructions can be encoded? A: 127

R-type 

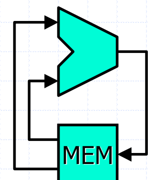
I-type 

J-type 

Operand Model: Memory Only

- Where (other than memory) can operands come from?
  - And how are they specified?
  - Example: A = B + C
  - Several options
- **Memory only**

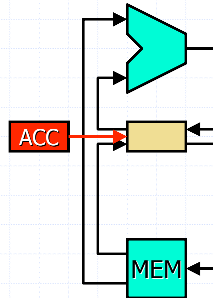
`add B,C,A` mem[A] = mem[B] + mem[C]



Operand Model: Accumulator

- **Accumulator**: implicit single element storage

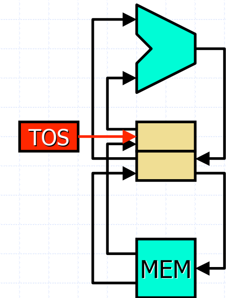
`load B`	ACC = mem[B]
`add C`	ACC = ACC + mem[C]
`store A`	mem[A] = ACC



Operand Model: Stack

- **Stack**: TOS implicit in instructions

`push B`	stk[TOS++] = mem[B]
`push C`	stk[TOS++] = mem[C]
`add`	stk[TOS++] = stk[--TOS] + stk[--TOS]
`pop A`	mem[A] = stk[--TOS]



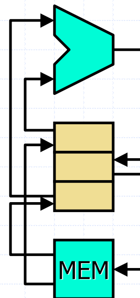
Operand Model: Registers

- **General-purpose register**: multiple explicit accumulator

`load B,R1`	R1 = mem[B]
`add C,R1`	R1 = R1 + mem[C]
`store R1,A`	mem[A] = R1

- **Load-store**: GPR and only loads/stores access memory

`load B,R1`	R1 = mem[B]
`load C,R2`	R2 = mem[C]
`add R1,R2,R1`	R1 = R1 + R2
`store R1,A`	mem[A] = R1



Operand Model Pros and Cons

- Metric I: **static code size**
  - Number of instructions needed to represent program, size of each
  - Want many implicit operands, high level instructions
  - Good → bad: memory, accumulator, stack, load-store
- Metric II: **data memory traffic**
  - Number of bytes move to and from memory
  - Want as many long-lived operands in on-chip storage
  - Good → bad: load-store, stack, accumulator, memory
- Metric III: **cycles per instruction**
  - Want short (1 cycle?), little variability, few nearby dependences
  - Good → bad: load-store, stack, accumulator, memory
- Upshot: most new ISAs are load-store or hybrids

How Many Registers?

- Registers faster than memory, have as many as possible?
  - **No**
  - One reason registers are faster is that there are **fewer of them**
    - Small is fast (hardware truism)
  - Another is that they are **directly addressed** (no address calc)
    - More of them, means larger specifiers
    - Fewer registers per instruction or indirect addressing
  - **Not everything can be put in registers**
    - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
  - More registers means **more saving/restoring**
  - Upshot: trend to more registers: 8 (x86)→32 (MIPS) →128 (IA32)
    - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

Register Windows

- **Register windows**: hardware activation records
  - Sun SPARC (from the RISC I)
  - 32 integer registers divided into: 8 global, 8 local, 8 input, 8 output
  - Explicit `save`/`restore` instructions
    - Global registers fixed
    - `save`: inputs "pushed", outputs → inputs, locals zeroed
    - `restore`: locals zeroed, inputs → outputs, inputs "popped"
    - Hardware stack provides few (4) on-chip register frames
    - Spilled-to/filled-from memory on over/under flow
  - \+ Automatic parameter passing, caller-saved registers
  - \+ No memory traffic on shallow (<4 deep) call graphs
  - – Hidden memory operations (some restores fast, others slow)
  - – A nightmare for register renaming (more later)

Virtual Address Size

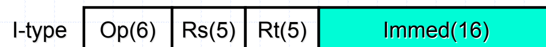
- What is a n-bit processor?
  - Support memory size of 2^n
  - Alternative (wrong) definition: size of calculation operations
- Virtual address size
  - Determines size of addressable (usable) memory
    - Current 32-bit or 64-bit address spaces
    - All ISAs moving to (if not already at) 64 bits
  - Most critical, inescapable ISA design decision
    - Too small? Will limit the lifetime of ISA
    - May require nasty hacks to overcome (E.g., x86 segments)
  - x86 evolution:
    - 4-bit (4004), 8-bit (8008), 16-bit (8086), 20-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD's Opteron & Intel's EM64T Pentium4)

Memory Addressing

- **Addressing mode:** way of specifying address
  - Used in memory-memory or load/store instructions in register ISA
- Examples
  - **Register-Indirect:** R1=mem[R2]
  - **Displacement:** R1=mem[R2+immed]
  - **Index-base:** R1=mem[R2+R3]
  - **Memory-indirect:** R1=mem[mem[R2]]
  - **Auto-increment:** R1=mem[R2], R2= R2+1
  - **Auto-indexing:** R1=mem[R2+immed], R2=R2+immed
  - **Scaled:** R1=mem[R2+R3*immed1+immed2]
  - **PC-relative:** R1=mem[PC+imm]
- What high-level program idioms are these used for?

Example: MIPS Addressing Modes

- MIPS implements only displacement
  - Why? Experiment on VAX (ISA with every mode) found distribution
  - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use small displacement or register indirect (displacement 0)
- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes? VAX experiment showed 1% accesses use displacement >16



- SPARC adds Reg+Reg mode

Two More Addressing Issues

- **Access alignment**: address % size == 0?
  - Aligned: `load-word @XXXX00`, `load-half @XXXXX0`
  - Unaligned: `load-word @XXXX10`, `load-half @XXXXX1`
  - Question: what to do with unaligned accesses (uncommon case)?
    - Support in hardware? Makes all accesses slow
    - Trap to software routine? Possibility
    - Use regular instructions
      - Load, shift, load, shift, and
    - **MIPS? ISA support**: unaligned access using two instructions
      `lwl @XXXX10; lwr @XXXX10`
- **Endian-ness**: arrangement of bytes in a word
  - Big-endian: sensible order (e.g., MIPS, PowerPC)
    - A 4-byte integer: "00000000 00000000 00000010 00000011" is 515
  - Little-endian: reverse order (e.g., x86)
    - A 4-byte integer: "00000011 00000010 00000000 00000000 " is 515
  - Why little endian? To be different? To be annoying? Nobody knows

Control Instructions

- One issue: **testing for conditions**
  - Option I: **compare and branch insns**
    `branch-less-than R1,10,target`
    + Simple, – two ALUs: one for condition, one for target address
  - Option II: **implicit condition codes**
    `subtract R2,R1,10   // sets "negative" CC`
    `branch-neg target`
    + Condition codes set "for free", – implicit dependence is tricky
  - Option III: **condition registers, separate branch insns**
    `set-less-than R2,R1,10`
    `branch-not-equal-zero R2,target`
    – Additional instructions, + one ALU per, + explicit dependence

Example: MIPS Conditional Branches

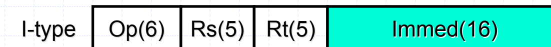
- MIPS uses combination of options II/III
  - Compare 2 registers and branch: `beq`, `bne`
    - Equality and inequality only
    + Don't need an adder for comparison
  - Compare 1 register to zero and branch: `bgtz`, `bgez`, `bltz`, `blez`
    - Greater/less than comparisons
    + Don't need adder for comparison
  - Set explicit condition registers: `slt`, `sltu`, `slti`, `sltiu`, etc.
- Why?
  - More than 80% of branches are (in)equalities or comparisons to 0
  - OK to take two insns to do remaining branches (MCCF)

Control Instructions II

- Another issue: **computing targets**
  - Option I: **PC-relative**
    - Position-independent within procedure
    - Used for branches and jumps within a procedure
  - Option II: **Absolute**
    - Position independent outside procedure
    - Used for procedure calls
  - Option III: **Indirect** (target found in register)
    - Needed for jumping to dynamic targets
    - Used for returns, dynamic procedure calls, switches
  - How far do you need to jump?
    - Typically not so far within a procedure (they don't get that big)
    - Further from one procedure to another

MIPS Control Instructions

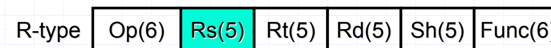
- MIPS uses all three
  - PC-relative conditional branches: `bne`, `beq`, `blez`, etc.
    - 16-bit relative offset, <0.1% branches need more

I-type 

  - Absolute jumps unconditional jumps: `j`
    - 26-bit offset

J-type

  - Indirect jumps: `jr`

R-type 

Control Instructions III

- Another issue: support for procedure calls?
  - Link (remember) address of calling insn + 4 so we can return to it
- MIPS
  - Implicit return address register is $31
  - Direct jump-and-link: `jal`
  - Indirect jump-and-link: `jalr`

RISC and CISC

- **RISC**: reduced-instruction set computer
  - Coined by Patterson in early 80's
  - Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- **CISC**: complex-instruction set computer
  - Term didn't exist before "RISC"
  - x86, VAX, Motorola 68000, etc.
- Religious war (one of several) started in mid 1980's
  - RISC "won" the technology battles
  - CISC won the commercial war
    - Compatibility a stronger force than anyone (but Intel) thought
    - Intel beat RISC at its own game

The Setup

- Pre 1980
  - Bad compilers
  - Complex, high-level ISAs
  - Slow multi-chip micro-programmed implementations
    - Vicious feedback loop
- Around 1982
  - Advances in VLSI made single-chip microprocessor possible…
    - Speed by integration, on-chip wires much faster than off-chip
  - …but only for very small, very simple ISAs
  - Compilers had to get involved in a big way
- **RISC manifesto**: create ISAs that…
  - Simplify single-chip implementation
  - Facilitate optimizing compilation

The RISC Tenets

- **Single-cycle execution**
  - CISC: many multicycle operations
- **Hardwired control**
  - CISC: microcoded multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance

The CISCs

- DEC VAX (**V**irtual **A**ddress e**X**tension to PDP-11): 1977
  - Variable length instructions: 1-321 bytes!!!
  - 14 GPRs + PC + stack-pointer + condition codes
  - Data sizes: 8, 16, 32, 64, 128 bit, decimal, string
  - Memory-memory instructions for all data sizes
  - Special insns: **`crc`**, **`insque`**, **`polyf`**, and a cast of hundreds
- Intel x86 (IA32): 1974
  - "Difficult to explain and impossible to love"
  - Variable length instructions: 1-16 bytes
  - 8 special purpose registers + condition codes
  - Data sizes: 8,16,32,**64** (new) bit (overlapping registers)
  - Accumulators (register and memory) for integer, stack for FP
  - Many modes: indirect, scaled, displacement + **segments**
  - Special insns: **`push`**, **`pop`**, string functions, MMX, SSE/2/3 (later)

The RISCs

- Many similar ISAs: MIPS, PA-RISC, SPARC, PowerPC, Alpha
  - 32-bit instructions
  - 32 registers
  - 64-bit virtual address space
  - Fews addressing modes (SPARC and PowerPC have more)
  - Why so many? Everyone invented their own new ISA
- DEC Alpha (**E**xtended **V**AX): 1990
  - The most recent, cleanest RISC ISA
  - 64-bit data (32,16,8 added only after software vendor riots)
  - Only aligned memory access
  - One addressing mode: displacement
  - Special instructions: conditional moves, prefetch hints

Post-RISC

- Intel/HP IA64 (Itanium): 2000
  - Fixed length instructions: 128-bit 3-operation bundles
    - EPIC: explicitly parallel instruction computing
  - 128 64-bit registers
  - Special instructions: true predication, software speculation
  - Every new ISA feature suggested in last two decades
    - More later in course

The RISC Debate

- RISC argument [Patterson et al.]
  - CISC is fundamentally handicapped
  - For a given technology, RISC implementation will be better (faster)
    - Current technology enables single-chip RISC
    - When it enables single-chip CISC, RISC will be pipelined
    - When it enables pipelined CISC, RISC will have caches
    - When it enables CISC with caches, RISC will have next thing...
- CISC rebuttal [Colwell et al.]
  - CISC flaws not fundamental, can be fixed with more transistors
  - Moore's Law will narrow the RISC/CISC gap (true)
    - Good pipeline: RISC = 100K transistors, CISC = 300K
    - By 1995: 2M+ transistors had evened playing field
  - Software costs dominate, compatibility is important (so true)

Current Winner (units sold): ARM

- ARM (Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - 1.2 billion units sold in 2004
    - More than half of all 32/64-bit CPUs sold
  - Low-power and embedded devices (iPod, for example)
- 32-bit RISC ISA
  - 16 registers
  - Many addressing modes (for example, auto increment)
  - Condition codes, each instruction can be conditional
- Multiple compatible implementations
  - Intel's X-scale (original design was DEC's, bought by Intel)
  - Others: Freescale (was Motorola), IBM, Texas Instruments, Nintendo, STMicroelectronics, Samsung, Sharp, Philips, etc.
- "Thumb" 16-bit wide instructions
  - Increase code density

Current Winner (revenue): x86

- x86 was first 16-bit chip by ~2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and "financial feedback"
    - x86 is most difficult ISA to implement and do it fast but…
    - Because Intel sells the most non-embedded processors…
    - It has the most money…
    - Which it uses to hire more and better engineers…
    - Which it uses to maintain competitive performance …
    - And given equal performance compatibility wins…
    - So Intel sells the most non-embedded processors…
  - AMD as a competitor keeps pressure on x86 performance
- Moore's law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors

Intel's Trick: RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
  - OOO was very hard to do with a coarse grain ISA like x86
  - Their solution? Translate x86 to RISC **uops** in hardware

```
push $eax
```
is translated (dynamically in hardware) to
```
store $eax [$esp-4]
addi $esp,$esp,-4
```

  - Processor maintains **x86 ISA externally for compatibility**
  - But executes **RISC μISA internally for implementability**
    - Translation itself is proprietary, but 1.6 uops per x86 insn
  - Given translator, x86 almost as easy to implement as RISC
    - Result: Intel implemented OOO before any RISC company
  - Idea co-opted by other x86 companies: AMD and Transmeta
    - The one company that resisted (Cyrix) couldn't keep up

Transmeta's Take: Code Morphing

- **Code morphing**: x86 translation performed in software
  - Crusoe/Astro are x86 emulators, no actual x86 hardware anywhere
  - Only "code morphing" translation software written in native ISA
  - Native ISA is invisible to applications, OS, even BIOS
  - Different Crusoe versions have (slightly) different ISAs: can't tell
  - How was it done?
    - Code morphing software resides in boot ROM
    - On startup boot ROM hijacks 16MB of main memory
    - Translator loaded into 512KB, rest is **translation cache**
    - Software starts running in **interpreter** mode
    - Interpreter profiles to find "hot" regions: procedures, loops
    - Hot region compiled to native, optimized, cached
    - Gradually, more and more of application starts running native

Emulation/Binary Translation

- Compatibility is still important but definition has changed
  - Less necessary that processor ISA be compatible
  - As long as some combination of ISA + software translation layer is
  - Advances in emulation, binary translation have made this possible
  - **Binary-translation**: transform static image, run native
  - **Emulation**: unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
  - Examples
    - FX!32: x86 on Alpha
    - IA32EL: x86 on IA64
    - Rosetta: PowerPC on x86
  - Downside: performance overheads

Virtual ISAs

- Machine virtualization
  - Vmware & Xen: x86 on x86 (what is this good for?)
  - Old idea (from IBM mainframes), big revival in the near future
- Java and C# use an ISA-like interface
  - JavaVM uses a stack-based bytecode
  - C# has the CLR (common language runtime)
  - Higher-level than machine ISA
    - Design for translation (not direct execution)
  - Goals:
    - Portability (abstract away the actual hardware)
    - Target for high-level compiler (one per language)
    - Source for low-level translator (one per ISA)
    - Flexibility over time

ISA Research

- Compatibility killed ISA research for a while
  - But binary translation/emulation has revived it
- Current projects
  - "ISA for Instruction-Level Distributed Processing" [Kim,Smith]
    - Multi-level register file exposes local/global communication
  - "DELI: Dynamic Execution Layer Interface" [HP]
    - An open translation/optimization/caching infrastructure
  - "WaveScalar" [Swanson,Shwerin,Oskin]
    - The vonNeumann alternative
  - "DISE: Dynamic Instruction Stream Editor" [Corliss,Lewis,Roth]
    - A programmable μISA: μISA/binary-rewriting hybrid
    - Local project: http://…/~eclewis/proj/dise/

Summary

- What makes a good ISA
  - {Programm|Implement|Compat}-ability
  - Compatibility is a powerful force
  - Compatibility and implementability: μISAs, binary translation
- Aspects of ISAs
- CISC and RISC
- Next up: caches