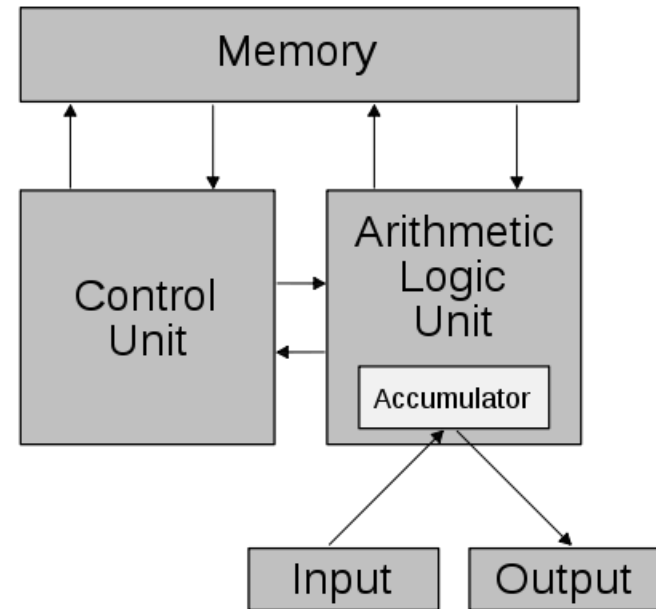


# Aspects of ISAs

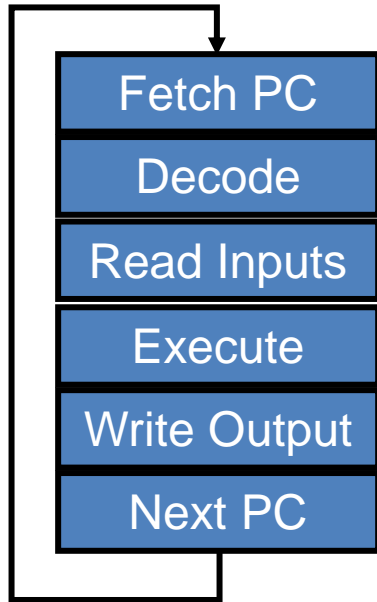
# Aspects of ISAs

## Begin with VonNeumann model

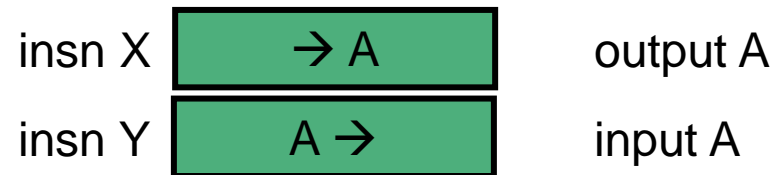
- Implicit structure of all modern ISAs
  - CPU + memory (data & insns)
  - Sequential instructions
- Format
  - Length and encoding
- **Operand model**
  - Where (other than memory) are operands stored?
- Datatypes and operations
- Control



# The Sequential Model



- Implicit model of all modern ISAs
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instruction
    - Next PC is PC++ (except for ctrl insns)
  - Order + **named storage** define computation  
Value flows from X to Y via storage A iff:

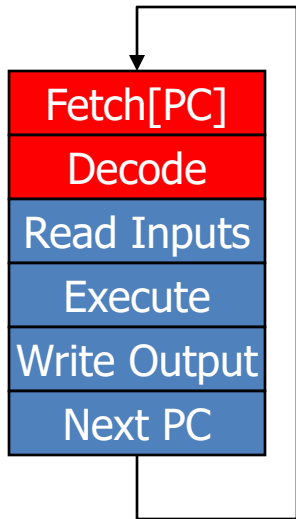


- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts
- More parallel alternatives have been proposed

# Instruction 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

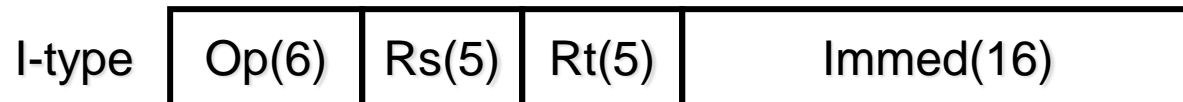
## Encoding

- A few simple encodings simplify decoder
  - x86 decoder one nasty piece of logic

# Example Instruction Encodings

## MIPS

- Fixed length
- 32-bits, 3 formats, simple encoding



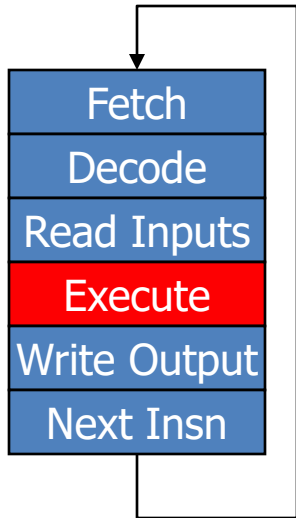
## x86

- Variable length encoding (1 to 15 bytes)



# Operations and Datatypes

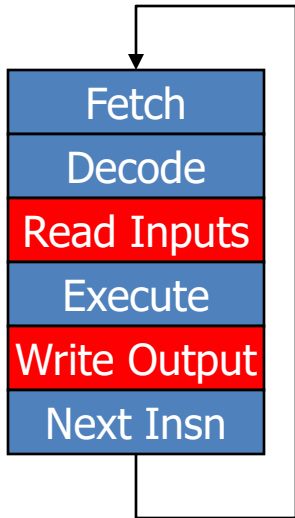
---



- **Datatypes**
  - S/W: attribute of data
  - H/W: attribute of operation, data is just 0/1's
- **All processors support**
  - 2's complement integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64 bit)
    - Intel has 80-bit floating-point
- **Most processors now support**
  - "Packed-integer" insns, *e.g.*, MMX
  - "Packed-fp" insns, *e.g.*, SSE/SSE2
  - For multimedia, more about these later
- **Processors 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
  - 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)
- Most 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
- More registers → **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

# How Are Memory Locations Specified?

---

- Registers are specified **directly**
  - Register names are short, 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?

# Memory Addressing

---

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

# Addressing Modes Examples

---

- 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:**  $R1 + (R2 * \text{Scale}) + \text{offset}$  (8/16/32-bit)  
Scale = 1, 2, 4, 8

2 more issues: alignment & endianness

# 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**)
  - **2**: multiple explicit accumulators (output also input)
    - add R1 , **R2** means  $[R2] = [R2] + [R1]$  (**x86**)
  - **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
- ISA w/register-only ALU insns are = **load-store architecture**

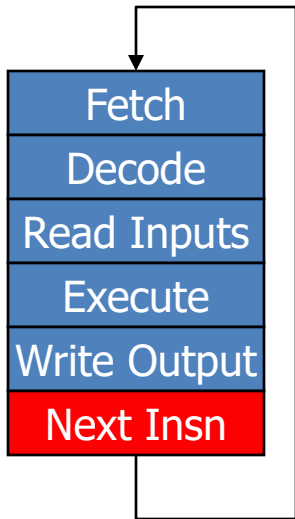
# Operand Model Pros and Cons

---

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

# Control Transfers

---



- Default next-PC is  $PC + \text{sizeof}(\text{current insn})$ 
  - Note: PC called IR (instruction register) in x86
- 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? (w/in fn vs outside)
  - Do you need to jump to a different place each time?
  - How many bits do you need to encode the target?
- **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
  - 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 (for condition & target address)

    - Extra latency

- **Implicit condition codes (x86)**

  - `cmp R1,10 // sets "negative" CC/flag`

  - `branch-neg target`

    - + More room for target, condition codes set "for free"

    - + Branch insn simple and fast

    - 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

    - > 80% of branches are (in)equalities/comparisons to 0

# 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: 4x32-bit float-point values in 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 80s
  - Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801 (Cocke), 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 1980s
  - 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)
- Around 1982
  - Moore's Law makes fast single-chip microprocessor possible... **...but only for small, simple ISAs**
  - Performance advantage of "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 (Virtual Address eXtension 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 1, SPARC/PowerPC more)
  - Why so many? Everyone wanted their own

# The Debate

---

- RISC argument
  - CISC is fundamentally handicapped by complexity
  - For a given technology, RISC 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, fixable 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

# Enter Micro-Ops (1)

---

Most instructions are a **single** micro-op, uop

- Add, xor, compare, branch, *etc.*
- Loads example: `mov -4(%rax), %ebx`
- Stores example: `mov %ebx, -4(%rax)`

Each operation on a memory location → micro-ops++

- `"addl -4(%rax), %ebx"` = 2 uops (load, add)
- `"addl %ebx, -4(%rax)"` = 3 uops (load, add, store)

What about address generation?

- **Simple** address generation: single micro-op
- **Complicated** (scaled addressing) & sometimes store addresses: calculated separately

# Enter Micro-Ops (2)

---

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 to return address

Other operations

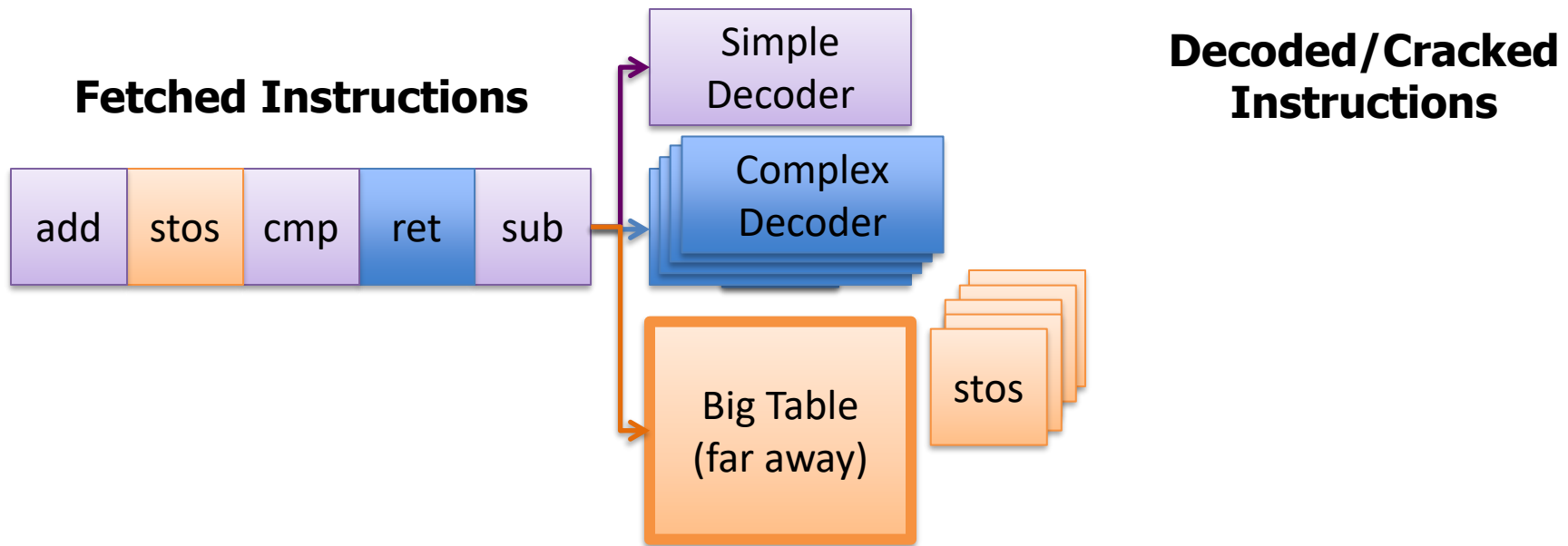
- String manipulations instructions
  - For example STOS is around six micro-ops, etc.

Micro-ops: part of the *microarchitecture*, not the architecture

# Cracking Macro-ops into Micro-ops

Two forms of  $\mu$ op “cracking”

- **Hard-coded logic:** fast, but expensive (for insn in few  $\mu$ ops)
  - Simple Decoder: 1  $\rightarrow$  1
  - Complex Decoder: 1  $\rightarrow$  2-4
    - 4x in size
- **Table Lookup:** slow, but “off to the side” (not shown)
  - $\rightarrow$  doesn’t complicate rest of machine
  - Handles *really* complicated instructions



# Micro-Op changes over time

---

x86 code is becoming more “RISC-like”.

IA32 → x86-64:

1. Double number of registers
2. Better function calling conventions
  - Result? Fewer pushes, pops, and complicated instructions  
~1.6  $\mu$ ops / macro-op → ~1.1  $\mu$ ops / macro-op

Fusion: Intel’s newest processors fuse certain instruction pairs

- **Macro-op fusion:** fuses “compare” and “branch” instructions
  - 2 macro-ops → 1 simple micro-op (uses simple decoder)
- **Micro-op fusion:** fuses ld/add pairs, fuses store “addr” & “data”
  - 1 complex micro-op → 1 simple micro-op (uses simple decoder)

# Ultimate Compatibility Trick

---

- Support old ISA with...
  - ...a simple processor for that ISA 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, optimize on-the-fly
  - Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime)
- **Transmeta's Code morphing**: x86 translation in software
  - Only "code morphing" translation software written in native ISA
  - Native ISA is invisible to applications and even OS
  - Guess who owns this technology now?

# RISC & CISC for Performance

---

Recall performance equation:

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

**CISC** (Complex Instruction Set Computing)

**RISC** (Reduced Instruction Set Computing)

	$\frac{\text{insns}}{\text{program}}$	$\frac{\text{cycles}}{\text{insn}}$	$\frac{\text{seconds}}{\text{cycle}}$	other
CISC				
RISC				



# RISC & CISC for Performance

Recall performance equation:

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

**CISC** (Complex Instruction Set Computing)

**RISC** (Reduced Instruction Set Computing)

	$\frac{\text{insns}}{\text{program}}$	$\frac{\text{cycles}}{\text{insn}}$	$\frac{\text{seconds}}{\text{cycle}}$	other
<b>CISC</b>	↓	↑	↑	+ Easy for assembly-level programmers + good code density
<b>RISC</b>	↑ hopefully not too much	↓	↓ <i>if designed aggressively</i>	+ smart compilers can help with insns/program