# Computer Architecture

# Instruction Set Architecture


Virendra Singh

Associate Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

E-mail: viren@ee.iitb.ac.in


*CS-683: Advanced Computer Architecture*

Lecture 2 (26 July 2013)

CADSL

# What Are the Components of an ISA?

- Sometimes known as *The Programmer's Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i.e., register transfers

# What Are the Components of an ISA?

- The instruction format
  - Size and meaning of fields within the instruction
- The nature of the fetch-execute cycle
  - Things that are done before the operation code is known

# Instruction

- C Statement

  f = (g+h) – (i+j)

- Assembly instructions

  add t0, g, h

  add t1, I, j

  sub f, t0, t1

- Opcode/mnemonic, operand , source/ destination

# Why not Bigger Instructions?

- Why not "f = (g+h) – (i+j)" as one instruction?
- Church's thesis: A very primitive computer can compute anything that a fancy computer can compute – you need only logical functions, read and write to memory, and data dependent decisions
- Therefore, ISA selection is for practical reasons
  – Performance and cost not computability
- Regularity tends to improve both
  – E.g, H/W to handle arbitrary number of operands is complex and slow, and UNNECESSARY

# What Must an Instruction Specify?(I)

Data Flow ←

- Which operation to perform add r0, r1, r3
  - Ans: Op code: add, load, branch, etc.
- Where to find the operands: add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add r0, r1, r3
  - Again CPU register or memory cell

# What Must an Instruction Specify?(II)

- Location of next instruction

  add r0, r1, r3
  br endloop

  – Almost always memory cell pointed to by program counter—PC

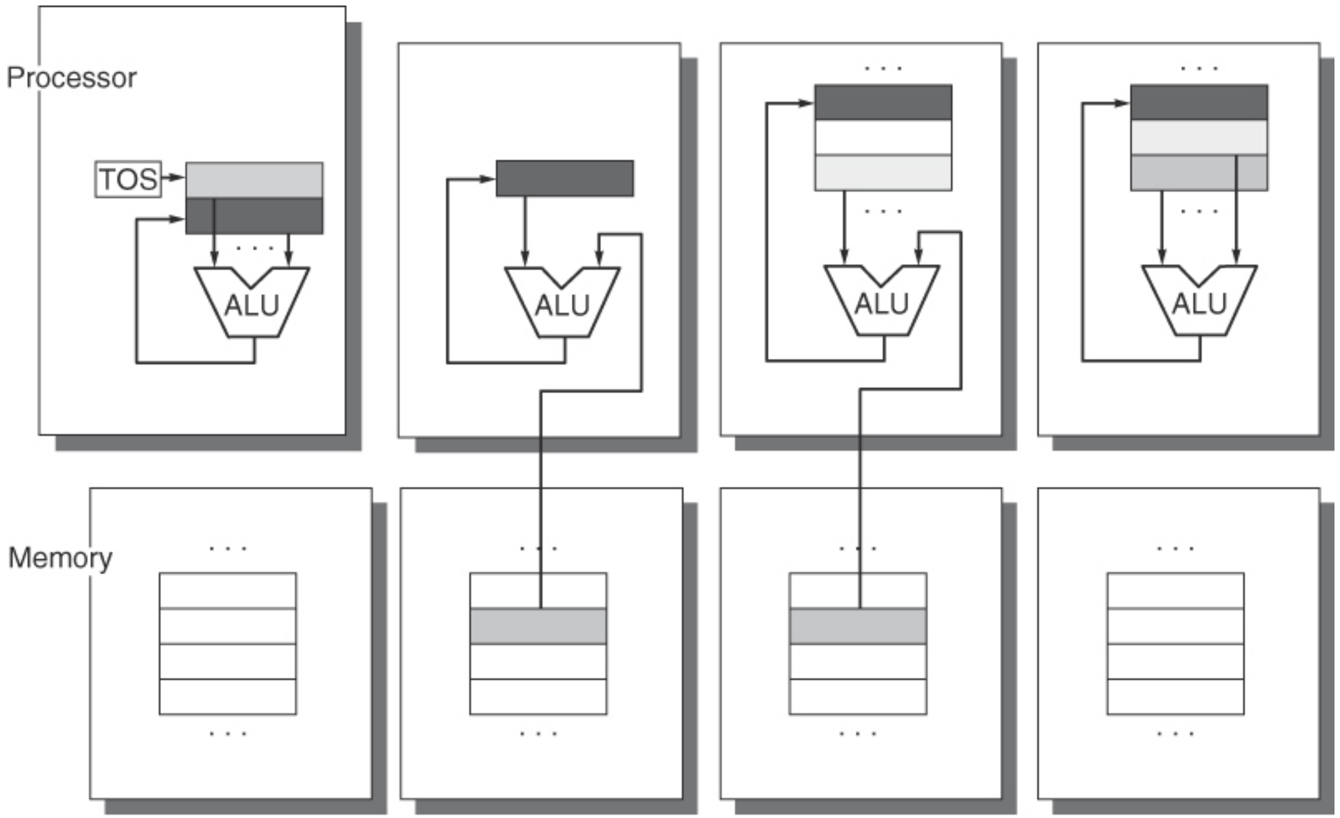
- Sometimes there *is* no operand, or no result, or no next instruction. Can you think of examples?

# Instructions Can Be Divided into 3 Classes (I)

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add*, *Sub*, *Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc*, *Brz Loc2*,—unconditional or conditional branches

# ISA Classification

- Type of internal storage in a processor is the most basic differentiator
- Stack Architecture
- Accumulator Architecture
- General Purpose Register Architecture




**Operand locations for four instruction set architecture classes.** The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS) points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).

Source: CA: A quantitative approach

# ISA Classification

| # Memory Address | Max. no. of operands allowed | Type of architecture | Examples |
|---|---|---|---|
| 0 | 3 | Load-Store | Alpha, ARM, MIPS, PowerPC |
| 1 | 2 | Reg-Mem | IBM360, Intel x86, 68000 |
| 2 | 2 | Mem-Mem | VAX |
| 3 | 3 | Mem-Mem | VAX |

# ISA Classification

| Type | Adv | Disadv |
|---|---|---|
| Reg-Reg | Simple, fixed length encoding, simple code generation, all instr. Take same no. of cycles | Higher instruction count, lower instruction density |
| Reg-Mem | Data can be accessed without separate load instruction first, instruction format tend to be easy to encode and yield good density | Encoding register no and memory address in each instruction may restrict the no. of registers. |
| Mem-Mem | Most compact, doesn't waste registers for temporaries | Large variation in instruction size, large variation in in amount of work (NOT USED TODAY) |

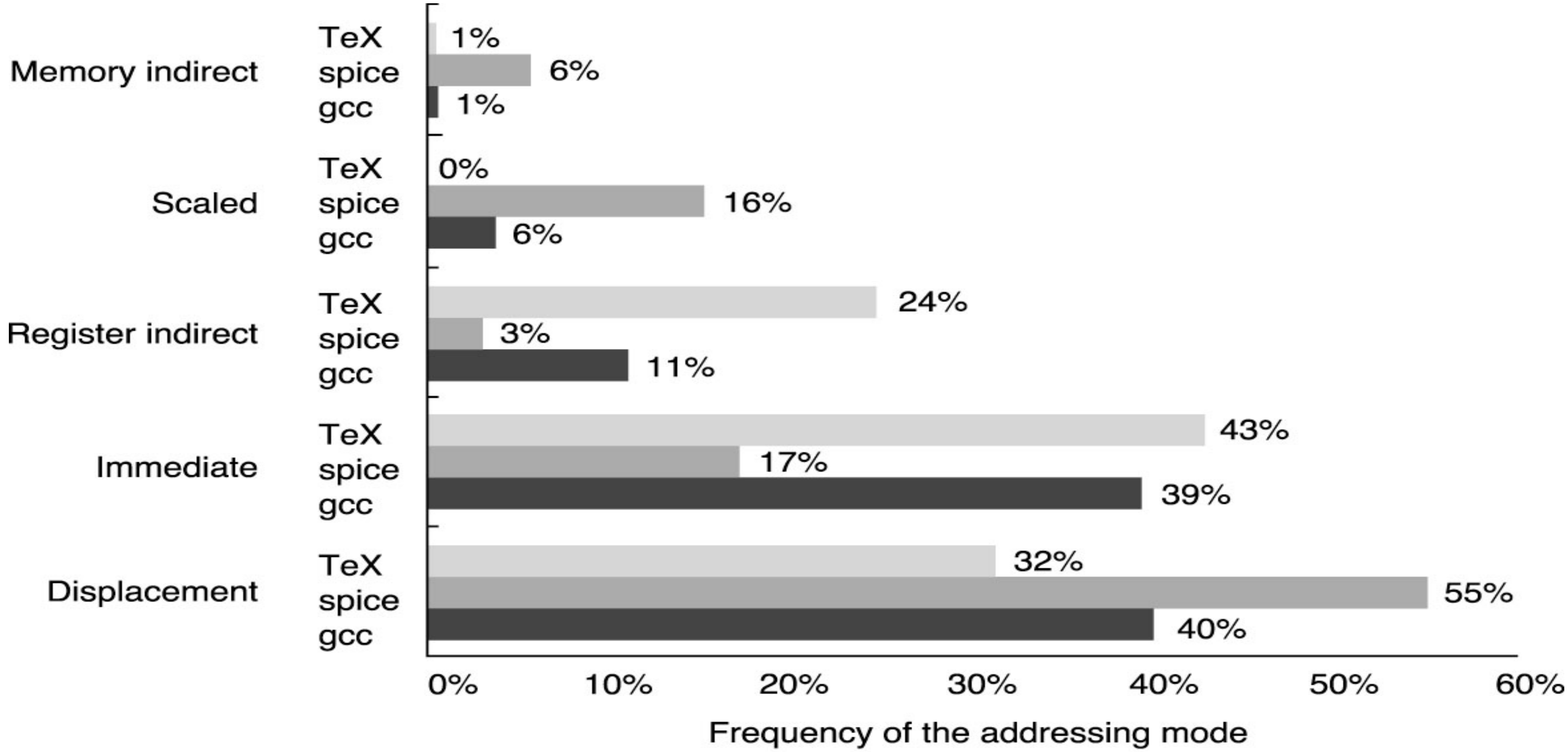
# Memory Address

- Interpreting memory address
  - Big Endian
  - Little Endian
- Instruction misalignment
- Addressing mode

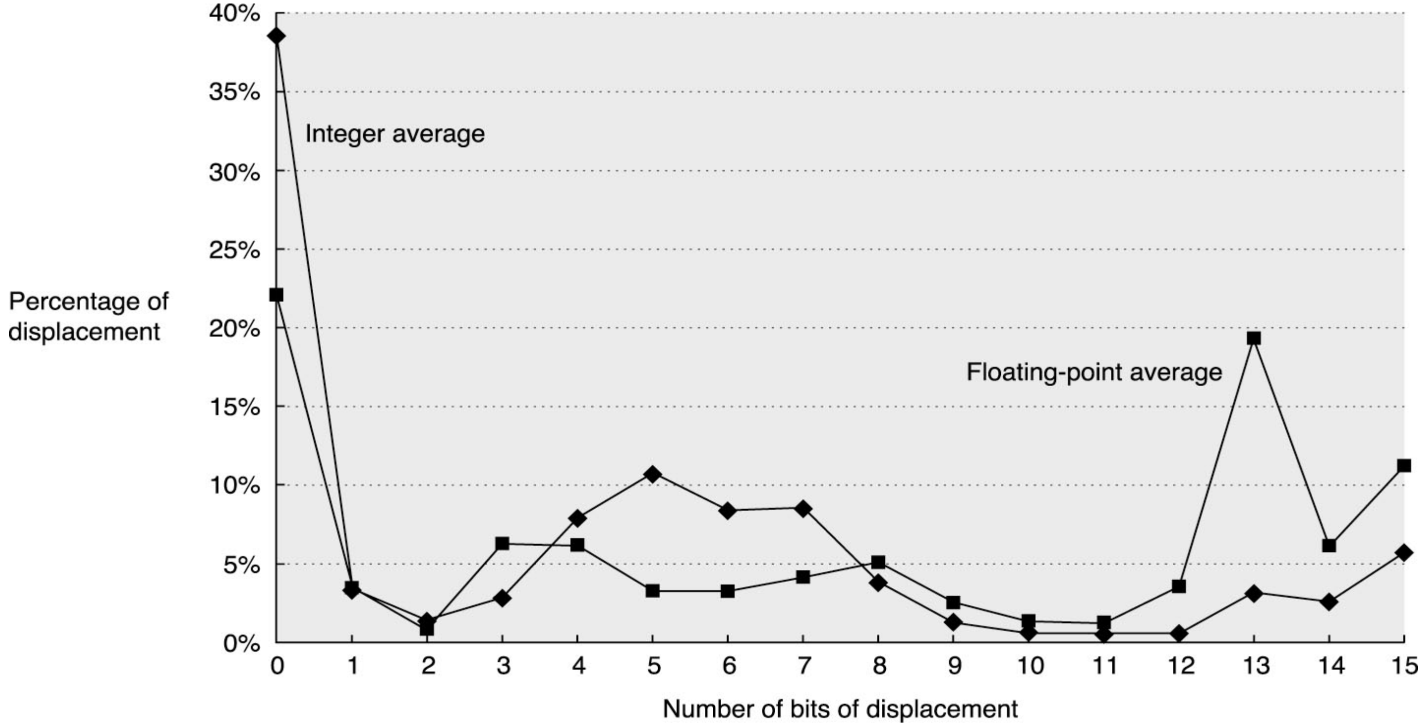
# Addressing Modes

- Register
- Immediate
- Register Indirect
- Displacement
- Indexed
- Direct Absolute
- Memory Indirect
- Auto Increment
- Auto decrement

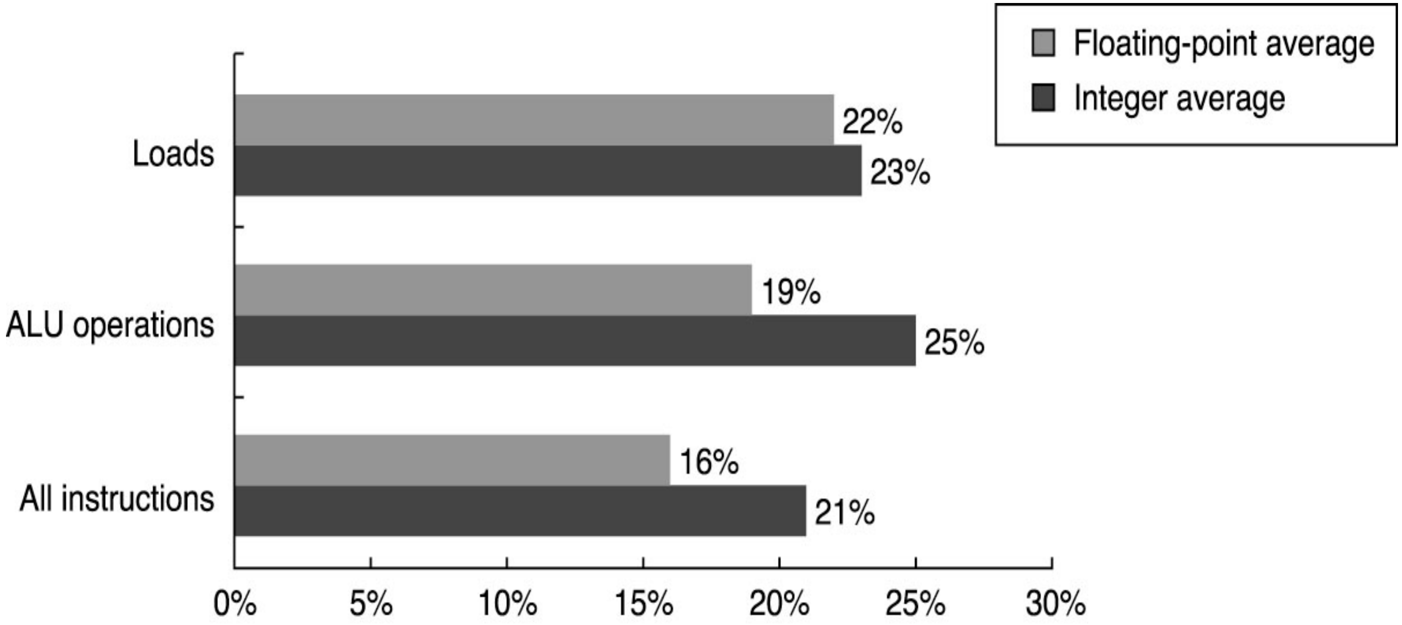
# Use of Addressing Modes



# Distribution of Displacement Values



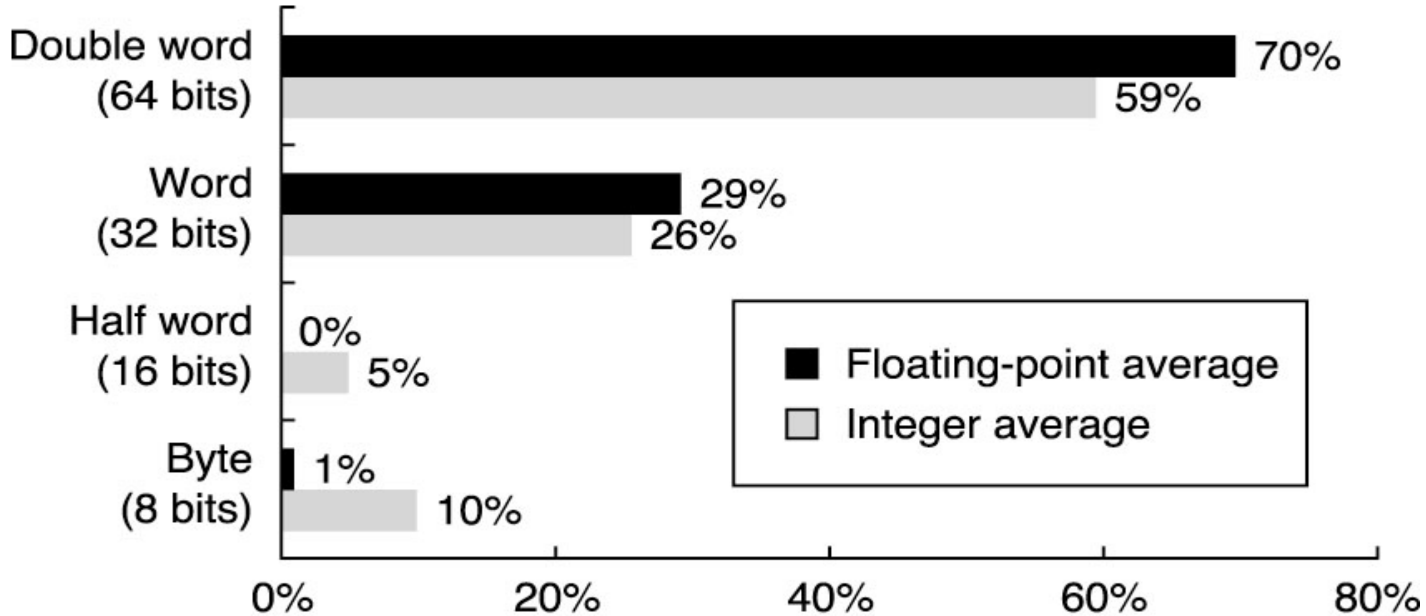
# Frequency of Immediate Operands



# Types of Operations

| | |
|---|---|
| ✓ Arithmetic and Logic: | AND, ADD |
| ✓ Data Transfer: | MOVE, LOAD, STORE |
| ✓ Control | BRANCH, JUMP, CALL |
| ✓ System | OS CALL, VM |
| ✓ Floating Point | ADDF, MULF, DIVF |
| ✓ Decimal | ADDD, CONVERT |
| ✓ String | MOVE, COMPARE |
| ✓ Graphics | (DE)COMPRESS |

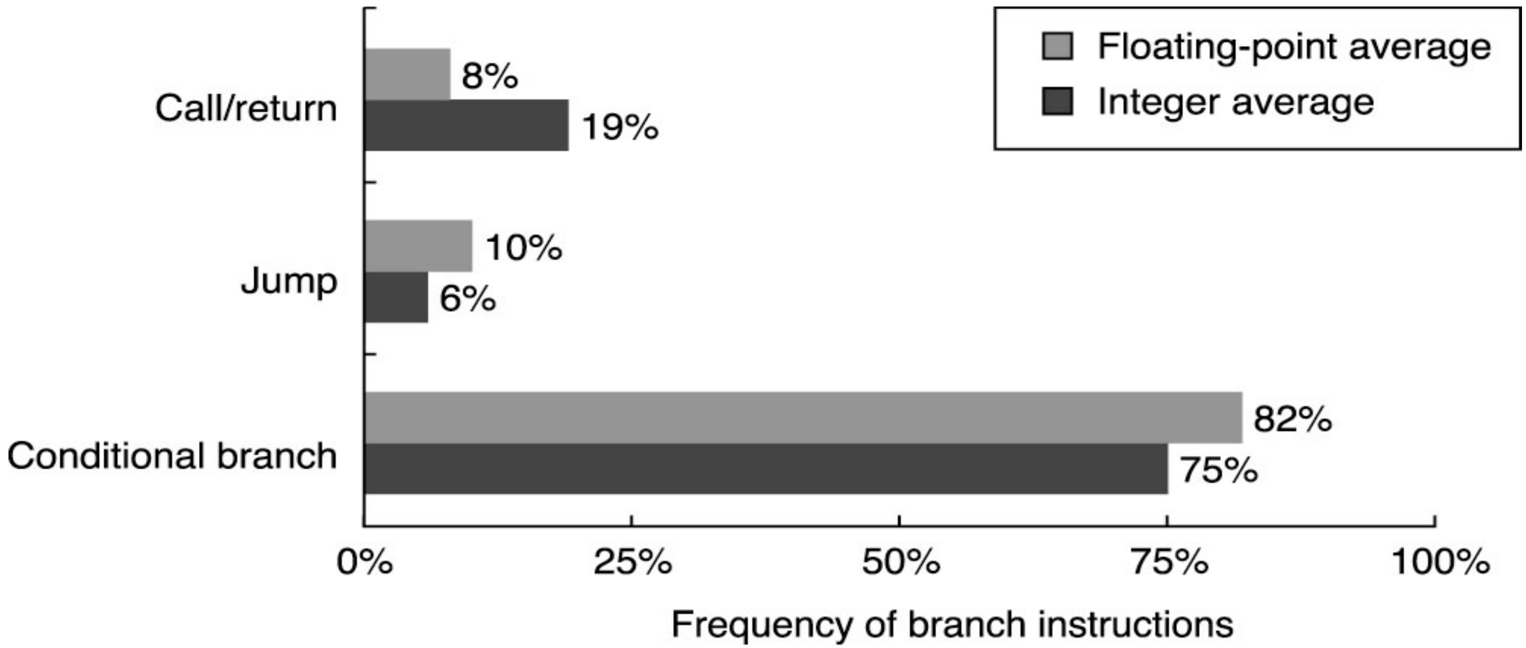
# Distribution of Data Accesses by Size



# 80x86 Instruction Frequency (SPECint92)

| *Rank* | *Instruction* | *Frequency* |
|---|---|---|
| 1 | load | 22% |
| 2 | branch | 20% |
| 3 | compare | 16% |
| 4 | store | 12% |
| 5 | add | 8% |
| 6 | and | 6% |
| 7 | sub | 5% |
| 8 | register move | 4% |
| 9 | call | 1% |
| 10 | return | 1% |
| Total | | 96% |

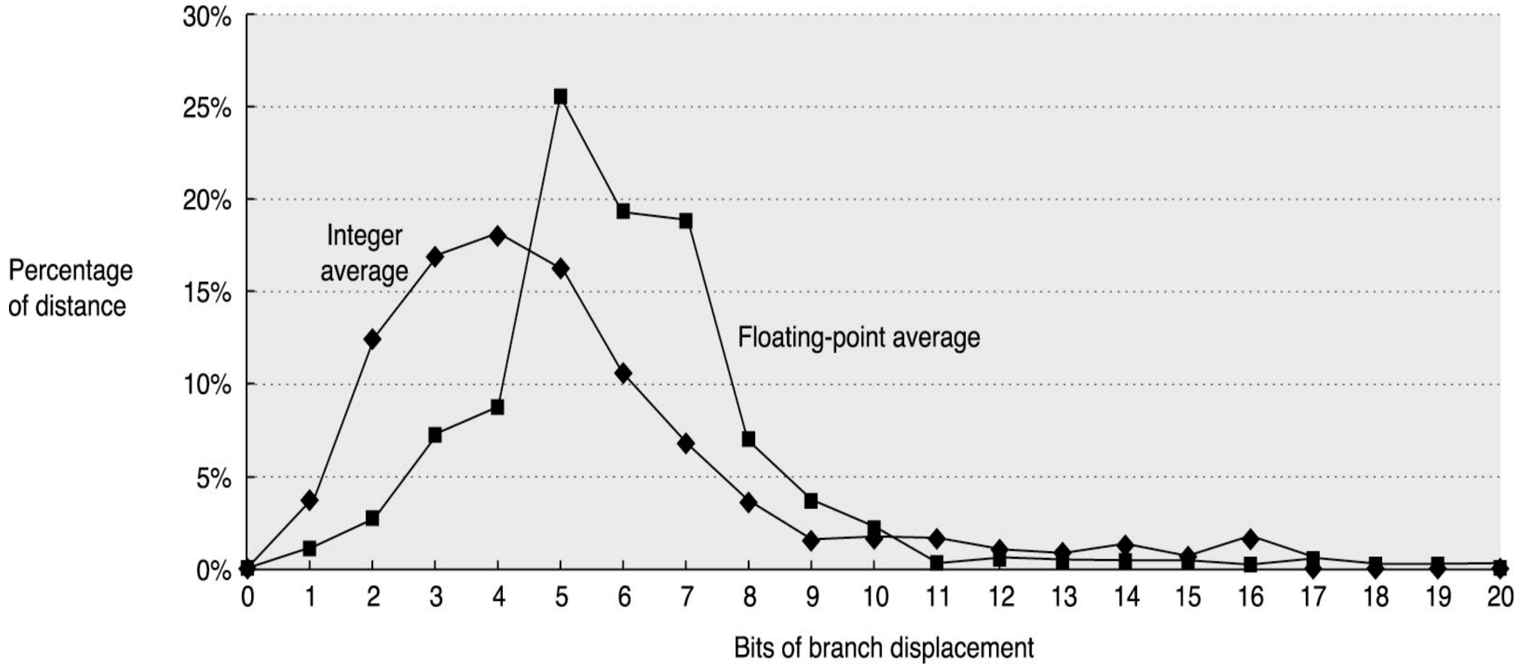
# Relative Frequency of Control Instructions



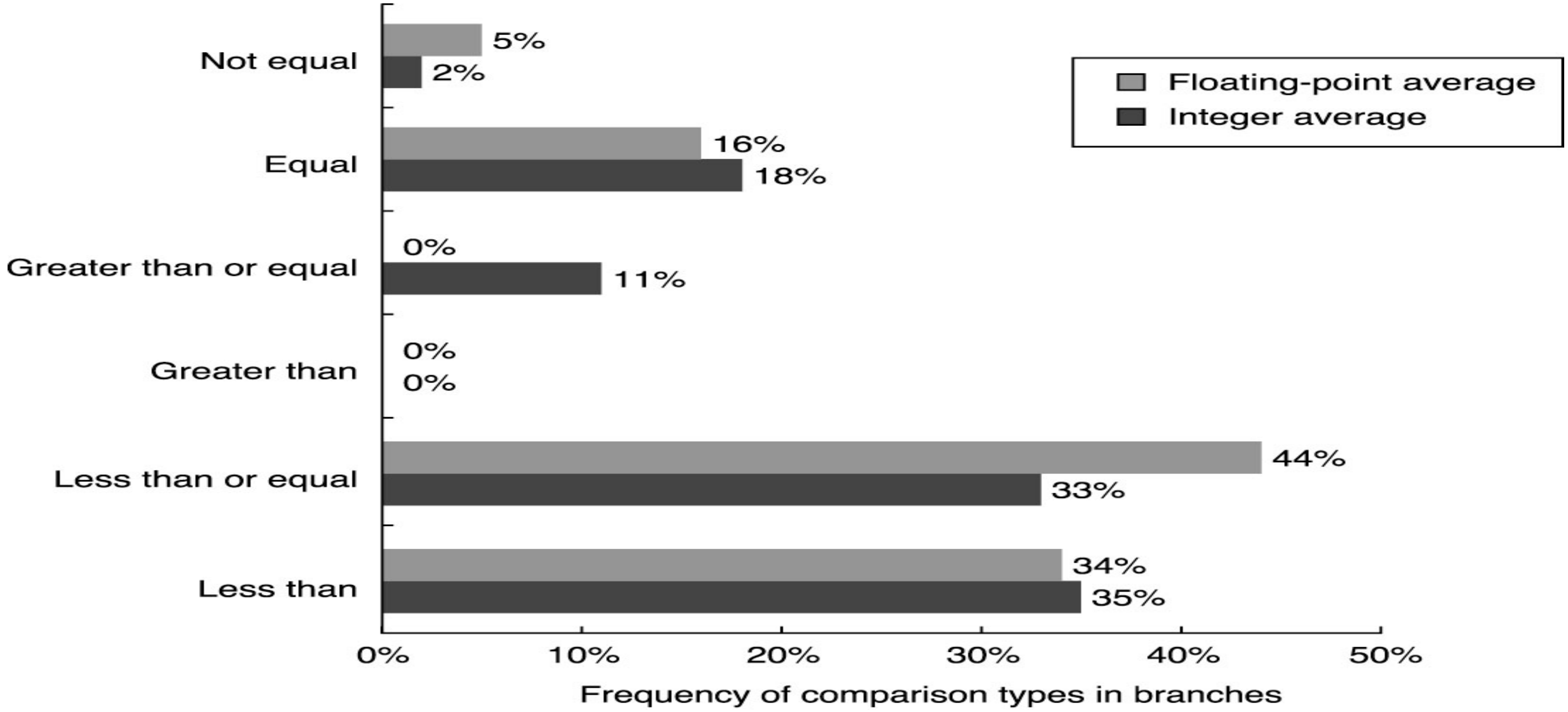
# Control instructions (contd.)

- Addressing modes
  - PC-relative addressing (independent of program load & displacements are close by)
    - Requires displacement (how many bits?)
    - Determined via empirical study. [8-16 works!]
  - For procedure returns/indirect jumps/kernel traps, target may not be known at compile time.
    - Jump based on contents of register
    - Useful for switch/(virtual) functions/function ptrs/ dynamically linked libraries etc.

# Branch Distances (in terms of number of instructions)



# Frequency of Different Types of Compares in Conditional Branches



# Encoding an Instruction set

- desire to have as many registers and addressing mode as possible
- the impact of size of register and addressing mode fields on the average instruction size and hence on the average program size
- a desire to have instruction encode into lengths that will be easy to handle in the implementation

# Three choice for encoding the instruction set

| | | | | | |
|---|---|---|---|---|---|
| Operation and no. of operands | Address specifier 1 | Address field 1 | • • • | Address specifier | Address field |

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

| | | | |
|---|---|---|---|
| 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 360/70, MIPS16, Thumb, TI TMS320C54x)

# RISC Design

# RISC Architecture

- Simple instructions
- Fixed Instruction Encoding
- Limited Addressing Mode
- Instruction count increases
- Simple controller
- Load/Store architecture
- Limited addressing modes

# Arithmetic Instructions

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code: **a = b + c + d;**

DLX code: **add a, b, c**
**add a, a, d**

- Operands must be registers
- 32 registers provided
- Each register contains 32 bits

# Instructions

- Load and store instructions
- Example:

C code: A[12] = h + A[8];

DLX code:
```
lw R1, 32(R3)   #addr of A in reg R3
add R1, R2, R1  #h in reg R2
sw R1, 48(R3)
```

- Can refer to registers by name (e.g., R2, R1) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: add 48(R3), R2, 32(R3)

# Memory Example

- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
    sll R2, R5, 2
    add R2, R4, R2
    lw R15, 0(R2)
    lw R16, 4(R2)
    sw R16, 0(R2)
    sw R15, 4(R2)
    jr R31
```

➢ Initially, k is in reg 5; addr of v is in reg 4; return addr is in reg 31

# Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be execute
- DLX conditional branch instructions:

```
bnez R1, Label
beqz R1, Label
```

- Example: `if (i/=0) h = 10 + j;`

```
      bnez R1, Label
      add R3, R2, 10
Label:     ....
```

# Control

- DLX unconditional branch instructions:
  **j label**
- Example:

```
if (i!=0)                        beqz R4,  Lab1
   h=10+j;                       add R3, R5, 10
else                             j Lab2
   h=j-32;             Lab1:     sub R3, $s5, 32
                                 Lab2:   ...
```

- Can you build a simple *for* loop?

# Four Ways to Jump

- j *addr* # jump to *addr*
- jr *reg* # jump to address in register *reg*
- jal *addr* # set R31=PC+4 and go to *addr* (jump and link)
- jalr *reg* # set R31=PC+4 and go to address in register *reg*

# Overview of DLX

- simple instructions, all 32 bits wide
- very structured, no unnecessary baggage
- only three  instruction formats

| | | | | | |
|---|---|---|---|---|---|
| R | op | rs1 | rs2 | rd | funct |
| I | op | rs1 | rd | 16 bit address | |
| J | op | 26 bit address | | | |

- rely on compiler to achieve performance

# Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate

## Design Principles:

  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

# DLX Instruction Set

| | | | |
|---|---|---|---|
| ADD Rd, Rs1, Rs2 | Rd ← Rs1 + Rs2<br>(overflow – exception) | R | 000_000<br>000_100 |
| SUB Rd, Rs1, Rs2 | Rd ← Rs1 - Rs2<br>(overflow – exception) | R | 000_000<br>000_110 |
| AND Rd, Rs1, Rs2 | Rd ← Rs1 and Rs2 | R | 000_000/ 001_000 |
| OR Rd, Rs1, Rs2 | Rd ← Rs1 or Rs2 | R | 000_000/ 001_001 |
| XOR Rd, Rs1, Rs2 | Rd ← Rs1 xor Rs2 | R | 000_000/ 001_010 |
| SLL Rd, Rs1, Rs2 | Rd ← Rs1 << Rs2 (logical)<br>(5 lsb of Rs2 are significant) | R | 000_000<br>001_100 |
| SRL Rd, Rs1, Rs2 | Rd ← Rs1 >> Rs2 (logical)<br>(5 lsb of Rs2 are significant) | R | 000_000<br>001_110 |
| SRA Rd, Rs1, Rs2 | Rd ← Rs1 >> Rs2 (arithmetic)<br>(5 lsb of Rs2 are significant) | R | 000_000<br>001_111 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| ADDI Rd, Rs1, Imm | Rd ← Rs1 + Imm (sign extended) (overflow – exception) | I | 010_100 |
| SUBI Rd, Rs1, Imm | Rd ← Rs1 – Imm (sign extended) (overflow – exception) | I | 010_110 |
| ANDI Rd, Rs1, Imm | Rd ← Rs1 and Imm (zero extended) | I | 011_000 |
| ORI Rd, Rs1, Imm | Rd ← Rs1 or Imm(zero extended) | I | 011_001 |
| XORI Rd, Rs1, Imm | Rd ← Rs1 xor Imm(zero extended) | I | 011_010 |
| SLLI Rd, Rs1, Imm | Rd ← Rs1 << Imm (logical) (5 lsb of Imm are significant) | I | 011_100 |
| SRLI Rd, Rs1, Imm | Rd ← Rs1 >> Imm (logical) (5 lsb of Imm are significant) | I | 011_110 |
| SRAI Rd, Rs1, Imm | Rd ← Rs1 >> Imm (arithmetic) (5 lsb of Imm are significant) | I | 011_111 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| LHI Rd, Imm | Rd(0:15) ← Imm<br>Rd(16:32) ← hex0000<br>(Imm: 16 bit immediate) | I | 011_011 |
| NOP | Do nothing | R | 000_000<br>000_000 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| SEQ Rd, Rs1, Rs2 | Rs1 = Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>010_000 |
| SNE Rd, Rs1, Rs2 | Rs1 /= Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>010_010 |
| SLT Rd, Rs1, Rs2 | Rs1 < Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>010_100 |
| SLE Rd, Rs1, Rs2 | Rs1 <= Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>010_110 |
| SGT Rd, Rs1, Rs2 | Rs1 > Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>011_000 |
| SGE Rd, Rs1, Rs2 | Rs1 >= Rs2: Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | R | 000_000<br>011_010 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| SEQI Rd, Rs1, Imm | Rs1 = Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000<br>(Imm: Sign extended 16 bit immediate) | I | 100_000 |
| SNEI Rd, Rs1, Imm | Rs1 /= Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | I | 100_010 |
| SLTI Rd, Rs1, Imm | Rs1 < Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | I | 100_100 |
| SLEI Rd, Rs1, Imm | Rs1 <= Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | I | 100_110 |
| SGTI Rd, Rs1, Imm | Rs1 > Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | I | 101_000 |
| SGEI Rd, Rs1, Imm | Rs1 >= Imm : Rd ← hex0000_0001<br>else: Rd ← hex0000_0000 | I | 101_010 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| BEQZ Rs, Label | Rs = 0: PC ← PC+4+Label<br>Rs /= 0: PC ← PC+4<br>(Label: Sign extended16 bit immediate) | I | 010_000 |
| BNEZ Rs, Label | Rs /= 0: PC ← PC+4+Label<br>Rs = 0: PC ← PC+4 | I | 010_001 |
| J Label | PC ← PC + 4 + sign_extd(imm26) | J | 001_100 |
| JAL Label | R31 ← PC + 4<br>PC ← PC+ 4 + sign_extd(imm26) | J | 001_100 |
| JAL Label | R31 ← PC + 4<br>PC ← PC+ 4 + sign_extd(imm26) | J | 001_101 |
| JR Rs | PC ← Rs | I | 001_110 |
| JALR Rs | R31 ← PC + 4<br>PC ← Rs | I | 001_111 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| LW Rd, Rs2 (Rs1) | Rd ← M(Rs1 + Rs2)<br>(word aligned address) | R | 000_000<br>100_000 |
| SW Rs2(Rs1), Rd | M(Rs1 + Rs2) ← Rd | R | 000_000<br>101_000 |
| LH Rd, Rs2 (Rs1) | Rd (16:31)← M(Rs1 + Rs2)<br>(Rd sign extended to 32 bit) | R | 000_000<br>100_001 |
| SH Rs2(Rs1), Rd | M(Rs1 + Rs2) ← Rd(16:31) | R | 000_000<br>101_001 |
| LB Rd, Rs2 (Rs1) | Rd (24:31)← M(Rs1 + Rs2)<br>(Rd sign extended to 32 bit) | R | 000_000<br>101_010 |
| SB Rs2(Rs1), Rd | M(Rs1 + Rs2) ← Rd(24:31) | R | 000_000<br>101_010 |

# DLX Instruction Set

| | | | |
|---|---|---|---|
| LWI Rd, Imm (Rs) | Rd ← M(Rs + Imm)<br>(Imm: sign extended 16 bit)<br>(word aligned address) | I | 000_100 |
| SWI Imm(Rs), Rd | M(Rs + Imm) ← Rd | I | 001_000 |
| LHI Rd, Imm (Rs) | Rd (16:31)← M(Rs + Imm)<br>(Rd sign extended to 32 bit) | I | 000_101 |
| SHI Imm(Rs), Rd | M(Rs1 + Rs2) ← Rd(16:31) | I | 001_001 |
| LBI Rd, Imm (Rs) | Rd (24:31)← M(Rs + Imm)<br>(Rd sign extended to 32 bit) | I | 000_110 |
| SBI Imm(Rs), Rd | M(Rs + Imm) ← Rd(24:31) | I | 001_010 |

# Thank You