Instruction Sets

- what is an instruction set?
- what is a good instruction set?
- the forces that shape instruction sets
- aspects of instruction sets
- instruction set examples
- RISC vs. CISC

Instruction Sets

“Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine”

–IBM introducing 360 (1964)

an instruction set specifies a processor’s functionality
- what operations it supports
- what storage mechanisms it has & how they are accessed
- how the programmer/compiler communicates programs to processor

instruction set architecture (ISA): “architecture” part of this course
- the rest is micro-architecture

Readings

Hennessy and Patterson
- chapter 2

Readings in Computer Architecture
- Wulf: “Compilers and Computer Architecture”

Recent Research Paper
- Transmeta Crusoe

What Makes a Good Instruction Set?

- implementability
  - supports a (performance/cost) range of implementations
    - implies support for high performance implementations

- programmability
  - easy to express programs (for human and/or compiler)

- backward/forward compatibility
  - implementability & programmability across generations
    - e.g., x86 generations: 8086, 286, 386, 486, Pentium, Pentium II, Pentium III, Pentium 4...

- think about these issues as we discuss aspects of ISAs
Implementability

want to implement across a spectrum of performance/cost
  • a family of implementations, each for different purposes

low performance implementation
  • easy, trap to software to emulate complex instructions

high performance implementation
  • more difficult
  • components: pipelining, parallelism, dynamic scheduling?
    • avoid artificial sequential dependences
    • deterministic execution latencies simplify scheduling
    • avoid instructions with multiple long latency components
    • avoid not-easily-interruptable instructions

Programmability

a history of programmability
  • pre-1975: most code was hand-assembled
  • 1975–1985: most code was compiled
    • but people thought that hand-assembled code was superior
  • 1985–: most code was compiled
    • and compiled code was at least as good as hand-assembly

over time, a big shift in what “programmability” means

pre-1975: Human Programmability

focus: instruction sets that were easy for humans to program
  • ISA semantically close to high-level language (HLL)
    • closing the “semantic gap”
  • semantically heavy (CISC-like) instructions
    • automatic saves/restores on procedure calls
    • e.g., the VAX had instructions for polynomial evaluation
  • people thought computers would someday execute HLL directly
    • never materialized
  • one problem with this approach: multiple HLLs
    • “semantic clash”: not exactly the semantics you want

1975–: Compiler Programmability

focus: instruction sets that are easy for compilers to compile to
  • primitive instructions from which solutions are synthesized
    • Wulf: provide primitives (not solutions)
    • hard for compiler to tell if complex instruction fits situation
  • regularity: do things the same way, consistently
    • “principle of least astonishment” (true even for hand-assembly)
  • orthogonality, composability
    • all combinations of operation, data type, addressing mode possible
    • e.g., ADD and SUB should have same addressing modes
  • few modes/obvious choices
    • compilers do complicated case analysis, don’t add more cases
Today’s Semantic Gap

popular argument: today’s ISAs are targeted to one HLL, and it just so happens that this HLL (C) is very low-level (assembly++)

• would ISAs be different if Java was dominant?
  • more object oriented?
  • support for garbage collection (GC)?
  • support for bounds-checking?
  • security support?

Compatibilities: Forward/Backward

basic tenet: make sure all written software works

• business reality: software cost greater than hardware cost
  • Intel was first company to realize this

backward compatibility: must run old software on new machine

• temptation: use ISA gadget for 5% performance gain
  • but must keep supporting gadget even if gain disappears!
  • examples: register windows, delayed branches

forward compatibility: allow for future enhancements

• reserve trap hooks to emulate future ISA extensions

Evolution of Instruction Sets

instruction sets evolve

• implementability driven by technology
  • microcode, VLSI, pipelining, superscalar
  • e.g., tough to pipeline VAX’s polynomial evaluation instructions

• programmability driven by (compiler) technology
  • hand assembly → compilers → register allocation (Ld/St ISA ok)

• instruction set features go from good to bad to good
  • just like microarchitecture ideas

lessons to be learned

• many non-technical (i.e., business) issues influence ISAs
• best solutions don’t always win (e.g., Intel x86)

Instruction Set Aspects

1) format
  • length, encoding

2) operations
  • operations, data types, number & kind of operands

3) storage
  • internal: accumulator, stack, general-purpose register
  • memory: address size, addressing modes, alignments

4) control
  • branch conditions, special support for procedures, predication
Aspect #1: Instruction Format

fixed length (most common: 32-bits)
  + easy for pipelining and for multiple issue (superscalar)
    • don’t have to decode current instruction to find next instruction
      – not compact (4-bytes for NOP?)

variable length
  + more compact
    – hard (but do-able) to superscalarize/pipeline

recent compromise: 2 lengths (32-bit + another length)
  • MIPS16, ARM Thumb: add 16-bit subset (compression)
  • TM Crusoe: adds 64-bit long-immediate instructions

Aspect #2: Operations

• arithmetic and logical: add, mult, and, xor
• data transfer: move, load, store
• control: conditional branch, jump, call, return
• system: system call, traps
• floating point: add, mul, div, sqrt
• decimal: addd, convert (not common today)
• string: move, compare (also not common today)
• multimedia: e.g., Intel MMX/SSE and Sun VIS
• vector: arithmetic/data transfer, but on vectors of data

Data Sizes and Types

• fixed point (integer) data
  • 8-bit (byte), 16-bit (half), 32-bit (word), 64-bit (doubleword)
• floating point data
  • 32/64 bit (IEEE754 single/double precision), 80-bit (Intel proprietary)
• address size (aka “machine size”)
  • e.g., 32-bit machine means addresses are 32-bits
  • key is virtual memory size: 32-bits ⇒ 4GB (not enough anymore)
  • especially since 1 bit is often used to distinguish I/O addresses
  • famous lesson: one of the few big mistakes in an architecture is not enabling a large enough address space

Fixed Point Operation Types

types: s/w (property of data) vs. h/w (property of operation)
• signed (−2^{n−1} to 2^{n−1}−1) vs. unsigned (0 to 2^{n−1}−1)
• packed (multimedia short vector)
  • treat 64-bit as 8x8, 4x16, or 2x32
  • e.g.: addb, addh (MMX)
    17  87 100 ...
    + 17 13 200 ...
    ___ ___ ___ ___
    34 100 255 (saturating or 44 with wraparound)
• MMX example: 16-element dot product: \( \sum a_i b_i \)
• plain: 200 instructions/76 cycles → MMX: 16/12 (6X perf. benefit)
• saturating (no wraparound on overflow)
• useful in RGBA calculations (for pixels)
Aspect #3: Internal Storage Model

choices
- stack
- accumulator
- memory-memory
- register-memory
- register-register (also called "load/store")

running example:
```
add C, A, B (C := A + B)
```

Storage Model: Stack

```
push A S[++TOS] = M[A];
push B S[++TOS] = M[B];
add T1=S[TOS--]; T2=S[TOS--]; S[++TOS]=T1+T2;
pop C M[C] = S[TOS--];
```

- operands implicitly on top-of-stack (TOS)
- ALU operations have zero explicit operands
  + code density (top of stack implicit)
  - memory, pipelining bottlenecks (why?)
- mostly 1960’s & 70’s
  - x86 uses stack model for FP (bad backward compatibility problem)
  - JAVA bytecodes also use stack model

Storage Model: Accumulator

```
load A accum = M[A];
add B accum += M[B];
store C M[C] = accum;
```

- accum is implicit destination/source in all instructions
- ALU operations have one operand
  + less hardware, better code density (accumulator implicit)
  - memory bottleneck
- mostly pre-1960’s
  - examples: UNIVAC, CRAY
  - x86 (IA32) uses extended accumulator for integer code

Storage Model: Memory-Memory

```
```

- no registers
  + best code density (most compact)
  - large variations in instruction lengths
  - large variations in work per-instruction
  - memory bottleneck
- no current machines support memory-memory
  - VAX did

• ISCA '02 paper on register-accumulator architecture
Storage Model: Memory-Register

- load R1, A \( R1 = M[A]; \)
- add R1, B \( R1 += M[B]; \)
- store C, R1 \( M[C] = R1; \)

• like an explicit (extended) accumulator
  + can have several accumulators at a time
• good code density, easy to decode instructions
  – asymmetric operands, asymmetric work per instruction
• 70's and early 80's
  - IBM 360/370
  - Intel x86, Motorola 68K

Storage-Model: Register-Register (Ld/St)

- load R1, A \( R1 = M[A]; \)
- load R2, B \( R2 = M[B]; \)
- add R3, R1, R2 \( R3 = R1 + R2; \)
- store C, R3 \( M[C] = R3; \)

• load/store architecture: ALU operations on registers only
  – poor code density
• easy decoding, operand symmetry
• deterministic length ALU operations
• key: fast decoding helps pipelining and superscalar
• 1960’s and onwards
  - RISC machines: Alpha, MIPS, PowerPC (but also Cray)

Registers vs. Memory

- registers (as compared to memory)
  + faster (direct access, smaller, no tags)
  + deterministic scheduling (i.e., fixed latency, no misses)
  + can replicate for more bandwidth
  + short identifier
    – must save/restore on procedure calls, context switches
    – fixed size
      • strings, structures (i.e., bigger than 64 bits) must live in memory
  + can’t take address of a register
    • pointed-to variables must live in memory

How Many Registers?

- as the number of registers increases ...
  + can hold more operands for longer periods
    • shorter average operand access time, lower memory traffic
    – longer specifiers (longer instructions?)
    – slower access to register operands (bigger is slower)
    – slower procedure calls/context-switch (more save/restore)
  trend is for more registers
  - x86: 8 → SPARC/MIPS/Alpha/PPC: 32 → Itanium: 128
  • why?
Memory Alignment

“natural boundaries” ⇒ (address modulo data size) == 0
  • e.g. word (4 bytes): @x00 ⇒ aligned, @x11 ⇒ unaligned

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>byte of data</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>half word of data</strong></td>
<td></td>
</tr>
</tbody>
</table>

Both data units are aligned

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>half word of data</strong></td>
<td></td>
</tr>
</tbody>
</table>

Data unit is NOT aligned!

Memory Alignment Solutions

alignment restrictions: kinds of alignments architecture supports
  • no restrictions (all in hardware)
    - hardware detects, makes 2 references (what if 2nd one faults?)
    - expensive logic, slows down all references (why?)
  • restricted alignment (software guarantee w/ hardware trap)
    - misaligned access traps, performed in s/w by handler
  • middle ground: multiple instructions for misaligned data
    • e.g., MIPS (lw/lw), Alpha (ldq_u)
    • compiler generates for known cases, h/w traps for unknown cases

Operand Addressing Modes

• immediate: #n (immediate values)
• register: Ri (register values)
• displacement: M[Ri + #n] (stack, structure fields)
• register indirect: M[Ri] (loaded/computed addresses)
• memory absolute: M[#n] (globals)
• indexed: M[Ri + Rj] (arrays of scalars)
• memory indirect: M[M[Ri]] (in-memory pointers)
• scaled: M[Ri + Rj * d + #n] (arrays of structures, x86)
• update/auto-increment/decrement: M[Ri = Ri + #n]

1–4 account for 93% of all VAX operands [Clark+Emer]

RISC machines typically implement 1–3
  • i.e., load/store with only register displacement
    • load: Rj = M[Ri + #n], store: M[Ri + #n] = Rj
  • synthesize all other modes
    • e.g., memory indirect: Rj = M[M[Ri]] ⇒ Rk = M[Rj]; Rj = M[Rk]
  • remember: provide primitives, not solutions
Aspect #4: Control Instructions

aspects
• 1. taken or not?
• 2. where is the target?
• 3. link return address?
• 4. save or restore state?

instructions that change the PC
• (conditional) branches [1, 2]
• (unconditional) jumps [2]
• function calls [2,3,4], function returns [2,4]
• system calls [2,3,4], system returns [2,4]

(1) Taken or Not?

ISA options for specifying if branch is taken or not
• “compare and branch” instruction
  • beq Ri, Rj, target    // if Ri=Rj, then goto target
    + single instruction branch
    – requires ALU usage in branch pipeline, restricts scheduling
  • separate “compare” and “branch” instructions
    • slt Rk, Ri, Rj    // if Ri < Rj, then Rk=1 (else Rk=0)
    • beqz Rk, target   // if Rk==0, then goto target
    – uses up a register, separates condition from branch logic
    + more scheduling opportunities, can maybe reuse comparison
  • condition codes: Zero, Negative, oVerflow, Carry
    + set “for free” by ALU operations
    – extra state to save/restore, scheduling problems

(2) Where is the Target?

ISA options for specifying the target of the control instruction
• PC-relative: branches/jumps within function
  • beqz Rk, #25    // if Rk==0, then goto PC+25
    + position independent, computable early. #bits: <4 (47%), <8 (94%)
    – target must be known statically, can’t jump far
  • absolute: function calls, long jumps within functions
    • jump #8675309
    + can jump farther
    – more bits to specify
• register: indirect calls (DLLs, virtual fns), returns, switch
  • jal Rk   // jump to Mem[Rk]  (and link the return address)
    + short specifier, can jump anywhere, dynamic target ok (return)
    – extra instruction (hidden!) - requires load to get value into Rk
  – branch and target separated in pipeline (an issue we’ll see soon)
• vectored trap: system calls
  • syscall calltype    // invoke system to handle calltype
    + protection
    – surprises are implementation headache
(3) Link Return Address?

ISA options for linking the return address (after a call)
- implicit register: many recent architectures use this
  + fast, simple
  - s/w save register before next call (pain: surprise trap)
- explicit register
  + may avoid saving register
  - register must be specified
- processor stack
  + recursion is direct
  - complex instructions

(4) Saving/Restoring State

Before a function call or system call, the caller and the callee must cooperate in order to not overwrite each other’s state
- function calls: save/restore registers
- system calls: save/restore registers, flags, PC, PSW, etc.

Option #1: software save/restore
- calling convention divides work
  - caller saves registers in use
  - callee saves registers it (or nested callees) will use

Option #2: explicit hardware save/restore
- VAX CALLS instruction, IBM STM

Option #3: implicit hardware (e.g., SPARC register windows)
- 32 registers: 8 in, 8 out, 8 local, 8 global
- call: out in (pass parameter), local/out “fresh”, global unchanged
- on return: opposite, 8 output of caller restored
- no saving/restoring for shallow call graphs
- makes register renaming (needed for out of order execution) hard

Alternative to Control: Predication

if \((a > 0)\) \(c = b*a\);

\[
\text{instr#0: } \text{blez } r1, #2 \\
\text{instr#1: } \text{mul } r3, r2, r1
\]

- problem? instr#0 is a branch
  - expensive if mis-predicted (later)
- predication: converts control-flow to data-flow
  - branch mis-prediction avoided
  - but data-dependences complicated
- two ways: conditional moves (left), or general predication (right)

\[
#0: \text{mul } r4, r2, r1 \\
#0: \text{sgtrp p1, r1} \\
#1: \text{cmovgt } r3, r4, r1 \\
#1: \text{divp r3, r4, r1, p1}
\]
Four Instruction Set Examples

DEC/Compaq(/Intel)
- (1) VAX (CISC) → (2) Alpha (RISC)

Intel
- (3) IA32 (CISC) → (4) IA64 (RISC VLIW)

DEC/Compaq VAX
1977: ultimate CISC
- virtual memory (Virtual Address eXtension to PDP-11)
- 32-bit ISA, variable length instructions (1 to 321 bytes!)
- 16 GPRs (r15 PC, r14 SP), condition codes (CCs)
- data types: 8, 16, 32, 64, 128, decimals, strings
- orthogonal, memory-memory, all operand modes
- hundreds of instructions: crc, insque, polyf
- touted as first MIPS-1 machine
  - oops: 10.6 CPI @ 200ns ⇒ 0.5 MIPS
  - 5 CPI just in decode (4 just to decode 16% of instructions)
  - flagrant violation of Amdahl's law

DEC/Compaq/? Alpha
1990: ultimate RISC
- first 64-bit machine
- 32 64-bit GPRs, 32 64-bit FPRs
  - RISC: first implementation had no support for 8, 16, 32-bit data
  - added later after software vendor protests (BWX)
- displacement addressing only
- privileged subset (PAL) for lightweight OS implementation
- other extensions
  - predication with conditional moves
  - non-binding memory instructions (prefetch hints)
- get familiar with this ISA via SimpleScalar

Intel x86 (IA32)
1974: most commercially successful ISA ever
- variable length instructions (1–16 bytes)
- 8 32-bit “not-so-general purpose” registers
  - partial 16- and 8-bit versions of each register (AX, AH, AL)
  - FP operand stack
- “extended accumulator” (two-operand instructions)
  - based on Intel 8080, which was pure accumulator ISA
  - register-register and register-memory
  - stack manipulation instructions (but no internal integer stack)
- scaled addressing: base + (index * scale) + displacement
- 2-level memory (segments)
- “difficult to explain and impossible to love” —an x86 designer
Intel Itanium (IA64)

4GB virtual memory in IA32 is too little, so what are options?
• segmentation (already present in IA32)
• 64-bit extensions (solution for 16- and 32-bits)
  • AMD Opteron (Intel hedge = Yamhill?)
2000: neither, create brand new ISA (IA64)
• EPIC: binary compatible (interlocked) VLIW
  • 3-instruction bundles (with explicit info about parallelism)
• support for ILP: predication, software speculation
• implementations
  – Itanium (Merced), under Moore’s curve
  – Itanium2 (McKinley), jury still out

RISC War: RISC vs. CISC

early 80’s: RISC movement challenges “CISC establishment”
• RISC (reduced instruction set computer)
  • Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801
• CISC (complex instruction set computer)
  • VAX, x86
• word “CISC” did not exist before word RISC came along

RISC Manifesto

• single-cycle operation (CISC: many multi-cycle ops)
• hardwired control (CISC: microcode)
• load/store organization (CISC: mem-reg, mem-mem)
• fixed instruction format (CISC: variable format)
• few modes (CISC: many modes)
• reliance on compiler optimization (CISC: hand assembly)
  + load/store ⇒ register allocation (+21% performance)
  + simple instructions ⇒ fine-grain CSE (+10%), scheduling (?)
    (Common Sub-expression Elimination)
• no equivalent “CISC manifesto”

RISC and CISC Arguments

RISC argument [Dave Patterson]
• CISC is fundamentally handicapped
  • for a given technology, RISC implementation will be faster
    • current VLSI technology enables single-chip RISC
    • when technology enables single-chip CISC, RISC will be pipelined
    • when technology enables pipelined CISC, RISC will have caches
    • when technology enables CISC with caches, RISC will have ...

CISC rebuttal [Bob Colwell]
• CISC flaws not fundamental (fixed with more transistors)
  • Moore's Law will narrow the RISC/CISC gap (true)
• software costs will dominate (very true)
RISC/CISC Showdown

VAX 8700 vs. MIPS R3000 [Clark+Bhandarkar]

| bench   | instr ratio | CPI ratio | RISC factor 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>1.6</td>
<td>6.0</td>
<td>3.7</td>
</tr>
<tr>
<td>eqntott</td>
<td>1.1</td>
<td>3.5</td>
<td>3.2</td>
</tr>
<tr>
<td>fpppp</td>
<td>2.9</td>
<td>10.5</td>
<td>3.6</td>
</tr>
<tr>
<td>tomtav</td>
<td>2.9</td>
<td>8.2</td>
<td>2.8</td>
</tr>
</tbody>
</table>

- argues
  - RISCs fundamentally better than CISCs
  - implementation effects and compilers are second order
- unfair because it compares specific implementations
  - VAX advantages: big immediates, not-taken branches
  - MIPS advantages: more registers, FPU, instruction scheduling, TB

The Joke on RISC

most commercially successful ISA is x86 (decidedly CISC)

- also: PentiumPro was first out-of-order microprocessor
  - good RISC pipeline, 100K transistors
  - good CISC pipeline, 300K transistors
  - by 1995: 2M+ transistors evened pipeline playing field
  - rest of transistors used for caches (diminishing returns)
- Intel's other trick?
  - decoder translates CISC into sequences of RISC \( \mu \)ops

\[
\begin{align*}
\text{push EAX} & \downarrow \\
\mu\text{addi ESP, ESP, 4} & \\
\mu\text{store EAX, 0(ESP)} & \\
\end{align*}
\]

- internally (micro-architecture) is actually RISC!

ISA → ISA + \( \mu \)ISA

Intel's trick (decode external ISA into internal ISA) is popular
  + stable external ISA gives compatibility
  + flexible internal ISA gives implementability
    - obsolete features? don’t implement, emulate
  - Intel PentiumII and AMD Athlon: \( \mu \)ops or Rops
    - translation by PLAs (hardware)
  - Intel Pentium4
    - translations are cached
  - Transmeta Crusoe: translates x86 to RISC VLIW
    - translation (code-morphing) by invisible software
    - primary goal is lower power/performance (VLIW)
    - TM3200 and TM5400 \( \mu \)ISAs are slightly different

Summary

- 3 *-ilities: implementability, programmability, compatibility
- design principles
- aspects: format, data types, operand modes/model...
- RISC vs. CISC

let’s discuss readings
  - Wulf: “Compilers and Computer Architecture”
  - Transmeta Crusoe

next up: Pipelining