55:132/22C:160, HPCA Spring 2011

Second Lecture Slide Set

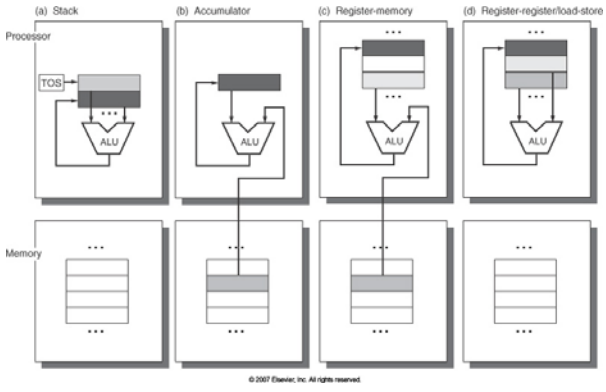
Instruction Set Architecture

Instruction Set Architecture

- ISA, the boundary between software and hardware
  - Specifies the logical machine that is visible to the programmer (or compiler).
  - Also, a functional spec for the processor designers
- What needs to be specified by an ISA
  - Operations
    - what to perform and what to perform next
  - Temporary Operand Storage in the CPU
    - accumulator, stacks, registers
  - Number of operands per instruction
  - Operand location
    - where and how to specify the operands
  - Type and size of operands
  - Instruction-to-Binary Encoding

Basic ISA Classification

- Stack Architecture (zero operand):
  - Operands popped from stack
  - Result pushed on stack
- Accumulator (one operand):
  - Special accumulator register is implicit operand
  - Other operand from memory
- Register-Memory (two operand):
  - One operand from register, other from memory or register
  - Generally, one of the source operands is also the destination
  - A few architectures—e.g. VAX, M68000—have allowed mem. to mem. operations
- Register-Register or Load/Store (three operand):
  - All operands for ALU instructions must be registers
  - General format R_d <= R_s op R_t
  - Separate Load and Store instructions for memory access



Other Important ISA Considerations

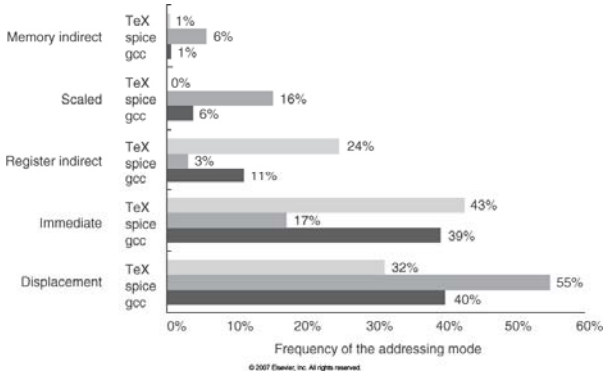
- Number of (architected) registers
- Addressing modes
- Data types/sizes
- Instruction functionality—simple vs. complex
- Branch/jump/subroutine call functionality
- Exception handling
- Instruction format/size/regularity
- Etc.

Addressing Modes

register: R_i
immediate: #n
indexed: $M[R_i + R_j]$
memory indirect: $M[M[R_i]]$
auto-decrement: $M[R_i]$; R_i -= d
scaled: $M[R_i + \#n + R_j * d]$
update: $M[R_i = R_i + \#n]$
displacement $M[R_i + \#n]$
register indirect $M[R_i]$
absolute: M[#n]
auto-increment: $M[R_i]$; R_i += d

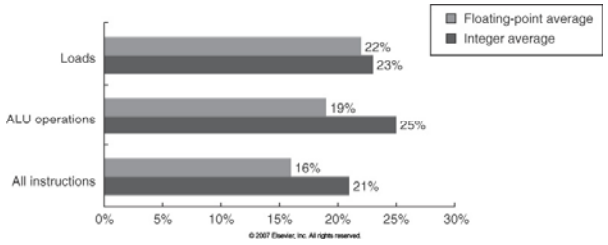
- Modes 1-4 account for 93% of all VAX operands [Clark and Emer]
- Note: For a review of addressing modes, see Figure B.6 in Appendix B of the text

VAX Addressing Mode Usage*



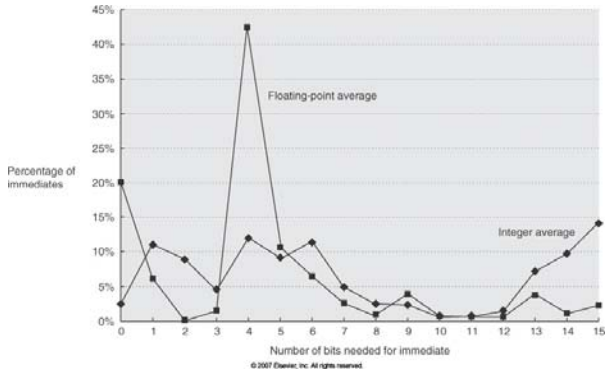
*Note: Register addressing, not shown, accounts for 50% of operand accesses

Percentage of Instructions with Immediate Operand



DEC Alpha Processor, SPEC2000 Benchmarks

Size Distribution of immediate Values



8-bit field would capture 50% of all immediate values and 12-bit field would capture 80%

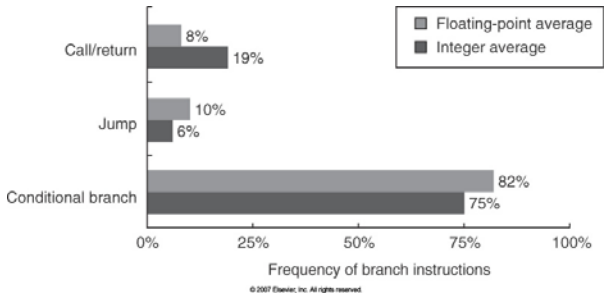
Operations

- arithmetic and logical - and, add …
- data transfer - move, load, store
- control - branch, jump, call
- system - system call, traps
- floating point - add, mul, div, sqrt
- decimal - addd, convert
- string - move, compare
- multimedia? 2D, 3D? e.g., Intel MMX/SSE and Sun VIS

Control Instructions (Branches)

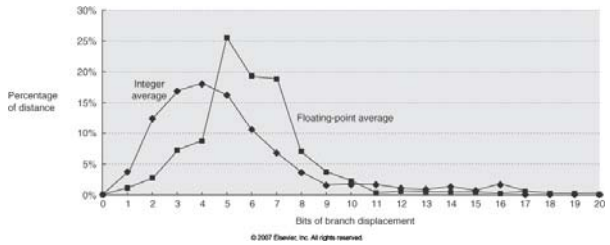
1. Types of Branches
   A. Conditional or Unconditional
   B. Save PC?
   C. How is target computed?
      - Single target (immediate, PC+immediate)
      - Multiple targets (register)
2. Branch Architectures
   A. Condition code or condition registers
   B. Register

Relative Frequency of Control Instructions



DEC Alpha Processor, SPEC2000 Benchmarks

of Bits Needed for Branch Displacement



Save or Restore State

- What state?
  - function calls: registers (CISC)
  - system calls: registers, flags, PC, PSW, etc
- Hardware need not save registers
  - caller can save registers in use
  - callee save registers it will use
- Hardware register save
  - IBM STM, VAX CALLS
  - faster?
- Most recent architectures do no register saving
  - Or do implicit register saving with register windows (SPARC)

Instruction Format (encoding)

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier *n*	Address field *n*

(a) Variable (e.g., Intel 80x86, VAX)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

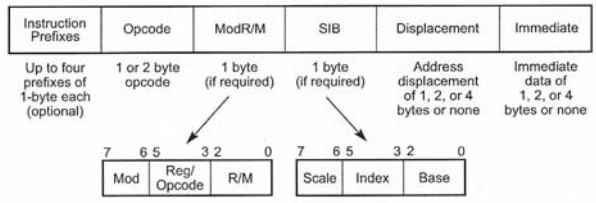
Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

A CISC ISA—x86 (IA-32)

- This ISA was first introduced with the Intel 8086 processor in 1978
- Has evolved, with many additions over the years
- Main characteristics:
  - Reg-mem architecture—ALU instructions can have memory operands
  - Two operand format—one source operand is also destination
  - Eight general purpose registers
  - Seven memory addressing modes
  - More than 500 instructions
  - Instruction set is non-orthogonal
  - Highly variable instruction size and format—instruction size varies from 1 to 17 bytes.

X86 Instruction Format



X86 Addressing Modes

- Absolute
- Register indirect
- Based
- Based indexed
- Based indexed with displacement
- Based with scaled index
- Based with scaled index and displacement

Anatomy of a RISC ISA

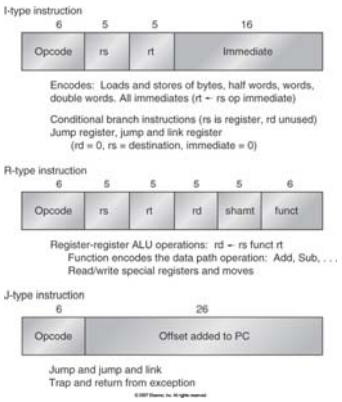
- Operations
  simple ALU op's, data movement, control transfer
- Temporary Operand Storage in the CPU
  Large General Purpose Register (GPR) File
- Load/Store Architecture
  - Three operands per ALU instruction (all registers)
    A ⇐ B op C
- Addressing Modes
  Limited addressing modes---e.g. register indirect addressing
- Type and size of operands
  32/64-bit integers, IEEE floats
- Instruction-to-Binary Encoding
  Fixed width, regular fields

Exceptions: Intel x86, IBM 390 (aka z900)

MIPS ISA

- The MIPS ISA was one of the first RISC instruction sets (1985)
- Similar to ISAs of other RISC processors: Sun SPARC, HP PA-RISC,DEC Alpha
- Main characteristics
  - Load-store architecture
  - Three operand format (R_d <= R_s op R_t)
  - 32 General Purpose Registers (actually 31)
  - Only one addressing mode for memory operands: reg.indirect w. displ.
  - Limited, highly orthogonal instruction set: 52 instructions
  - Simple branch/jump/subroutine call architecture

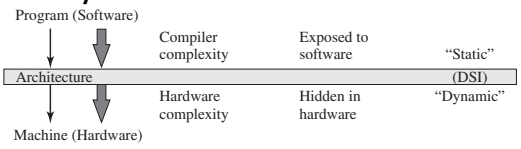
MIPS Instruction Format



The Role of the Compiler

- Phases to manage complexity
  - **Parsing --> intermediate representation**
  - **Procedure inlining**
  - **Loop Optimizations**
  - **Common Sub-Expression**
  - **Jump Optimization**
  - **Constant Propagation**
  - **Register Allocation**
  - **Strength Reduction**
  - **Pipeline Scheduling**
  - **Code Generation --> assembly code**

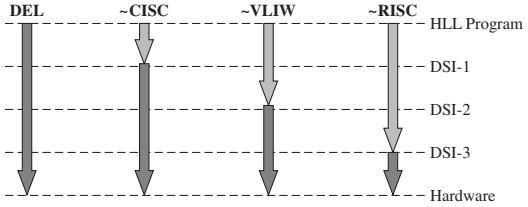
Dynamic-Static Interface*



- Semantic gap between s/w and h/w
- Placement of DSI determines how gap is bridged

*This term comes from Shen and Lipasti

Dynamic-Static Interface



- Low-level DSI exposes more knowledge of hardware through the ISA
  - Places greater burden on compiler/programmer
- Optimized code becomes specific to implementation
  - In fact: happens for higher-level DSI also