Instruction Set Architecture (ISA)

- What is an ISA?
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two “philosophies”: CISC/RISC
    - Difference is blurring
- Good ISA...
  - Enables high-performance
  - At least doesn’t get in the way
- Compatibility is a powerful force
  - Tricks: binary translation, μISAs
Readings

• Suggested reading:
  • “The Evolution of RISC Technology at IBM”
    by John Cocke and V. Markstein

Execution Model
Program Compilation

**App** | **App** | **App**
---|---|---
**System software**

**Mem** | **CPU** | **I/O**

- **Program** written in a “high-level” programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
  - Compiler itself another application ... who compiled compiler?

```plaintext
int array[100], sum;
void array_sum() {
    for (int i=0; i<100; i++ ) {
        sum += array[i];
    }
}
```

Assembly & Machine Language

**App** | **App** | **App**
---|---|---
**System software**

**Mem** | **CPU** | **I/O**

- **Assembly language**
  - Human-readable representation
- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in “hex”)
- **Assembler**
  - Translates assembly to machine

<table>
<thead>
<tr>
<th>Machine code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x9A00</td>
<td>CONST R5, #0</td>
</tr>
<tr>
<td>x9200</td>
<td>CONST R1, array</td>
</tr>
<tr>
<td>xD320</td>
<td>HICONST R1, array</td>
</tr>
<tr>
<td>x9464</td>
<td>CONST R2, sum</td>
</tr>
<tr>
<td>x520</td>
<td>HICONST R2, sum</td>
</tr>
<tr>
<td>x6640</td>
<td>LDR R3, R1, #0</td>
</tr>
<tr>
<td>x6880</td>
<td>LDR R4, R2, #0</td>
</tr>
<tr>
<td>x18C4</td>
<td>ADD R4, R3, R4</td>
</tr>
<tr>
<td>x7880</td>
<td>STR R4, R2, #0</td>
</tr>
<tr>
<td>x1261</td>
<td>ADD R1, R1, #1</td>
</tr>
<tr>
<td>x1BA1</td>
<td>ADD R5, R5, #1</td>
</tr>
<tr>
<td>x2B64</td>
<td>CPI R5, #100</td>
</tr>
<tr>
<td>x03F8</td>
<td>BRn array_sum_loop</td>
</tr>
</tbody>
</table>

Example is in “LC4” a toy **instruction set architecture**, or ISA
Example Assembly Language & ISA

- **MIPS**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 64 registers
    - 32 integer, 32 floating point
  - ~100 different insns

Example code is MIPS, but all ISAs are similar at some level

Instruction Execution Model

- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Called "instruction pointer" in x86
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its inputs (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its outputs (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just “data in memory”**
  - Makes computers programmable (“universal”)
What is an ISA?

What Is An ISA?

- **ISA (instruction set architecture)**
  - A well-defined hardware/software interface
  - The "contract" between software and hardware
    - Functional definition of storage locations & operations
      - Storage locations: registers, memory
      - Operations: add, multiply, branch, load, store, etc
    - Precise description of how to invoke & access them
  - Not in the "contract": non-functional aspects
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less

- **Instructions**
  - Bit-patterns hardware interprets as commands
  - 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**
  - Often called VonNeuman, but in ENIAC before
- **Program order**: total order on dynamic insns
  - Order and **named storage** define computation
- **Convenient feature**: program counter (PC)
  - Insn itself stored in memory at location pointed to by PC
  - Next PC is next insn unless insn says otherwise
- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
  - Implementations can break this constraint physically
  - But must maintain illusion to preserve correctness
ISA Design Goals

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?

- **Performance/Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design low-cost implementations?

- **Compatibility**
  - Easy to maintain as languages, programs, and technology evolve?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, ...
Programmability

- Easy to express programs efficiently?
  - For whom?

  - Before 1980s: human
    - Compilers were terrible, most code was hand-assembled
    - Want high-level coarse-grain instructions
      - As similar to high-level language as possible

  - After 1980s: 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

- This shift changed what is considered a “good” ISA...

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
  - Special-purpose registers: complicate compiler optimizations
  - Difficult to interrupt instructions: complicate many things
    - Example: memory copy instruction
Performance, Performance, Performance

- How long does it take for a program to execute?
  - Three factors

1. How many insn must execute to complete program?
  - **Instructions per program** during execution
  - “Dynamic insn count” (not number of “static” insns in program)

2. How quickly does the processor “cycle”?
  - **Clock frequency** (cycles per second) 1 gigahertz (Ghz)
  - or expressed as reciprocal, **Clock period** nanosecond (ns)
  - Worst-case delay through circuit for a particular design

3. How many **cycles** does each instruction take to execute?
  - **Cycles per Instruction** (CPI) or reciprocal, **Insn per Cycle** (IPC)

<table>
<thead>
<tr>
<th>Execution time = (instructions/program) * (seconds/cycle) * (cycles/instruction)</th>
</tr>
</thead>
</table>

Computer Architecture  | Prof. Milo Martin  | Instruction Sets

Maximizing Performance

<table>
<thead>
<tr>
<th>Execution time = (instructions/program) * (seconds/cycle) * (cycles/instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 billion instructions) * (1ns per cycle) * (1 cycle per insn) = 1 second</td>
</tr>
</tbody>
</table>

- Instructions per program:
  - Determined by program, compiler, instruction set architecture (ISA)

- Cycles per instruction: “CPI”
  - Typical range today: 2 to 0.5
  - Determined by program, compiler, ISA, micro-architecture

- Seconds per cycle: “clock period”
  - Typical range today: 2ns to 0.25ns
  - Reciprocal is frequency: 0.5 Ghz to 4 Ghz (1 Htz = 1 cycle per sec)
  - Determined by micro-architecture, technology parameters

- For minimum execution time, minimize each term
  - Difficult: **often pull against one another**
Example: Instruction Granularity

<table>
<thead>
<tr>
<th>Execution time = (instructions/program) * (seconds/cycle) * (cycles/instruction)</th>
</tr>
</thead>
</table>

- **CISC** (Complex Instruction Set Computing) ISAs
  - Big heavyweight instructions (lots of work per instruction)
    + Low “insns/program”
    - Higher “cycles/insn” and “seconds/cycle”
      - We have the technology to get around this problem

- **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

Compiler Optimizations

- Primarily goal: reduce instruction 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
Compatibility

- In many domains, ISA must remain compatible
  - IBM’s 360/370 (the first “ISA family”)
  - Another example: Intel’s x86 and Microsoft Windows
    - x86 one of the worst designed ISAs EVER, but survives
- **Backward compatibility**
  - New processors supporting old programs
    - Can’t drop features (**caution in adding new ISA features**)
    - Or, update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs
    - Include a “CPU ID” so the software can test of features
    - Add ISA hints by overloading no-ops (example: x86’s PAUSE)
    - New firmware/software on old processors to emulate new insn

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 advances over the years)

- **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)
  - NVIDIA’s “PTX”
Ultimate Compatibility Trick

- Support old ISA by...
  - ...having a simple processor for that ISA somewhere in the system
  - How did PlayStation2 support PlayStation1 games?
    - Used PlayStation processor for I/O chip & emulation

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 averages 3 bytes (ranges from 1 to 16)
      - 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 nasty piece of logic

Operations and Datatypes

- **Datatypes**
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1’s

- All processors support
  - Integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64-bit)

- More recently, most processors support
  - “Packed-integer” insns, e.g., MMX
  - “Packed-floating point” insns, e.g., SSE/SSE2/AVX
  - For “data parallelism”, more about this later

- Other, infrequently supported, data types
  - Decimal, other fixed-point arithmetic
Where Does Data Live?

- **Registers**
  - “short term memory”
  - Faster than memory, quite handy
  - Named directly in instructions

- **Memory**
  - “longer term memory”
  - Accessed via “addressing modes”
    - Address to read or write calculated by instruction

- **“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 registers, means more bits per register in instruction
  - Thus, fewer registers per instruction or larger instructions

- **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**
      - Across function calls, traps, and context switches

- **Trend toward more registers:**
  - 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)
Memory Addressing

- **Addressing mode**: way of specifying address
  - Used in memory-memory or load/store instructions in register ISA

- **Examples**
  - **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?

Addressing Modes Examples

- **MIPS**
  - **I-type**
  - | Op(6) | Rs(5) | Rt(5) | Immed(16) |
  - **Displacement**: R1+offset (16-bit)
  - Why? Experiments on VAX (ISA with every mode) found:
    - 80% use small displacement (or displacement of zero)
    - Only 1% accesses use displacement of more than 16bits

- Other ISAs (SPARC, x86) have reg+reg mode, too
  - Impacts both implementation and insn count? (How?)

- **x86 (MOV instructions)**
  - **Absolute**: zero + offset (8/16/32-bit)
  - **Register indirect**: R1
  - **Displacement**: R1+offset (8/16/32-bit)
  - **Indexed**: R1+R2
  - **Scaled**: R1 + (R2*Scale) + offset(8/16/32-bit)  Scale = 1, 2, 4, 8
Access Granularity & Alignment

- **Byte addressability**
  - An address points to a byte (8 bits) of data
  - The ISA’s minimum granularity to read or write memory
  - ISAs also support wider load/stores
    - “Half” (2 bytes), “Longs” (4 bytes), “Quads” (8 bytes)

|---------------------|---------------------|---------------------|

However, physical memory systems operate on even larger chunks

|--------------------|---------------------|---------------------|

- **Access alignment**: if address % size is not 0, then it is “unaligned”
  - A single unaligned access may require multiple physical memory accesses

Handling Unaligned Accesses

- **Access alignment**: if address % size is not 0, then it is “unaligned”
  - A single unaligned access may require multiple physical memory accesses

- **How do handle such unaligned accesses?**
  1. Disallow (unaligned operations are considered illegal)
    - MIPS takes this route
  2. Support in hardware? (allow such operations)
    - x86 allows regular loads/stores to be unaligned
      - Unaligned access still slower, adds significant hardware complexity
  3. Trap to software routine? (allow, but hardware traps to software)
    - Simpler hardware, but high penalty when unaligned
  4. In software (compiler can use regular instructions when possibly unaligned)
    - Load, shift, load, shift, and (slow, needs help from compiler)
  5. MIPS? ISA support: unaligned access by compiler using two instructions
    - Faster than above, but still needs help from compiler
      ```
lwl @XXXX10; lwr @XXXX10
```
Operand Model: Register or Memory?

- "Load/store" architectures
  - Memory access instructions (loads and stores) are distinct
  - Separate addition, subtraction, divide, etc. operations
  - Examples: MIPS, ARM, SPARC, PowerPC

- Alternative: mixed operand model (x86, VAX)
  - Operand can be from register or memory
  - x86 example: `addl 100, 4(%eax)`
    - 1. Loads from memory location [4 + %eax]
    - 2. Adds "100" to that value
    - 3. Stores to memory location [4 + %eax]
  - Would requires three instructions in MIPS, for example.

How Much Memory? Address Size

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

- 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 Pentium4)

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

- Default next-PC is PC + sizeof(current insn)
  - Branches and jumps can change that
- **Computing targets**: where to jump to
  - For all branches and jumps
  - PC-relative: for branches and jumps with function
  - Absolute: for function calls
  - Register indirect: for returns, switches & dynamic calls
- **Testing conditions**: whether to jump at all
  - Implicit condition codes or “flags” (x86)
    - `cmp R1,10` // sets “negative” flag
    - `branch-neg target`
  - Use registers & separate branch insns (MIPS)
    - `set-less-than R2,R1,10`
    - `branch-not-equal-zero R2,target`

ISAs Also Include Support For...

- Function calling conventions
  - Which registers are saved across calls, how parameters are passed
- 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
  - RISC-I (Patterson), 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 started in mid 1980’s
  - RISC “won” the technology battles
  - CISC won the high-end commercial space (1990s to today)
    - Compatibility was a strong force
    - RISC winning the embedded computing space
CISCs and RISCs

- The CISCs: x86, VAX (Virtual Address eXtension to PDP-11)
  - Variable length instructions: 1-321 bytes!!!
  - 14 registers + 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, ARM
  - 32-bit instructions
  - 32 integer registers, 32 floating point registers
    - ARM has 16 registers
  - Load/store architectures with few addressing modes
  - Why so many basically similar ISAs? Everyone wanted their own

Historical Development

- 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
- RISC manifesto: create ISAs that...
  - Simplify single-chip implementation
  - Facilitate optimizing compilation
The RISC Design Tenets

- **Single-cycle execution**
  - CISC: many multicycle operations
- **Hardwired (simple) control**
  - CISC: “microcode” for multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed-length instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
- **Many registers** (compilers can use them effectively)
  - CISC: few registers

---

RISC vs CISC Performance Argument

- **Performance equation:**
  - \((\text{instructions/program}) \times (\text{cycles/instruction}) \times (\text{seconds/cycle})\)

- **CISC** (Complex Instruction Set Computing)
  - Reduce “instructions/program” with “complex” instructions
    - But tends to increase “cycles/instruction” or clock period
  - 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 insn**
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

Intel’s x86 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 micro-ops (μops) in hardware**
  - `push $eax`
    - becomes (we think, uops are proprietary)
  - `store $eax, -4($esp)`
  - `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
    - “out-of-order” also helps x86 more (because ISA limits compiler)
    - Also used by other x86 implementations (AMD)
  - Different μops for different designs
    - **Not part of the ISA specification**, not publically disclosed
Potential Micro-op Scheme

- Most instructions are a **single** micro-op
  - Add, xor, compare, branch, etc.
  - Loads example: `mov -4(%rax), %ebx`
  - Stores example: `mov %ebx, -4(%rax)`
- Each memory access adds a micro-op
  - "addl -4(%rax), %ebx" is two micro-ops (load, add)
  - "addl %ebx, -4(%rax)" is three micro-ops (load, add, store)
- Function call (CALL) – 4 uops
  - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) – 3 uops
  - Adjust stack pointer, load return address from stack, jump register
- Again, just a basic idea, micro-ops are specific to each chip

Winner for Desktop PCs: CISC

- x86 was first mainstream 16-bit microprocessor by ~2 years
  - IBM put it into its PCs...
  - Rest is historical inertia, Moore’s law, and “financial feedback”
    - x86 is most difficult ISA to implement and do it fast but...
    - Because Intel sells the most **non-embedded** processors...
    - It hires more and better engineers...
    - Which help it 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
Winner for Embedded: RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - 3 billion units sold in 2009 (>60% of all 32/64-bit CPUs)
  - Low-power and embedded devices (phones, for example)
    - Significance of embedded? ISA Compatibility less powerful force
- 32-bit RISC ISA
  - 16 registers, PC is one of them
  - Rich addressing modes, e.g., auto increment
  - Condition codes, each instruction can be conditional
- ARM does not sell chips; it licenses its ISA & core designs
- ARM chips from many vendors
  - Qualcomm, Freescale (was Motorola), Texas Instruments, 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
    - 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)
  - Open question: can x86 be as power efficient as ARM?
- 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
  - Open question: will “ARM compatibility” be the next x86?
Instruction Set Architecture (ISA)

- What is an ISA?
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two “philosophies”: CISC/RISC
    - Difference is blurring
- Good ISA...
  - Enables high-performance
  - At least doesn’t get in the way
- Compatibility is a powerful force
  - Tricks: binary translation, µISAs