240 Review: Applications

- **Applications** (Firefox, iTunes, Skype, Word, Google)
  - Run on hardware ... but how?

240 Review: I/O

- Apps interact with us & each other via **I/O (input/output)**
  - With us: display, sound, keyboard, mouse, touch-screen, camera
  - With each other: disk, network (wired or wireless)
  - Most I/O proper is analog-digital and domain of EE
  - I/O devices present rest of computer a digital interface (1s and 0s)

Instruction Set Architecture (ISA)

- What is an ISA?
  - And what is a good ISA?
- Aspects of ISAs
  - With examples: LC4, MIPS, x86
- RISC vs. CISC
- Compatibility is a powerful force
  - Tricks: binary translation, µISAs

Readings

- Introduction
  - P&H, Chapter 1
- ISAs
  - P&H, Chapter 2, x86 info on CD
240 Review: OS

- I/O (& other services) provided by **OS (operating system)**
  - A super-app with privileged access to all hardware
  - Abstracts away a lot of the nastiness of hardware
  - Virtualizes hardware to isolate programs from one another
    - Each application is oblivious to presence of others
    - Simplifies programming, makes system more robust and secure
  - Privilege is key to this
- Commons OSes are Windows, Linux, MACOS

---

240 Review: ISA

- App/OS are software ... execute on hardware
- HW/SW interface is **ISA (instruction set architecture)**
  - A "contract" between SW and HW
  - Encourages compatibility, allows SW/HW to evolve independently
  - **Functional definition** of HW storage locations & operations
    - Storage locations: registers, memory
    - Operations: add, multiply, branch, load, store, etc.
  - **Precise description** of how to invoke & access them
    - Instructions (bit-patterns hardware interprets as commands)

---

240 Review: LC4

- **LC4**: a toy ISA you know
  - 16-bit ISA (what does this mean?)
  - 16-bit insns
  - 8 registers (integer)
  - ~30 different insns
  - Simple OS support
  - **Assembly language**
    - Human-readable ISA representation

---

371 Preview: MIPS

- **MIPS**: a real ISA (used in book)
  - 32/64-bit ISA
  - 32-bit insns
  - 64 registers (32 integer, 32 FP)
  - ~100 different insns
  - Full OS support

---

Similar? All ISAs are at some level
240 Review: C

- **C**: “high-level” programming language
  - Java, Python, C# much higher
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures

- **Compiler**: translates HLL to assembly
  - Straight translation is formulaic and canonical
  - Compiler also optimizes
  - Compiler itself another application ... who compiled compiler?

240 Review: Machine Language

- **Machine language**
  - Machine-readable ISA representation
  - 1s and 0s

- **Assembler**
  - Translates assembly to machine

- **Hex(adecimal)**
  - 1/0 short form
  - Each group of 4 bits is 0-F

---

240 Review: VonNeumann Model

- A CPU is essentially interpreter for an ISA
  - Logically executes VonNeumann loop

- **Program order**: total order on dynamic insns
  - Order & storage define computation

- **Atomic**: insn X finishes before insn X+1 starts
  - Actually, only has to “appear” atomic

- **Feature**: program counter (PC)
  -Insn itself at memory[PC]
  -Next PC is PC++ unless insn says otherwise

- **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 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 → Insns
  - ‘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)**
  - Insns itself at memory[PC]
  - Next PC is PC++ unless insn says otherwise

- Processor logically executes loop at left

- **Atomic**: insn X finishes before insn X+1 starts
  - Can break this constraint physically (pipelining)
  - But must maintain illusion to preserve programmer sanity

Where Does Data Live?

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

- **Memory**
  - Fundamental storage space
  - “longer term memory”

- **Immediates**
  - Values spelled out as bits in instructions
  - Input only
LC4

- LC4 highlights
  - 1 datatype: 16-bit 2C integer
  - Addressable memory locations, insns also 16 bits
  - Most arithmetic operations
  - 8 registers, load-store model, one addressing mode
  - Condition codes for branches

- Why is LC4 this way? (and not some other way?)
- What are some other options?

Real World Other ISAs

- LC4 has the basic features of a real-world ISA
  - Lacks a good bit of realism
    - Only 16-bit
    - Only one data type
    - Little support for system software, none for multiprocessing
      - Talk about these later on in semester

- Two real world ISAs
  - Intel x86
  - MIPS (used in book)

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

Implementability

- **Lends itself to high-performance implementations**
  - Every ISA can be implemented
  - Not every ISA can be implemented well

- **Background: CPU performance equation**
  - Execution time: \texttt{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)

  - For high performance all three factors should be low

ISAs & Performance

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

- A good ISA balances three three aspects

- One example:
  - Big complicated instructions:
    - Reduce “insn/program” (good!)
    - Increases “cycles/instruction” (bad!)
  - Simpler instructions
    - Reverse of above

Insns/Program: Compiler Optimizations

- Compilers do two things
  - Translate high-level languages to assembly functionally
    - Deterministic and fast compile time (\texttt{gcc -O0})
    - "Canonical": not an active research area
    - CIS 341

  - “Optimize” generated assembly code
    - “Optimize”? Hard to prove optimality in a complex system
      - In systems: “optimize” means improve... hopefully
    - Involved and relatively slow compile time (\texttt{gcc -O4})
      - Some aspects: reverse-engineer programmer intention
      - Not "canonical": being actively researched
    - CIS 570
Compiler Optimizations

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

Seconds/Cycle and Cycle/Insn: Hmmm...

- For simple “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

Compiler Optimization Example (LC4)

```
;; temp = *first
LDR R7, R5, #2 ; R7=first
LDR R4, R7, #0
STR R4, R5, #1
;; *first = *second
LDR R3, R5, #3 ; R3=second
LDR R2, R3, #0
LDR R7, R5, #2 ; redundant
STR R2, R7, #0
;; *second = temp
LDR R4, R5, #1
LDR R3, R5, #3 ; redundant
STR R4, R3, #0
```

- Left: common sub-expression elimination
  - Remove calculations whose results are already in some register
- Right: register allocation
  - Keep temporary in register across statements, avoid stack spill/fill

Foreshadowing: Pipelining

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

- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - Minimalist approach to an ISA: simple insns only
    + Low "cycles/insn" and "seconds/cycle"
  - Higher "insn/program", but hopefully not as much
    - Rely on compiler optimizations

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - A more heavyweight approach: both simple and complex insns
    + Low "insns/program"
  - Higher "cycles/insn" and "seconds/cycle"
    - We have the technology to get around this problem

- More on this later, but first ISA basics

### ISA Code Example

- **Array Sum Loop: LC4**

```assembly
.DATA
array .BLKW #100
sum .FILL #0

.CODE

.array_sum

CONST R5, #0
LEA R1, array
LEA R2, sum
L1
LDR R3, R1, #0
LDR R4, R2, #0
ADD R4, R3, R4
STR R4, R2, #0
ADD R1, R1, #1
ADD R5, R5, #1
CMP R5, #100
BRn L1
```

- **Array Sum Loop: LC4 ➔ MIPS**

```assembly
.MIPS (right) similar to LC4

ываулкюва 
array .BLKW #100 
sum .FILL #0 

.CODE 

.array_sum 

CONST R5, #0 
LEA R1, array 
LEA R2, sum 
L1 
LDR R3, R1, #0 
LDR R4, R2, #0 
ADD R4, R3, R4 
STR R4, R2, #0 
ADD R1, R1, #1 
ADD R5, R5, #1 
CMP R5, #100 
BRn L1 
```

Syntactic differences: register names begin with $
immediates are un-prefixed

Displacement addressing syntax: displacement(reg)
Aspects of ISAs

Length and Format

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

- Encoding
  - A few simple encodings simplify decoder
  - x86 decoder one nasty piece of logic
LC4/MIPS/x86 Length and Encoding

- LC4: 2-byte insns, 3 formats
  - 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) J(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, many formats

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-fp” insns, e.g., SSE/SSE2
  - For multimedia, more about these later

- Other, infrequently supported, data types
  - 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

LC4/MIPS/x86 Operations and Datatypes

- LC4
  - 16-bit integer: add, and, not, sub, mul, div, or, xor, shifts
  - No floating-point

- MIPS
  - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
  - 32(64) bit floating-point: add, sub, mul, div

- x86
  - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
  - 80-bit floating-point: add, sub, mul, div, sqrt
  - 64-bit packed integer (MMX): padd, pmul...
  - 64(128)-bit packed floating-point (SSE/2): padd, pmul...
How Many Registers?

- Registers faster than memory, have as many as possible?
  - **No**

- One reason registers are faster: there are few
  - simpler (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: more registers: 8 (x86) → 32 (MIPS) → 128 (IA64)
  - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

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 virtual address (VA) 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)

LC4/MIPS/x86 Registers

- **LC4**
  - 8 16-bit integer registers
  - No floating-point registers

- **MIPS**
  - 32 32-bit integer registers ($0$ hardwired to 0)
  - 32 32-bit floating-point registers (or 16 64-bit registers)

- **x86**
  - 8 8/16/32-bit integer registers (not general purpose)
  - No floating-point registers!

- **64-bit x86**
  - 16 64-bit integer registers
  - 16 128-bit floating-point registers

LC4/MIPS/x86 Memory Size

- **LC4**
  - 16-bit $(2^{16}$ 16-bit words) x 2 (split data and instruction memory)

- **MIPS**
  - 32-bit
  - 64-bit

- **x86**
  - 8086: 16-bit
  - 80286: 24-bit
  - 80386: 32-bit
  - AMD Opteron/Athlon64, Intel's newer Pentium4, Core 2: 64-bit
How Are Memory Locations Specified?

- Registers are specified **directly**
  - Register names are short, can be encoded in instructions
  - Some instructions implicitly read/write certain registers

- How are addresses specified?
  - Addresses are as big or bigger than insns
  - **Addressing mode**: how are insn bits converted to addresses?
  - Think about: what high-level idiom addressing mode captures

Memory Addressing

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

Examples

- **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?

LC4/MIPS/x86 Addressing Modes

- **LC4**
  - **Displacement**: R1+offset (6-bit)

- **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)
  - **Displacement**: R1+offset (8/16/32-bit)
  - **Indexed**: R1+R2
  - **Scaled**: R1 + (R2*Scale) + offset (8/16/32-bit)  Scale = 1, 2, 4, 8
  - **PC-relative**: PC + offset (32-bit)

x86 Addressing Modes

```
.LFE2
.globl array_sum
array_sum:
    movl $0, -4(%rbp)
.L1:
    movl -4(%rbp), %eax
    movl array(%eax,4), %edx
    movl sum(%rip), %eax
    addl %edx, %eax
    movl %eax, sum(%rip)
    addl $1, -4(%rbp)
    cmpl $99, -4(%rbp)
    jle .L1
```

- **Displacement**
  - Used for sequential array access

- **Scaled**: address = array + ([%eax] * 4)

- **PC-relative**
  - Note: movl is load, store, reg-move, load-const
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
      - lw1 @XXXX10; lwz @XXXXX1

- **Endianness**: 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]
  - **4+**: useful only in special situations

- Why have fewer?
  - Primarily code density (size of each instruction in program binary)
  - Examples show register operands...
    - But operands can be memory addresses, or mixed register/memory
  - ISAs with register-only ALU insns are "load-store"

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.

LC4/MIPS/x86 Operand Models

- **LC4**
  - Integer: 8 general-purpose registers, load-store
  - Floating-point: none
- **MIPS**
  - Integer/floating-point: 32 general-purpose registers, load-store
- **x86**
  - Integer (8 registers) reg-reg, reg-mem, mem-reg, but no mem-mem
  - Floating point: stack (why x86 floating-point lagged for years)
    - SSE introduced 16 general purpose floating-point registers
  - 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
x86 Operand Model: Accumulators

- RISCs use general-purpose registers
- x86 uses explicit accumulators
  - Both register and memory
  - Distinguished by addressing mode

```
.LFE2
.comm array,400,32
.comm sum,4,4
.globl array_sum
array_sum:
  movl $0, -4(%rbp)

.Register accumulator: %eax = %eax + %edx

.L1:
  movl -4(%rbp), %eax
  movl array(%eax), %edx
  movl sum(%ebp), %eax
  addl %edx, %eax
  movl %eax, sum(%ebp)
  addl $1, -4(%rbp)
  cmpl $99, -4(%rbp)
  jle .L1
```

Operand Model & Compiler Optimizations

- How do operand model & addressing mode affect compiler?
  - Reduce insn count, reduce load/store count (important), schedule

- What features enable or limit these?
  + (Many) general-purpose registers let you reduce stack accesses
    - Implicit operands clobber values
      - addl %edx, %eax destroys initial value in %eax
      - Requires additional insns to preserve if needed
  - Implicit operands also restrict scheduling
    - Classic example, condition code
  - Upshot: you want a general-purpose register load-store ISA (MIPS)

Control Transfers

- Default next-PC is PC + sizeof(current insn)
- Branches and jumps can change that
  - Otherwise dynamic program == static program

**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
- 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`
  - Fewer instructions
  - Two ALUs: one for condition, one for target address
  - Less room for target in insn
  - Extra latency

- **Implicit condition codes (x86, LC4)**
  - `cmp R1,10 // sets "negative" CC`
  - More room for target in insn, condition codes often set "for free"
  - Branch insn simple and fast
  - Implicit dependence is tricky

- **Condition registers, separate branch insns (MIPS)**
  - `set-less-than R2,R1,10`
  - `branch-not-equal-zero R2,target`
  ± A compromise

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)

LC4, MIPS, x86 Control Transfers

- **LC4**
  - 9-bit offset PC-relative branches (condition codes)
  - 11-bit offset PC-relative jumps
  - 11-bit absolute 16-byte aligned calls

- **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 condition registers: `slt, sltu, slti, sltiu`, etc.
  - 26-bit target absolute jumps and calls

- **x86**
  - 8-bit offset PC-relative branches
    - **Uses condition codes**
    - Explicit compare instructions (and others) to set condition codes

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 (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 Context

- Pre 1980
  - Bad compilers (so assembly written by hand)
  - Complex, high-level ISAs (easier to write assembly)
  - Slow multi-chip micro-programmed implementations
    - Vicious feedback loop

- Around 1982
  - Moore’s Law makes single-chip microprocessor possible...
    - ...but only for small, simple ISAs
  - Performance advantage of this “integration” was compelling
  - Compilers had to get involved in a big way

**RISC manifesto**: create ISAs that...
- Simplify single-chip implementation
- Facilitate optimizing compilation

Role of Compilers

- Who is generating assembly code?

- Humans like high-level “CISC” ISAs (close to prog. langs)
  + 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 prog. lang. directly

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

- No equivalent “CISC manifesto”
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 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, load-store
  - 64-bit virtual address space
  - Few addressing modes
  - Why so many basically similar ISAs? Everyone wanted their own

The Debate

- **RISC argument**
  - CISC is fundamentally handicapped
  - For a given technology, RISC implementation will be better (faster)
    - Current technology enables single-chip RISC
    - When it enables single-chip CISC, RISC will be pipelined
    - When it enables pipelined CISC, RISC will have caches
    - When it enables CISC with caches, RISC will have next thing...
- **CISC rebuttal**
  - CISC flaws not fundamental, can be fixed with more transistors
  - Moore’s Law will narrow the RISC/CISC gap (true)
    - Good pipeline: RISC = 100K transistors, CISC = 300K
    - By 1995: 2M+ transistors had evened playing field
  - Software costs dominate, compatibility is paramount

Compatibility

- 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
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 \( \mu \text{ops} \) in hardware
  
  ```
  \begin{verbatim}
  push $eax
  becomes (we think, uops are proprietary)
  store $eax \{ \$esp-4 \}
  addi $esp, $esp, -4
  \end{verbatim}
  
  + Processor maintains \textbf{x86 ISA externally for compatibility}
  + But executes \textbf{RISC \( \mu \text{ISA internally for implementability} \)}
  - 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

- 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 \( \sim 1.6 \ \mu \text{ops} / \text{x86 insn} \)
  - Logic for common insns that translate into 1–4 \( \mu \text{ops} \)
  - Table for rare insns that translate into 5+ \( \mu \text{ops} \)
- x86-64: average \( \sim 1.1 \ \mu \text{ops} / \text{x86 insn} \)
  - More registers (can pass parameters too), fewer \texttt{pushes/pops}
  - Core2: logic for 1–2 \( \mu \text{ops} \), table for 3+ \( \mu \text{ops} \)?
- More recent: “macro-op fusion” and “micro-op fusion”
  - Intel’s recent processors fuse certain instruction pairs
  - Macro-op fusion: fuses “compare” and “branch” instructions
  - Micro-op fusion: fuses load/add pairs, fuses store “address” & “data”

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - \textbf{Binary-translation}: transform static image, run native
  - \textbf{Emulation}: unmodified image, interpret each dynamic insn
  - Typically optimized with just-in-time (JIT) compilation
  - Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
  - Performance overheads reasonable (many recent advances)
  - Transmeta’s “code morphing” translation layer
  - Performed with a software layer below OS
  - Looks like x86 to the OS & applications, different ISA underneath
- \textbf{Virtual ISAs}: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime)

Ultimate Compatibility Trick

- Support old ISA by...
  - ...having a simple processor for that ISA somewhere in the system
  - How first Itanium supported x86 code
    - x86 processor (comparable to Pentium) on chip
  - How PlayStation2 supported PlayStation games
    - Used PlayStation processor for I/O chip & \texttt{emulation}
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...
  - Moore’s law has helped Intel in a big way
    - Most engineering problems can be solved with more transistors

- 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...
  - Moore’s law has helped Intel in a big way
    - Most engineering problems can be solved with more transistors

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?

Current Winner (Volume): 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
  - 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.

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