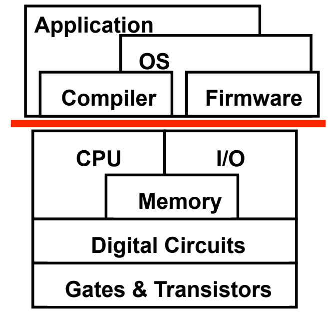
CIS 501
Computer Architecture

Unit 3: Instruction Set Architecture

Slides originally developed by Amir Roth with contributions by Milo Martin at University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

Instruction Set Architecture (ISA)



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

Readings

- H+P
  - Appendix B
  - Appendix J.2 (RISC ISAs) and J.3 (x86)
    - On the Book's CD
- 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

ISA Design Goals

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": 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
- 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

RISC vs CISC Foreshadowing

- Recall performance equation:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- **CISC** (Complex Instruction Set Computing)
  - Reduce "instructions/program" with "complex" instructions
    - But tends to increase CPI 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 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, Core2…

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)

Today's Semantic Gap

- Today's ISAs are actually 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

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
    - Easier to synthesize complex insns than to compare them

  - Rules of thumb
    - Regularity: "**principle of least astonishment**"
    - Orthogonality & composability
    - One-vs.-all

Compiler Optimizations

- Compilers do two things
- **Code generation**
  - Translate HLL to machine insns naively, one statement at a time
  - Canonical, there are compiler-generating programs
- **Optimization**
  - Transform insns to preserve meaning but improve performance
  - Active research area, but some standard optimizations
    - Register allocation, common sub-expression elimination, loop-invariant code motion, loop unrolling, function inlining, code scheduling (to increase insn-level parallelism), etc.

Compiler Optimizations

- Primarily reduce dynamic 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
  - Reduce cache misses
  - Reduce dependences between nearby insns (for parallelism)
    – 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

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
    - Example: memory copy instruction

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 (cumbersome)
    - 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

The Compatibility Trap

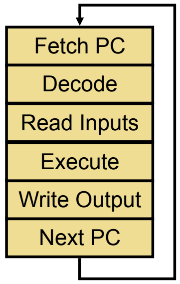
- Easy compatibility requires forethought
  - Temptation: use some ISA extension for 5% performance gain
  - Frequent outcome: gain diminishes, disappears, or turns to loss
    - **Must continue to support gadget for eternity**
  - Example: register windows (SPARC)
    - Adds difficulty to out-of-order implementations of SPARC
- Compatibility trap door
  - How to rid yourself of some ISA mistake in the past?
  - Make old instruction an "illegal" instruction on new machine
  - Operating system handles exception, emulates instruction, returns
    - Slow unless extremely uncommon for all programs

Aspects of ISAs

Aspects of ISAs

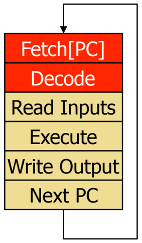
- **VonNeumann model**
  - Implicit structure of all modern ISAs
- Format
  - Length and encoding
- **Operand model**
  - Where (other than memory) are operands stored?
- Datatypes and operations
- Control
- Overview only
  - Read about the rest in the book and appendices

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
- More parallel alternatives have been proposed

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 nasty piece of logic

Examples Instruction Encodings

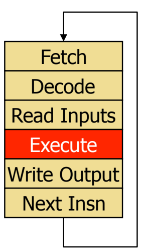
- MIPS
  - Fixed length
  - 32-bits, 3 formats, simple encoding
  - (MIPS16 has 16-bit versions of common insn for code density)

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
  - Variable length encoding (1 to 16 bytes)

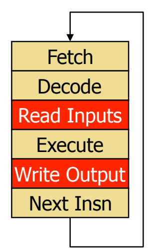
Prefix*(1-4)	Op	OpExt*	ModRM*	SIB*	Disp*(1-4)	Imm*(1-4)

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
  - Binary-coded decimal (BCD)

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 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)

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

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)

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 long (64-bit)
  - **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

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

- SPARC adds Reg+Reg mode
  - Why? What impact on both implementation and insn count?

Addressing Modes Examples

- MIPS
  - **Displacement**: R1+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+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"**

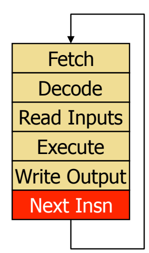
MIPS and x86 Operand Models

- MIPS
  - Integer: 32 32-bit general-purpose registers (load/store)
  - Floating point: same (can also be used as 16 64-bit registers)
  - 16-bit displacement addressing
- x86
  - Integer: 8 accumulator registers (reg-reg, reg-mem, mem-reg)
    - Can be used as 8/16/32 bits
  - Floating point: 80-bit **stack** (why x86 had slow floating point)
  - Displacement, absolute, reg indirect, indexed and scaled addressing
    - All with 8/16/32 bit constants (why not?)
  - Note: integer `push`, `pop` for managing software stack
  - Note: also reg-mem and mem-mem string functions in hardware
- x86-64 (i.e., IA32-EM64T)
  - Integer: **16 64-bit accumulator registers**
  - Floating point: **16 128-bit accumulator registers**

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

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)**

```
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

MIPS and x86 Control Transfers

- MIPS
  - 16-bit offset PC-relative conditional branches
    - **Uses register for condition**
  - Compare two regs: **`beq`**, **`bne`**
  - Compare reg to 0: **`bgtz`**, **`bgez`**, **`bltz`**, **`blez`**
  - Why?
    - More than 80% of branches are (in)equalities or comparisons to 0
    - Don't need adder for these cases (fast, simple)
    - OK to take two insns to do remaining branches
      - It's the uncommon case
  - Explicit "set condition into registers": **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, etc.
- x86
  - 8-bit offset PC-relative branches
    - **Uses condition codes**
  - Explicit compare instructions (and others) to set condition codes

MIPS Control Instructions

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

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

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

J-type	Op(6)	Target(26)

- Indirect jumps: **`jr`**

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

ISAs Also 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"
  - 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 fast 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-length 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, ARM
  - 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

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.

Current Winner (Revenue): CISC

- x86 was first 16-bit microprocessor 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

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

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**

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)
- **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)

Transmeta's Take: Code Morphing

- **Code morphing**: x86 translation in software
  - Crusoe was an x86 emulator, no actual x86 hardware anywhere
  - Only "code morphing" translation software written in native ISA
  - Native ISA is invisible to applications and even OS
  - Different Crusoe versions have (slightly) different ISAs: can't tell
- How was it done?
  - Code morphing software resides in boot read-only memory (ROM)
  - On startup, 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

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

ISA Research

- Compatibility killed ISA research for a while
  - But binary translation/emulation has revived it
- Current ISA-related projects
  - "WaveScalar" [Washington], "TRIPS EDGE" [Texas]
    - Explicit dataflow ISAs (vonNeumann alternatives)
  - "Mini-graphs: Application-Specific RISC/CISC Hybrid" [Penn]
    - A programmable μISA: μISA/binary-rewriting hybrid
    - Similar project at Michigan too
  - DISE: Dynamic Instruction Stream Editor" [Corliss, Lewis, Roth]
    - A programmable μISA: μISA/binary-rewriting hybrid
  - "Hardbound" [Devietti, Blundell, Martin, Zdancewic]
    - Hardware support for bounds checking C programs

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?**

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