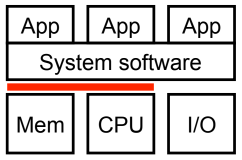
CIS 371
Computer Organization and Design

Unit 1: Instruction Set Architectures

Instruction Set Architecture (ISA)



- What is an ISA?
  - And what is a good ISA?
- Aspects of ISAs
- RISC vs. CISC
- Compatibility is a powerful force
  - Tricks: binary translation, μISAs

Readings

- Introduction
  - P+H, Chapter 1
- ISAs
  - P+H, Chapter 2

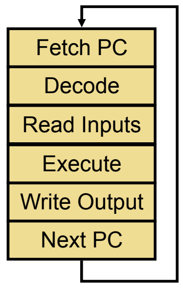
What Is An ISA?

- **ISA (instruction set architecture)**
  - A well-defined 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
  - Not in the "contract"
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less
- Instruction → Insn
  - 'Instruction' is too long to write in slides

A Language Analogy for ISAs

- Communication
  - Person-to-person → software-to-hardware
- Similar structure
  - Narrative → program
  - Sentence → insn
  - Verb → operation (add, multiply, load, branch)
  - Noun → data item (immediate, register value, memory value)
  - Adjective → addressing mode
- Many different languages, many different ISAs
  - Similar basic structure, details differ (sometimes greatly)
- Key differences between languages and ISAs
  - Languages evolve organically, many ambiguities, inconsistencies
  - ISAs are explicitly engineered and extended, unambiguous

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
- More parallel alternatives have been proposed

ISA Design Goals

What Is A Good ISA?

- **Lends itself to high-performance implementations**
  - Every ISA can be implemented
  - Not every ISA can be implemented well
- Background: **CPU performance equation**
  - Execution time: **seconds/program**
  - Convenient to factor into three pieces
  - (**insns/program**) * (**cycles/insn**) * (**seconds/cycle**)
    - Insns/program: dynamic insns executed
    - Seconds/cycle: clock period
    - Cycles/insn (CPI): hmmm…
- For high performance all three factors should be low

Insns/Program: Compiler Optimizations

- Compilers do two things
- Translate high-level languages to assembly functionally
  - Deterministic and fast compile time (`gcc –O0`)
  - "Canonical": not an active research area
  - CIS 341
- "Optimize" generated assembly code
  - "Optimize"? Hard to prove optimality in a complex system
    - In systems: "optimize" means improve… hopefully
  - Involved and relatively slow compile time (`gcc –O4`)
    - Some aspects: reverse-engineer programmer intention
  - Not "canonical": being actively researched
  - CIS 570

Compiler Optimizations

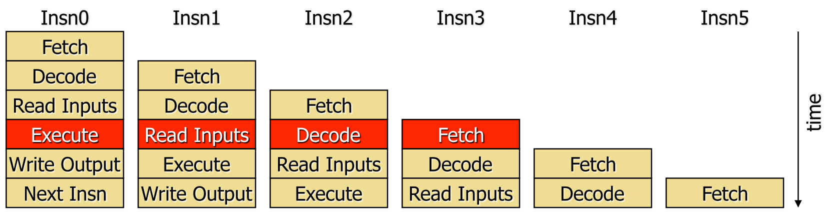
- Primarily reduce insn count
  - Eliminate redundant computation, keep more things in registers
    + Registers are faster, fewer loads/stores
    – An ISA can make this difficult by having too few registers
- But also…
  - Reduce branches and jumps (later)
  - Reduce cache misses (later)
  - Reduce dependences between nearby insns (later)
    – An ISA can make this difficult by having implicit dependences
- How effective are these?
  + Can give 4X performance over unoptimized code
  – Collective wisdom of 40 years ("Proebsting's Law"): 4% per year
  - Funny but … shouldn't leave 4X performance on the table

Seconds/Cycle and Cycle/Insn: Hmmm…

- For single-cycle datapath
  - Cycle/insn: 1 by definition
  - Seconds/cycle: proportional to "complexity of datapath"
  - ISA can make seconds/cycle high by requiring a complex datapath

Foreshadowing: Pipelining

- **Sequential model**: insn X finishes before insn X+1 starts
  - An illusion designed to keep programmers sane
- **Pipelining**: important performance technique
  - Hardware overlaps "processing iterations" for insns
  – Variable insn length/format makes pipelining difficult
  – Complex datapaths also make pipelining difficult (or clock slow)
  - More about this later

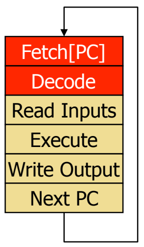


Instruction Granularity: RISC vs CISC

- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - Minimalist approach to an ISA: simple insns only
  - + Low "cycles/insn" and "seconds/cycle"
  - – Higher "insn/program", but hopefully not as much
    - Rely on compiler optimizations
- **CISC** (Complex Instruction Set Computing) **ISAs**
  - A more heavyweight approach: both simple and complex insns
  - + Low "insns/program"
  - – Higher "cycles/insn" and "seconds/cycle"
    - We have the technology to get around this problem
- More on this later, but first ISA basics

Aspects of ISAs

Length and Format



- **Length**
  - Fixed length
    - Most common is 32 bits
    - + Simple implementation (next PC often just PC+4)
    - – Code density: 32 bits to increment a register by 1
  - Variable length
    - + Code density
      - x86 can do increment in one 8-bit instruction
    - – Complex fetch (where does next instruction begin?)
  - Compromise: two lengths
    - E.g., MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify decoder
    - x86 decoder one of nastiest pieces of logic

LC3/MIPS/x86 Length and Format

- LC3: 2-byte insns, 3 formats (LC4 similar)

0-reg	Op(4)	Offset(12)			
1-reg	Op(4)	R(3)	Offset(9)		
2-reg	Op(4)	R(3)	R(3)	Offset(6)	
3-reg	Op(4)	R(3)	R(3)	U(3)	R(3)

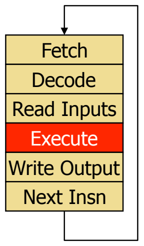
- MIPS: 4-byte insns, 3 formats

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)		
J-type	Op(6)	Target(26)				

- x86: 1–16 byte insns

Prefix*(1-4)	Op	OpExt*	ModRM*	SIB*	Disp*(1-4)	Imm*(1-4)

Operations and Datatypes

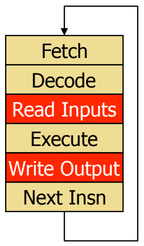


- Datatypes
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1's
- All processors support
  - 2C integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64 bit)
    - Intel has 80-bit floating-point
- More recently, most processors support
  - "Packed-integer" insns, e.g., MMX
  - "Packed-fp" insns, e.g., SSE/SSE2
  - For multimedia, more about these later
- Processor no longer (??) support
  - Decimal, other fixed-point arithmetic

LC4/MIPS/x86 Operations and Datatypes

- LC4
  - 16-bit integer: add, and, not, sub, mul, div, or, xor, shifts
  - No floating-point
- MIPS
  - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
  - 32(64) bit floating-point: add, sub, mul, div
- x86
  - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
  - 80-bit floating-point: add, sub, mul, div, sqrt
  - 64-bit packed integer (MMX): padd, pmul…
  - 64(128)-bit packed floating-point (SSE/2): padd, pmul…

Where Does Data Live?



- **Memory**
  - Fundamental storage space
- **Registers**
  - Faster than memory, quite handy
  - Most processors have these too
- Immediates
  - Values spelled out as bits in instructions
  - Input only

How Many Registers?

- Registers faster than memory, have as many as possible?
  - **No**
- One reason registers are faster: there are **fewer of them**
  - Small is fast (hardware truism)
- Another: 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**

- Trend: more registers: 8 (x86)→32 (MIPS) →128 (IA64)
  - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

LC4/MIPS/x86 Registers

- LC4
  - 8 16-bit integer registers
  - No floating-point registers
- MIPS
  - 32 32-bit integer registers ($0 hardwired to 0)
  - 32 32-bit floating-point registers (or 16 64-bit registers)
- x86
  - 8 8/16/32-bit integer registers (not general purpose)
  - No floating-point registers!
- 64-bit x86
  - 16 64-bit integer registers
  - 16 128-bit floating-point registers

How Much Memory? Address Size

- What does "64-bit" in a 64-bit ISA mean?
  - **Support memory size of 2^{64}**
  - Alternative (wrong) definition: width 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), 24-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD's Opteron & Intel's EM64T Pentium4)
- All ISAs moving to 64 bits (if not already there)

LC4/MIPS/x86 Memory Size

- LC4
  - 16-bit (2^{16} 16-bit words) x 2 (split data and instruction memory)
- MIPS
  - 32-bit
  - 64-bit
- x86
  - 8086: 16-bit
  - 80286: 24-bit
  - 80386: 32-bit
  - AMD Opteron/Athlon64, Intel's newer Pentium4, Core 2: 64-bit

How Are Memory Locations Specified?

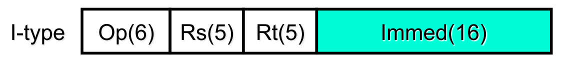
- Registers are specified **directly**
  - Register names are short, can be encoded in instructions
  - Some instructions implicitly read/write certain registers
- How are addresses specified?
  - Addresses are as big or bigger than insns
  - **Addressing mode**: how are insn bits converted to addresses?
  - Think about: what high-level idiom addressing mode captures

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?
- What implementation impact? What impact on insn count?

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
  - Why? What impact on both implementation and insn count?

LC4/MIPS/x86 Addressing Modes

- MIPS
  - **Displacement**: R1+offset (16-bit)
    - Experiments showed this covered 80% of accesses on VAX
- LC4
  - **Displacement**: R1+offset (6-bit)
- LC3 had two more modes:
  - **PC-displacement**: PC+offset (9-bit)
  - **Memory-indirect/PC-displacement**: mem[[PC]+offset(9-bit)]
- x86 (MOV instructions)
  - **Absolute**: zero + offset (8/16/32-bit)
  - **Register indirect**: R1
  - **Indexed**: R1+R2
  - **Displacement**: R1+offset (8/16/32-bit)
  - **Scaled:** R1 + (R2*Scale) + offset(8/16/32-bit) Scale = 1, 2, 4, 8

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

How Many Explicit Operands / ALU Insn?

- **Operand model**: how many explicit operands / ALU insn?
  - **3**: general-purpose
    `add R1,R2,R3` means [R1] = [R2] + [R3] **(MIPS uses this)**
  - **2**: multiple explicit accumulators (output doubles as input)
    `add R1,R2` means [R1] = [R1] + [R2] **(x86 uses this)**
  - **1**: one implicit accumulator
    `add R1` means ACC = ACC + [R1]
  - **0**: hardware stack
    `add` means STK[TOS++] = STK[--TOS] + STK[--TOS]
  - **4+**: useful only in special situations
- Examples show register operands…
  - But operands can be memory addresses, or mixed register/memory
  - ISAs with register-only ALU insns are **"load-store"**

How Do Values Get From/To Memory?

- How do values move from/to memory (primary storage)…
  - … to/from registers/accumulator/stack?
  - Assume displacement addressing for these examples
- **Registers**: load/store
  `load r1, 8(r2)` means [R1] = mem[[R2] + 8]
  `store r1, 8(r2)` means mem[[R2] + 8] = [R1]
- **Accumulator**: load/store
  `load 8(r2)` means ACC = mem[[R2] + 8]
  `store 8(r2)` means mem[[R2] + 8] = ACC
- **Stack**: push/pop
  `push 8(r2)` means STK[TOS++]= mem[[R2] + 8]
  `pop 8(r2)` means mem[[R2] + 8] = STK[TOS--]

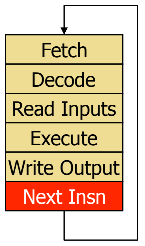
Operand Model Pros and Cons

- Metric I: **static code size**
  - Want: many Implicit operands (stack), high level insns
- Metric II: **data memory traffic**
  - Want: as many long-lived operands in on-chip storage (load-store)
- Metric III: **CPI**
  - Want: short latencies, little variability (load-store)
- CPI and data memory traffic more important these days
  - In most niches
- Trend: most new ISAs are load-store or hybrids

LC4/MIPS/x86 Operand Models

- LC4
  - Integer: 8 general-purpose registers, load-store
  - Floating-point: none
- MIPS
  - Integer/floating-point: 32 general-purpose registers, load-store
- x86
  - Integer (8 registers) reg-reg, reg-mem, mem-reg, but no mem-mem
  - Floating point: stack (why x86 floating-point lagged for years)
  - Note: integer `push`, `pop` for managing software stack
  - Note: also reg-mem and mem-mem string functions in hardware
- x86-64
  - Integer/floating-point: 16 registers

Control Transfers



- Default next-PC is PC + sizeof(current insn)
- Branches and jumps can change that
  - Otherwise dynamic program == static program
  - Not useful
- **Computing targets**: where to jump to
  - For all branches and jumps
  - Absolute / PC-relative / indirect
- **Testing conditions**: whether to jump at all
  - For (conditional) branches only
  - Compare-branch / condition-codes / condition registers

Control Transfers I: Computing Targets

- The issues
  - How far (statically) do you need to jump?
    - Not far within procedure, further from one procedure to another
  - Do you need to jump to a different place each time?
- **PC-relative**
  - Position-independent within procedure
  - Used for branches and jumps within a procedure
- **Absolute**
  - Position independent outside procedure
  - Used for procedure calls
- **Indirect** (target found in register)
  - Needed for jumping to dynamic targets
  - Used for **returns**, dynamic procedure calls, `switch` statements

Control Transfers II: Testing Conditions

- **Compare and branch insns**

```
branch-less-than R1,10,target
```

  + Simple
  – Two ALUs: one for condition, one for target address
  – Extra latency
- **Implicit condition codes (x86, LC4)**

```
subtract R2,R1,10    // sets "negative" CC
branch-neg target
```

  + Condition codes set "for free"
  – Implicit dependence is tricky
- **Conditions in regs, separate branch (MIPS)**

```
set-less-than R2,R1,10
branch-not-equal-zero R2,target
```

  – Additional insns
  + one ALU per insn, explicit dependence

LC4, MIPS, x86 Control Transfers

- LC4
  - 9-bit offset PC-relative branches/jumps (uses condition codes)
  - 11-bit offset PC-relative calls and indirect calls
- MIPS
  - 16-bit offset PC-relative conditional branches (uses register for condition)
  - Simple banches
    - Compare two registers: **`beq`**, **`bne`**
    - Compare reg to zero: **`bgtz`**, **`bgez`**, **`bltz`**, **`blez`**
    + Don't need adder for these, cover 80% of cases
  - Explicit "set condition into registers": **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, etc.
  - 26-bit target absolute jumps and function calls
- x86
  - 8-bit offset PC-relative branches (uses condition codes)
  - Explicit compare instructions to set condition codes
  - 8/16-bit target absolute jumps and function calls (within segment)
    - Far jumps and calls (change code segment) for longer jumps

Later: ISA Include Support For…

- Operating systems & memory protection
  - Privileged mode
  - System call (TRAP)
  - Exceptions & interrupts
  - Interacting with I/O devices
- Multiprocessor support
  - "Atomic" operations for synchronization
- Data-level parallelism
  - Pack many values into a wide register
    - Intel's SSE2: four 32-bit float-point values into 128-bit register
  - Define parallel operations (four "adds" in one cycle)

The RISC vs. CISC Debate

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"
  - Examples: x86, VAX, Motorola 68000, etc.
- Philosophical war (one of several) started in mid 1980's
  - RISC "won" the technology battles
  - CISC won the high-end commercial war (1990s to today)
    - Compatibility a stronger force than anyone (but Intel) thought
  - RISC won the embedded computing war

The Setup

- Pre 1980
  - Bad compilers (so assembly written by hand)
  - Complex, high-level ISAs (easier to write assembly)
  - Slow multi-chip micro-programmed implementations
    - Vicious feedback loop
- Around 1982
  - Moore's Law makes single-chip microprocessor possible…
    - **…but only for small, simple ISAs**
  - Performance advantage of this "integration" was compelling
  - 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
- **Many registers** (compilers are better at using them)
  - CISC: few registers

CISCs and RISCs

- The CISCs: x86, VAX (**V**irtual **A**ddress e**X**tension to PDP-11)
  - 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
  - x86: "Difficult to explain and impossible to love"
- The RISCs: MIPS, PA-RISC, SPARC, PowerPC, Alpha
  - 32-bit instructions
  - 32 integer registers, 32 floating point registers, load-store
  - 64-bit virtual address space
  - Few addressing modes (Alpha has one, SPARC/PowerPC have more)
  - Why so many basically similar ISAs? Everyone wanted their own

The Debate

- RISC argument
  - 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
  - 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 paramount

Compatibility

- No-one buys new hardware… if it requires new software
  - Intel greatly benefited from this (IBM, too)
  - 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

Intel's Compatibility Trick: RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
  - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC **μops** in hardware

  `push $eax`

  becomes (we think, uops are proprietary)

  ```
  store $eax [$esp-4]
  addi $esp,$esp,-4
  ```

  + Processor maintains **x86 ISA externally for compatibility**

  + But executes **RISC μISA internally for implementability**
  - Given translator, x86 almost as easy to implement as RISC
    - Intel implemented out-of-order before any RISC company
    - Also, OoO also benefits x86 more (because ISA limits compiler)
  - Idea co-opted by other x86 companies: AMD and Transmeta

More About Micro-ops

- Even better? Two forms of hardware translation
  - Hard-coded logic: fast, but complex
  - Table: slow, but "off to the side", doesn't complicate rest of machine
- x86: average 1.6 μops / x86 insn
  - Logic for common insns that translate into 1–4 μops
  - Table for rare insns that translate into 5+ μops
- x86-64: average 1.1 μops / x86 insn
  - More registers (can pass parameters too), fewer `pushes`/`pops`
  - Core2: logic for 1–2 μops, Table for 3+ μops?
- More recent: "macro-op fusion" and "micro-op fusion"
  - Intel's recent processors fuse certain instruction pairs

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - **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), Rosetta (PowerPC on x86)
  - Performance overheads reasonable (many recent advances)
  - Transmeta's "code morphing" translation layer
    - Performed with a software layer below OS
    - Looks like x86 to the OS & applications, different ISA underneath
- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime)

Ultimate Compatibility Trick

- Support old ISA by…
  - …having a simple processor for that ISA somewhere in the system
  - How first Itanium supported x86 code
    - x86 processor (comparable to Pentium) on chip
  - How PlayStation2 supported PlayStation games
    - Used PlayStation processor for I/O chip **& emulation**

Current Winner (Revenue): CISC

- 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 competitive 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

Current Winner (Volume): RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - 1.2 billion units sold in 2004 (>50% of all 32/64-bit CPUs)
  - Low-power and **embedded** devices (iPod, for example)
    - Significance of embedded? ISA Compatibility less powerful force
- 32-bit RISC ISA
  - 16 registers, PC is one of them
  - Many addressing modes, e.g., auto increment
  - Condition codes, each instruction can be conditional
- Multiple implementations
  - X-scale (design was DEC's, bought by Intel, sold to Marvel)
  - Others: Freescale (was Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

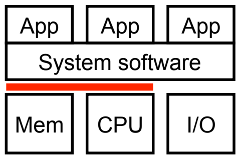
Aside: Post-RISC -- VLIW and EPIC

- ISAs explicitly targeted for multiple-issue (superscalar) cores
  - VLIW: Very Long Insn Word
  - Later rebranded as "EPIC": Explicitly Parallel Insn Computing
- Intel/HP IA64 (Itanium): 2000
  - EPIC: 128-bit 3-operation bundles
  - 128 64-bit registers
  - + Some neat features: Full predication, explicit cache control
    - Predication: every instruction is conditional (to avoid branches)
  - – But lots of difficult to use baggage as well: software speculation
    - Every new ISA feature suggested in last two decades
  - – Relies on younger (less mature) compiler technology
  - – Not doing well commercially

Redux: Are ISAs Important?

- Does "quality" of ISA actually matter?
  - Not for performance (mostly)
    - Mostly comes as a design complexity issue
    - Insn/program: everything is compiled, compilers are good
    - Cycles/insn and seconds/cycle: μISA, many other tricks
  - What about power efficiency?
    - Maybe
    - ARMs are most power efficient today..
      - …but Intel is moving x86 that way (e.g, Intel's Atom)
- Does "nastiness" of ISA matter?
  - Mostly no, only compiler writers and hardware designers see it
- Even compatibility is not what it used to be
  - Software emulation

Summary



- What is an ISA?
  - A functional contract
- All ISAs are basically the same
  - But many design choices in details
  - Two "philosophies": CISC/RISC
- Good ISA enables high-performance
  - At least doesn't get in the way
- Compatibility is a powerful force
  - Tricks: binary translation, μISAs

- Next: single-cycle datapath/control