Instruction Set Architecture (ISA)

- What is an ISA?
  - And what is a good ISA?
- Aspects of ISAs
  - With examples: LC3, MIPS, x86
  - RISC vs. CISC
  - Compatibility is a powerful force
  - Tricks: binary translation, μISAs

Readings

- 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

- Basic structure of all modern ISAs
  - Processor logically executes loop at left
    - **Atomically**: insn X finishes before insn X+1 starts
- Program order: total order on dynamic insns
  - Order and **named storage** define computation
  - Value flows from insn X to Y via storage A iff...
    - A=X’s output, X=Y’s input, Y after X in program order
    - No interceding insn Z where A=Z’s output
- Convenient feature: **program counter** (PC)
  -Insn itself at memory[PC]
  - Next PC is PC++ unless insn says otherwise

LC3

- LC3 highlights
  - 1 datatype: 16-bit 2C integer
  - Addressible of memory locations: 16 bits
  - Instructions are 16 bits
  - 3 arithmetic operations: add, and, not
    - Build everything else from these
  - 8 registers, load-store model, three addressing modes
  - Condition codes for branches
  - Support for traps and interrupts
- Why is LC3 this way? (and not some other way?)
- What are some other options?

P37X

- Similar to LC3 in some ways (but better)
- Similarities
  - 16-bit data types
  - 16-bit instructions, four-bit opcode
  - Similar TRAPs and devices
- Differences
  - More ALU ops: Add, Sub, Mul, Or, Not, And, Xor, Shift Left/Right
  - No LDI, STI (indirect load/stores)
  - No condition codes
- Designed for CIS372
  - PennSim supports this with a command-line mode switch
Some Other ISAs

- LC3 & P37X has the basic features of a real-world ISA
  - Lacks a good bit of realism
  - Only 16-bit
  - Not byte addressable
  - Fewer arithmetic insns (more for LC3 than P37X)
  - Little support for system software, none for multiprocessing

- Two real world ISAs
  - Intel x86 (IA32): a CISC ISA
  - MIPS: a "real world" RISC ISA (also used in book)

- P37X: ISA used in 372
  - A more RISC’y LC3

- What is this RISC/CISC thing?

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: \( \text{seconds/program} \)
  - Convenient to factor into three pieces
    - \( \text{insns/program} \times \text{cycles/insn} \times \text{seconds/cycle} \)
      - 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 HLLs to assembly functionally
  - Deterministic and fast compile time (\( \text{gcc} \ -\text{o0} \))
  - "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 (\( \text{gcc} \ -\text{o4} \))
    - 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

RISC/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

ISA Basics

- Aspects of ISAs
  - VonNeumann model
  - Data types and operations
  - Operand model
  - Control
  - Encoding
  - Operating system support
  - Multiprocessing support

- Examples
  - LC3 (P37X)
  - MIPS
  - x86
**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
  - Binary-coded decimal (BCD)

**LC3/MIPS/x86 Operations and Datatypes**

- **LC3**
  - 16-bit integer: add, and, not
  - P37X also has sub, mul, 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...
  - BCD!!! (not really used obviously)

**Where Does Data Live?**

- **Memory**
  - Fundamental storage space
  - Processor w/o memory = table w/o chairs
- **Registers**
  - Faster than memory, quite handy
  - Most processors have these too
- **Immediates**
  - Values spelled out as bits in insns
  - Input only

**How Many Registers?**

- Registers faster than memory, have as many as possible?
  - No
  - One reason registers are faster: there are few of them
    - Small is fast (hardware truism)
  - Another: they are directly addressed (no address calc)
    - More of them, means larger specifiers
    - Fewer registers per insn or indirect addressing
  - Not everything can be put in registers
    - Structures, arrays, anything pointed-to
    - Compilers are getting better at putting more things in
      - More registers means more saving/restoring
### LC3/MIPS/x86 Registers

- **LC3/P37X**
  - 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 (EM64T)**
  - 16 64-bit integer registers
  - 16 128-bit floating-point registers

### How Much Memory?

- What does “64-bit” in 64-bit ISA mean?
  - Each program can address (i.e., use) $2^{64}$ bytes
  - 64 is the virtual address (VA) size
  - Alternative (wrong) definition: width of arithmetic operations

- Most critical, inescapable ISA design decision
- Too small?
  - Limits the lifetime of ISA
  - May require nasty hacks to overcome (e.g., x86 segments)

- All ISAs moving to 64 bits (if not already there)

### LC3/MIPS/x86 Memory Size

- **LC3/P37X**
  - 16-bit ($2^{16}$ 16-bit words)

- **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 insn
  - Some insns 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
LC3/MIPS/x86 Addressing Modes

- LC3
  - **Displacement**: $R1 + \text{offset (6-bit)}$
  - **PC-displacement**: $\text{PC} + \text{offset (9-bit)}$
  - **Memory-indirect/PC-displacement**: $\text{mem}[[\text{PC}] + \text{offset (9-bit)}]$
    - Nasty, requires accessing memory twice, P37X doesn't have this

- MIPS
  - **Displacement**: $R1 + \text{offset (16-bit)}$
  - Experiments showed this covered 80% of accesses on VAX

- x86 (MOV instructions)
  - **Absolute**: zero + offset (8/16/32-bit)
  - **Register indirect**: $R1$
  - **Indexed**: $R1 + R2$
  - **Displacement**: $R1 + \text{offset (8/16/32-bit)}$
  - **Scaled**: $R1 + (R2 \times \text{Scale}) + \text{offset (8/16/32-bit)}$
    - Scale = 1, 2, 4, 8

How Many Explicit Operands / ALU Insn?

- **Operand model**: how many explicit operands / ALU insn?
  - **3**: general-purpose
    - $\text{add } R1, R2, R3$ means $[R1] = [R2] + [R3]$ (MIPS uses this)
  - **2**: multiple explicit accumulators (output doubles as input)
    - $\text{add } R1, R2$ means $[R1] = [R1] + [R2]$ (x86 uses this)
  - **1**: one implicit accumulator
    - $\text{add } R1$ means $\text{ACC} = \text{ACC} + [R1]$
  - **0**: hardware stack (like Java bytecodes)
    - $\text{add}$ means $\text{STK}[-\text{TOS}++] = \text{STK}[-\text{TOS}] + \text{STK}[-\text{TOS}]$
  - **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
  - $\text{load } r1, 8(r2)$ means $[R1] = \text{mem}[[R2] + 8]$
  - $\text{store } r1, 8(r2)$ means $\text{mem}[[R2] + 8] = [R1]$

- **Accumulator**: load/store
  - $\text{load } 8(r2)$ means $\text{ACC} = \text{mem}[[R2] + 8]$
  - $\text{store } 8(r2)$ means $\text{mem}[[R2] + 8] = \text{ACC}$

- **Stack**: push/pop
  - $\text{push } 8(r2)$ means $\text{STK}[\text{TOS}++] = \text{mem}[[R2] + 8]$
  - $\text{pop } 8(r2)$ means $\text{mem}[[R2] + 8] = \text{STK}[\text{TOS}--]$

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
LC3/MIPS/x86 Operand Models

- **LC3**
  - Integer: 8 accumulator registers
  - 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 sucked 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

- **Testing conditions**: whether to jump at all
  - For (conditional) branches only

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, LC3)**
  - `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, P37X)**
  - `set-less-than R2,R1,10`
  - `branch-not-equal-zero R2,target`
  - Additional insns
  - One ALU per insn, explicit dependence
LC3, MIPS, x86 Control Transfers

- **LC3**
  - 9-bit offset PC-relative branches/jumps (uses condition codes)
  - 11-bit offset PC-relative calls and indirect calls
- **P37X**
  - 6-bit offsets PC-relative simple branches (uses register for condition)
  - 12-bit offset on calls and unconditional branches
- **MIPS**
  - 16-bit offset PC-relative conditional branches (uses register for condition)
  - Compare 2 regs: `beq`, `bne` or reg to 0: `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)
  - 8/16-bit target absolute jumps and function calls (within segment)
  - Far jumps and calls (change code segment) for longer jumps

Length and Format

- **Length**
  - Fixed length
    - Most common is 32 bits
      + Simple implementation: next PC = PC+4
      + Longer reach for branch/jump targets
        - Code density: 32 bits to increment a register by 1?
    - Variable length
      - Complex implementation
      - Code density
    - Compromise: two lengths
      - MIPS16 or ARM’s Thumb
  - **Encoding**
    - A few simple encodings simplify decoder

LC3/MIPS/x86 Length and Format

- **LC3**: 2-byte insns, 3 formats (P37X is 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* SI* Disp*(1-4) Imm*(1-4)`

Operating System Support

- ISA support required to implement an operating system
- **At least two privilege modes**: `user` (low), `kernel` (high)
  - Some operations, storage locations accessible in all modes
  - Others accessible only in high privilege mode
    - Deal with I/O, exceptions, virtual memory, privilege itself
    - Anything that allows one process to interfere with another
- Support for safely up-grading and down-grading privilege
  - Programmatically: system calls
  - Transparently: interrupts
Traps and System Calls

- What if a user process wanted to access an I/O device?
  - Can’t actually "call" kernel procedures
  - Kernel is "shared" by all user applications → a separate process
  - Should not be allowed to "call" or "jump" into arbitrary kernel code
  - Should not be allowed to upgrade privilege outside of kernel

- How does this work then?
  - Kernel publishes a set of service codes (not function addresses)
  - User processes use special insn to invoke desired service
  - TRAP, INTERRUPT, SYSCALL: a (process-changing) call only...
    - Specifies function "code" rather than address
    - Upgrades privilege: only way to do this
  - Return-from-interrupt: a (process-changing) return only...
    - Downgrades privilege

LC3/MIPS/x86 OS Support

- LC3
  - Trap, return from interrupt
  - Interrupts supported but not used in CIS 240

- MIPS
  - Trap, return from trap
  - “Exception coprocessor”
  - Interrupts

- X86
  - Trap, return from trap
  - Exception flags
  - Multi-level interrupts

Multiprocessing Support

- ISA support also required for multiprocessing
  - Memory model
  - Atomic "conditional reg/mem swap” insns
  - “Fence” insns

- LC3
  - No multiprocessing support

- MIPS/x86
  - Yes, please

  More about this later

RISC and CISC

- RISC: reduced-instruction set computer
  - Coined by Patterson in early 80’s
    - Berkeley RISC-I, Stanford MIPS, IBM 801
    - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC

- CISC: complex-instruction set computer
  - Term didn’t exist before "RISC”
  - Examples: x86, Motorola 68000, VAX (makes x86 look like LC3), etc.

- Religious war started in mid 1980’s
  - RISC “won” the (technology) battle, CISC won the (commercial) war
  - Compatibility a stronger force than anyone (but Intel) thought
  - Intel & AMD beat RISC at its own game
Pre 1980: The Setup

- Vicious feedback pendulum
  - Bad compilers ↔ complex ISAs ↔ slow multi-chip designs
  - Assembly commonly written by hand

Complex ISAs ↔ Bad Compilers

- Who is generating assembly code?
- Humans like high-level “CISC” ISAs (close to HLLs)
  + Can “concretize” (“drill down”): move down a layer
  + Can “abstract” (“see patterns”): move up a layer
  - Can deal with few things at a time → like things at a high level

- Computers (compilers) like low-level “RISC” ISAs
  + Can deal with many things at a time → can do things at any level
  + Can “concretize”: 1-to-many lookup functions (databases)
  - Difficulties with abstraction: many-to-1 lookup functions (AI)
    - Translation should move strictly “down” levels

- Stranger than fiction
  - People once thought computers would execute HLLs directly

Complex ISAs ↔ Slow Implementations

- Complex ISAs have nasty datapaths
- Nasty datapaths are difficult to pipeline
  - And pipelining doesn’t help that much
- If you aren’t going to pipeline, you want a high-level ISA
  - To amortize fetch/decode

Early 1980s: The Tipping Point

- Moore’s Law makes single-chip microprocessor possible...
  - ...but only for small, simple ISAs
- Performance advantage of “integration” was compelling

- RISC manifesto: create ISAs that...
  + Simplify implementation
  + Facilitate optimizing compilation
  + Some guiding principles (“tenets”)
    - Single cycle execution/hard-wired control
    - Fixed instruction length, format
    - Lots of registers, load-store architecture, few addressing modes

- No equivalent “CISC manifesto”
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 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
- Solution? Translate x86 to RISC **uops** 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
  + Executes **RISC µISA internally for datapath implementation**
  ```
- Given translator, x86 almost as easy to implement as RISC
  - Intel implemented out-of-order before any RISC company
  - Idea co-opted by other x86 companies: AMD and Transmeta
  - The one company that resisted (Cyrix) couldn’t keep up

More About Uops

**Even better? Two forms of hardware translation**
- Optimized logic: for common insns that translate into 1–4 uops
  - Fast
  - Complex
- Table: for rare insns or nasty insns that translate into 5+ uops
  - Slow
  - “Off to the side”, doesn’t complicate rest of machine

- x86: average 1.6 uops / x86 insn
- x86-64: average 1.1 uops / x86 insn
  - More registers (can pass parameters too), fewer **pushes/pops**
  - Speculation about Core 2: PLA for 1–2 uops, Table for 3+ uops?
Transmeta’s Take: Code Morphing

- **Code morphing**: x86 translation 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

How x86 Won the Commercial War

- x86 was first 16-bit chip by ~2 years
- IBM put it into its PCs: there was no competing choice
- Rest is Moore’s Law, 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 keeps pressure on x86 performance

Actually, The Volume Winner is 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)

- 32-bit RISC ISA
  - 16 registers (PC is one of them: to branch, just write to the PC)
  - Many addressing modes, e.g., auto increment
  - Predication: Condition codes, each instruction can be conditional

- Multiple compatible implementations
  - Intel’s X-scale (original design was DECs, bought by Intel)
  - Others: Freescale (was Motorola), IBM, Texas Instruments, Nintendo, STMicroelectronics, Samsung, Sharp, Philips, etc.

Redux: Are ISAs Important?

- Does “quality” of ISA actually matter?
  - Not for performance (mostly)
  - Mostly comes as a design complexity issue
  - Insns/program: everything is compiled, compilers are good
  - Cycles/insn and seconds/cycle: μISA, many other tricks

- Does “nastiness” of ISA matter?
  - No, only compiler writers and hardware designers see it

- Even compatibility is not what it used to be...
  - Software emulation
Compatibility Trap Door

- “Trap”: add some ISA feature for 5% gain
  - Must support feature forever... even if gain turns to loss
  - Classic: SPARC’s register windows (hardware activation records)
- Trap: insn makes low-level “function call” to OS handler
  - Compatibility’s friend
- Backward compatibility: rid yourself of some ISA mistakes
  - New design: “mistake feature” opcodes trap, emulated in software
  - Performance (of that feature) suffers
    - Legal: ISA says nothing about performance
    - Actually good: feature will be used less (“deprecation cycle”)
- Forward compatibility
  - Reserve set of trap opcodes (don’t define uses)
  - Add ISA functionality by overloading traps
  - Release firmware patch to “add” to old implementation

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: FXI32 (x86 on Alpha), Rosetta (PowerPC on x86)
    - Performance overheads not that high
- 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

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