CIS 501
Computer Architecture

Unit 1: Instruction Set Architecture

Slides developed by Milo Martin & Amir Roth at the 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 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

- Baer’s “MA:FSPTCM”
  - Chapter 1.1-1.4 of MA:FSPTCM
  - Mostly Section 1.1.1 for this lecture (that’s it!)
  - Lots more in these lecture notes
- Paper
  - *The Evolution of RISC Technology at IBM* by John Cocke

Big Picture (and Review)
Program Compilation

- **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?

---

Example Assembly Language & ISA

- **MIPS**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 64 registers (32 integer, 32 FP)
  - ~100 different insns
  - Full OS support

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

---

Assembly & Machine Language

- **Assembly language**
  - Human-readable representation

- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in “hex”)

- **Assembler**
  - Translates assembly to machine

Example is in “LC4” a toy instruction set architecture, or ISA

---

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.)
  - Writes its outputs (registers & memory)
  - Next insn (adjust the program counter)

- **Program is just “data in memory”**
  - Makes computers programmable (“universal”)
Reasoning about 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)

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

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

Foreshadowing: ISAs & Performance

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

- A good ISA helps balances three three aspects

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

- We’ll revisit this when we talk about “RISC” vs “CISC”

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
  - Can break this constraint physically (pipelining)
  - But must maintain illusion to preserve programmer sanity

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... (more talk a bit about these later on)
  - Reduce branches and jumps
  - Reduce cache misses
  - Reduce dependences between nearby insns
    - An ISA can make this difficult

- 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
x86 Assembly Instruction Example 1

```c
int func(int x, int y)
{
    return (x+10) * y;
}
```

```
.register names begin with `%`
.immediates begin with $

 Inputs are passed to function in registers:
x is in %edi, y is in %esi

 Two operand insns:
(right-most is typically source & destination)

 Function output is in %eax

 "L" insn suffix and "%e..." reg. prefix mean "32-bit value"
```

x86 Assembly Instruction Example 2

```
int f(int x);
int g(int x);

int func(int x, int y)
{
    int val1;
    if (x > 10) {
        val1 = f(y);
    } else {
        val1 = g(y);
    }
    return val1 * 100;
}
```

```
"cmp" compares to values, sets the "flags"
"jg" looks at flags, and jumps if greater

 "q" insn suffix and "%r..." reg. prefix mean "64-bit value"
```

x86 Assembly Instruction Example 3

```
struct list_t {
    int value;
    list_t* next;
};

int func(list_t* l)
{
    int counter = 0;
    while (l != NULL) {
        counter++;
        l = l->next;
    }
    return counter;
}
```

```
"mov" with () accesses memory
"test" sets flags to test for NULL

 "movq" with () accesses memory
 "testq" sets flags to test for NULL

 "q" insn suffix and "%r..." reg. prefix mean "64-bit value"
```

Assembly Code Examples
Array Sum Loop: x86

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

x86 Operand Model

- x86 uses explicit accumulators
- Both register and memory
- Distinguished by addressing mode

Array Sum Loop: x86 → Optimized x86

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

Array Sum Loop: MIPS, Unoptimized

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

Array Sum Loop: MIPS, Unoptimized

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

Examples Instruction Encodings

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

<table>
<thead>
<tr>
<th>R-type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rl(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>Op(6)</td>
<td>Rs(5)</td>
<td>Rl(5)</td>
<td></td>
<td>Immed(16)</td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>Op(6)</td>
<td></td>
<td></td>
<td>Target(26)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **x86**
  - Variable length encoding (1 to 16 bytes)

<table>
<thead>
<tr>
<th>Prefix*(1-4)</th>
<th>Op</th>
<th>OpExt*</th>
<th>ModRM*</th>
<th>SIB*</th>
<th>Disp*(1-4)</th>
<th>Imm*(1-4)</th>
</tr>
</thead>
</table>
Where Does Data Live?

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

- **Memory**
  - Fundamental storage space
  - "longer term memory"
  - Location to read or write calculated from registers

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

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 and other fixed-point arithmetic

How Many Explicit Register Operands

- **Operand model**: how many explicit operands
  - 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
    - Fused multiply & accumulate instruction

- **Why have fewer?**
  - Primarily code density (size of each instruction in program binary)

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: 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, 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?

Addressing Modes Examples

• **MIPS**
  • **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

Example: x86 Addressing Modes

```
.LFE2
  .comm array,400,32
  .comm sum,4,4

.globl array_sum
array_sum:
  movl $0, -4($rbp)

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

Note: “mov” can be load, store, or reg-to-reg move
### 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)

### 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
      lw $t0, @XXXX10; lw $t0, @XXXXX10

- **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

### 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.

### x86 Operand Model: Accumulators

- x86 uses explicit accumulators
  - Both register and memory
  - Distinguished by addressing mode

---

CIS 501 (Martin): Instruction Set Architectures 37

CIS 501 (Martin): Instruction Set Architectures 38

CIS 501 (Martin): Instruction Set Architectures 39

CIS 501 (Martin): Instruction Set Architectures 40
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
  - 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-bit mode”** extends number of registers
  - Integer: 16 64-bit registers
  - Floating point: 16 128-bit registers

Operand Model & Compiler Optimizations

- How do operand model & addressing mode affect compiler?
- Again, what does a compiler try to do?
  - 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 (flags)
  - Ideally, 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**
  - \texttt{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 or “flags” (x86, LC4)**
  - \texttt{cmp R1,10 // sets “negative” flag}
  - \texttt{branch-neg target}
    - 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)**
  - \texttt{set-less-than R2,R1,10}
  - \texttt{branch-not-equal-zero R2,target}
  - \pm A compromise

MIPS and x86 Control Transfers

- **MIPS**
  - 16-bit offset PC-relative conditional branches
    - **Uses register for condition**
    - Compare two regs: \texttt{beq, bne}
    - Compare reg to 0: \texttt{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)
    - Okay to take two insns to do remaining branches
      - It’s the uncommon case
    - Explicit “set condition into registers”: \texttt{slt, sltu, slti, sltiu}, etc.
- **x86**
  - 8-bit offset PC-relative branches
    - **Uses condition codes (“flags”)**
    - Explicit compare instructions (and others) to set condition codes

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)

ISA Design Goals
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**
  - 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 says: “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.

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 (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 insns
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

• 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

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

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 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 insn
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

The RISC Design 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
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.

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 microops 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 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)
  - Different microops for different designs
    - Not part of the ISA specification, not publically disclosed

Potential Micro-op Scheme (1 of 2)

- 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 operation 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)
  - What about address generation?
    - Simple address generation is generally part of single micro-op
      - Sometime store addresses are calculated separately
    - More complicated (scaled addressing) might be separate micro-op
Potential Micro-op Scheme (2 of 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.

Again, this is just a basic idea (and what we will use in our assignments), the exact micro-ops are specific to each chip.

More About Micro-ops

- Two forms of μops “cracking”
  - Hard-coded logic: fast, but complex (for insn in few μops)
  - Table: slow, but “off to the side”, doesn’t complicate rest of machine
    - Handles the really complicated instructions
- x86 code is becoming more “RISC-like”
  - In 32-bit to 64-bit transition, x86 made two key changes:
    - Double number of registers, better function calling conventions
    - More registers (can pass parameters too), fewer pushes/pops
    - Result? Fewer complicated instructions
    - Moved from ~1.6 μops / x86 insn to ~1.1 μops / x86 insn
- 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”

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 & emulate

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: FX132 (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)
    - NVIDIA’s “PTX”
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

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 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