Computer System

Instruction Set Architecture

Virendra Singh

Associate Professor

Computer **A**rchitecture and **D**ependable **S**ystems **L**ab

Department of Electrical Engineering

Indian Institute of Technology Bombay

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

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

CP-226:Computer Architecture

Lecture 6 (08 Feb 2013)

CADSL

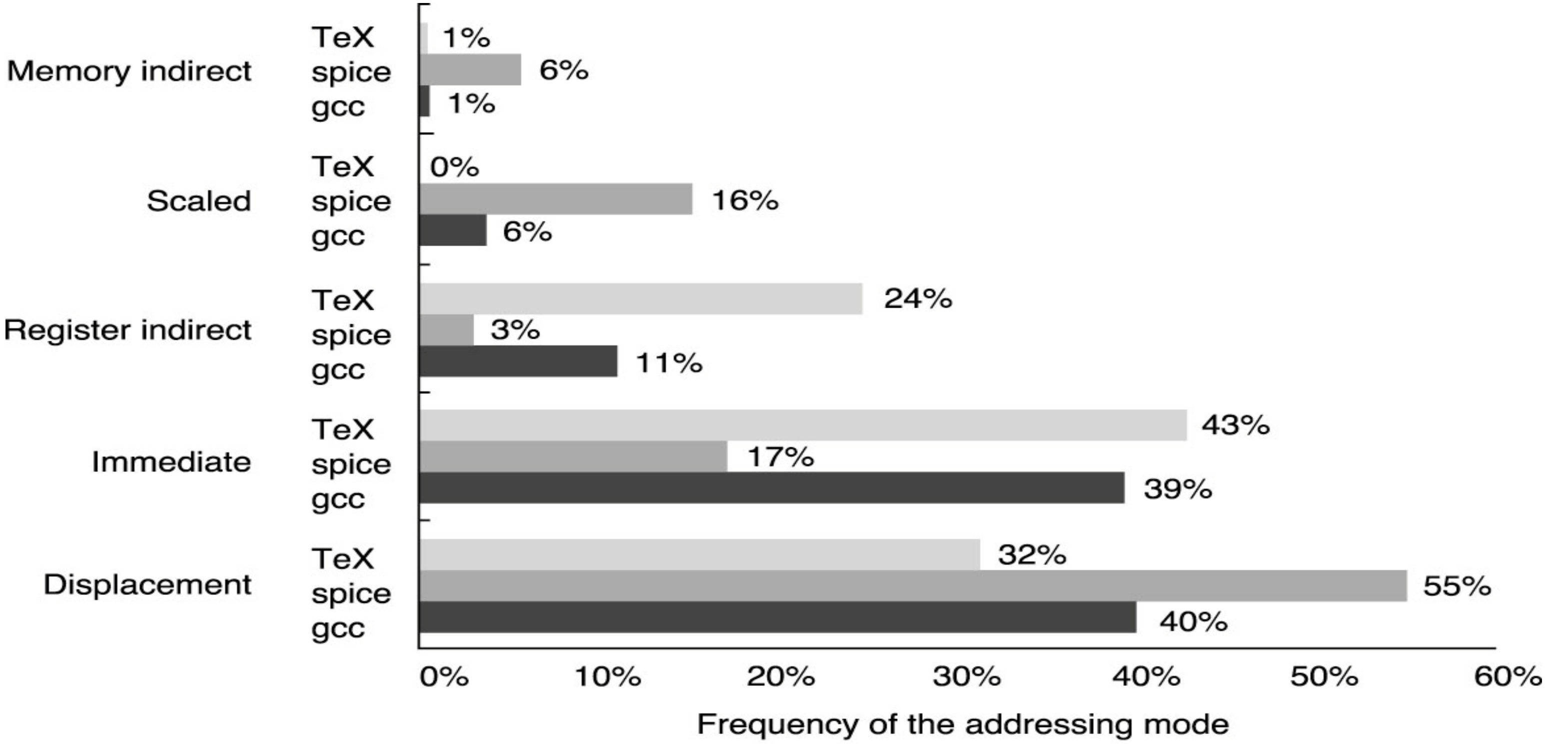
ISA Classification

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)

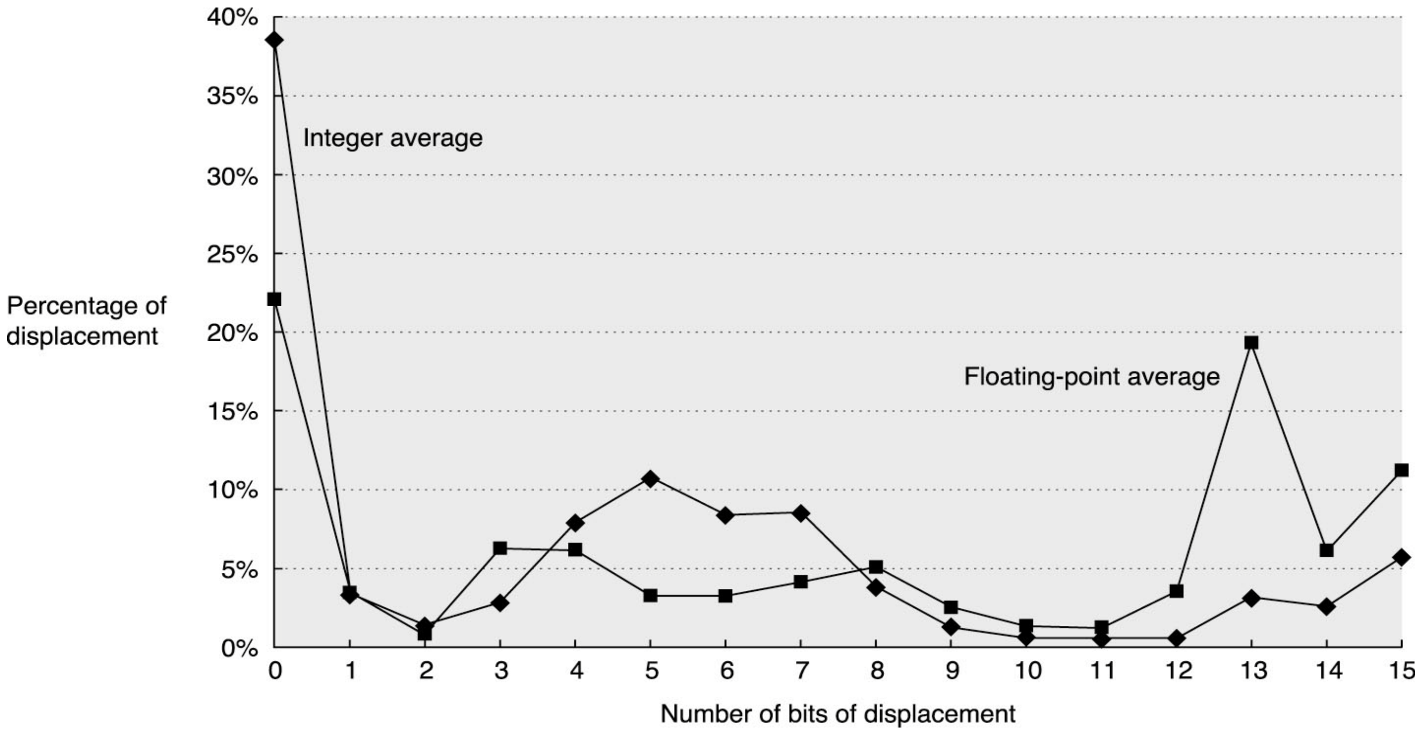
Memory Address

- Interpreting memory address
  - Big Endian (0 1 2 3)
  - Little Endian (3 2 1 0)
- Instruction misalignment
- Addressing mode

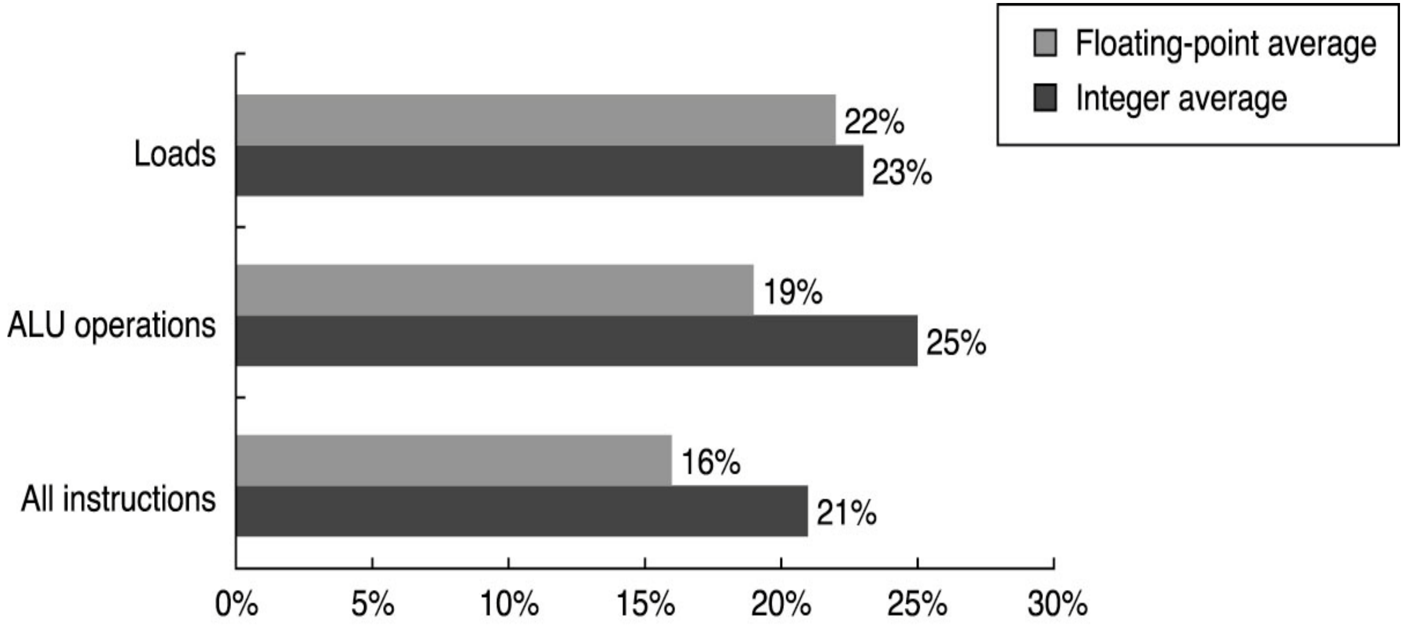
Summary of Use of Addressing Modes



Distribution of Displacement Values



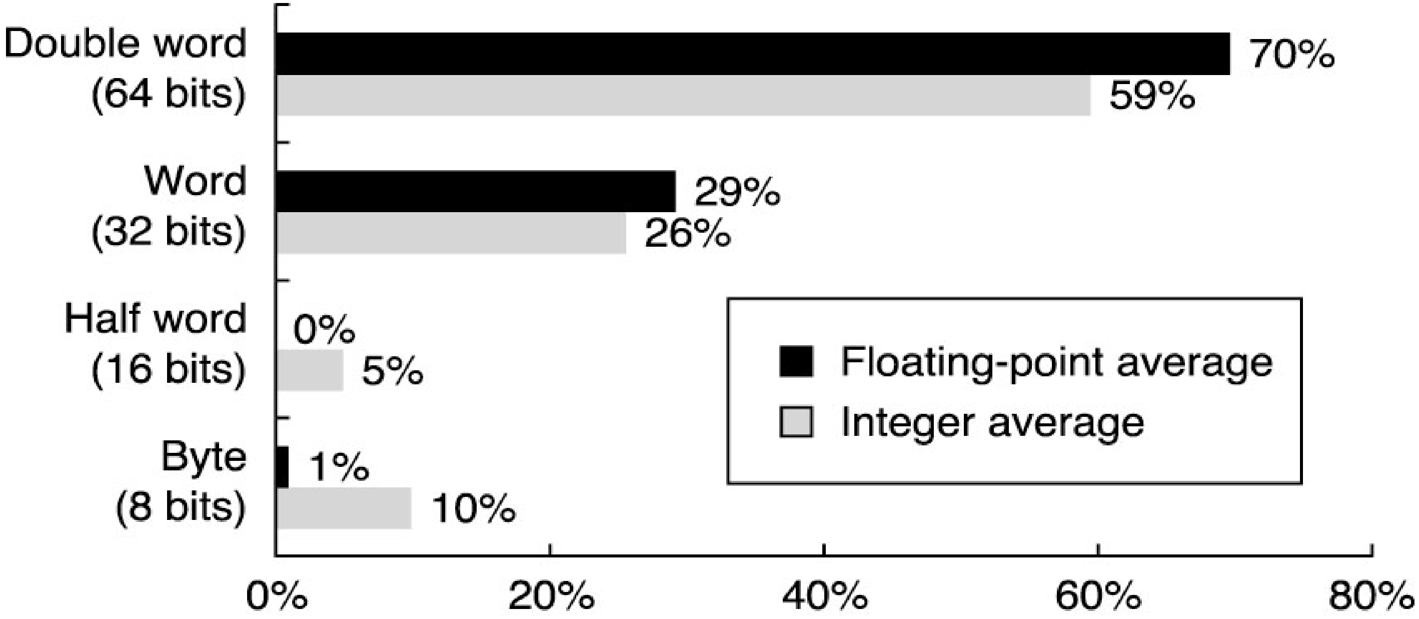
Frequency of Immediate Operands



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

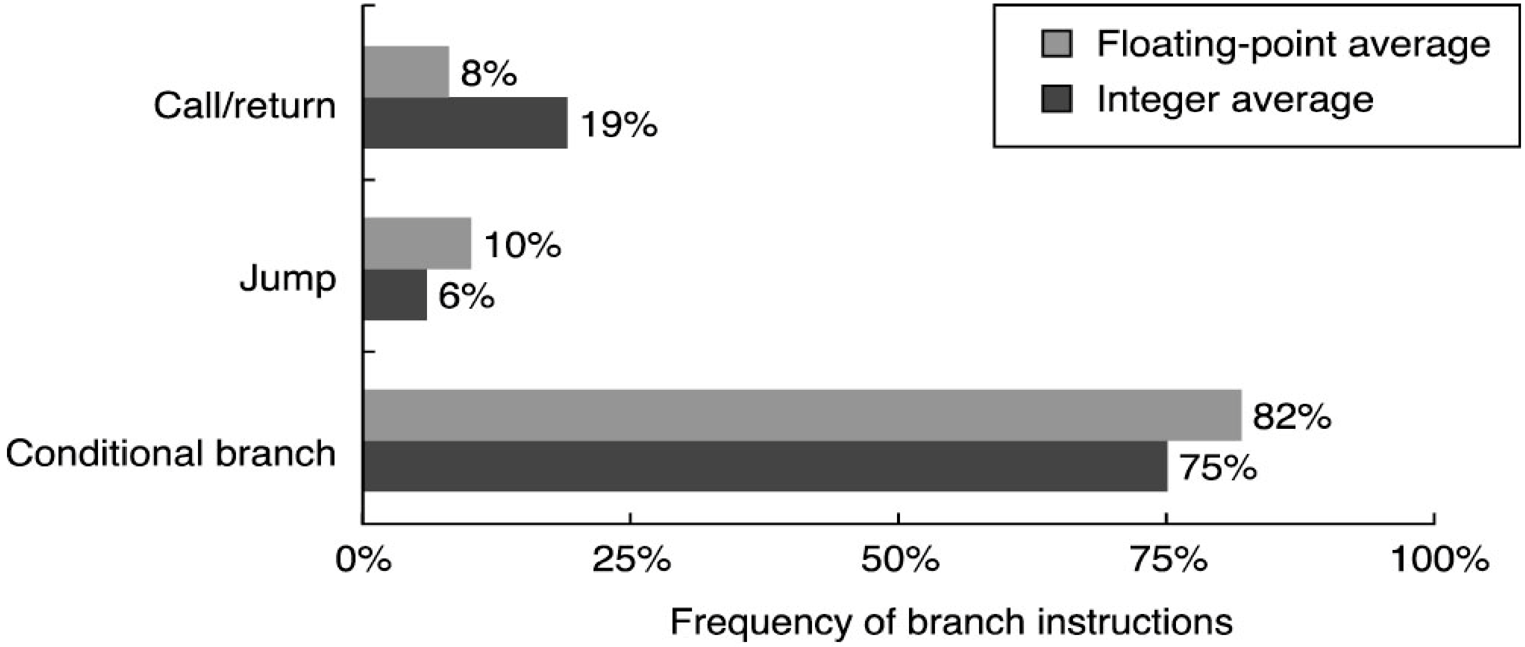
Distribution of Data Accesses by Size



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%**

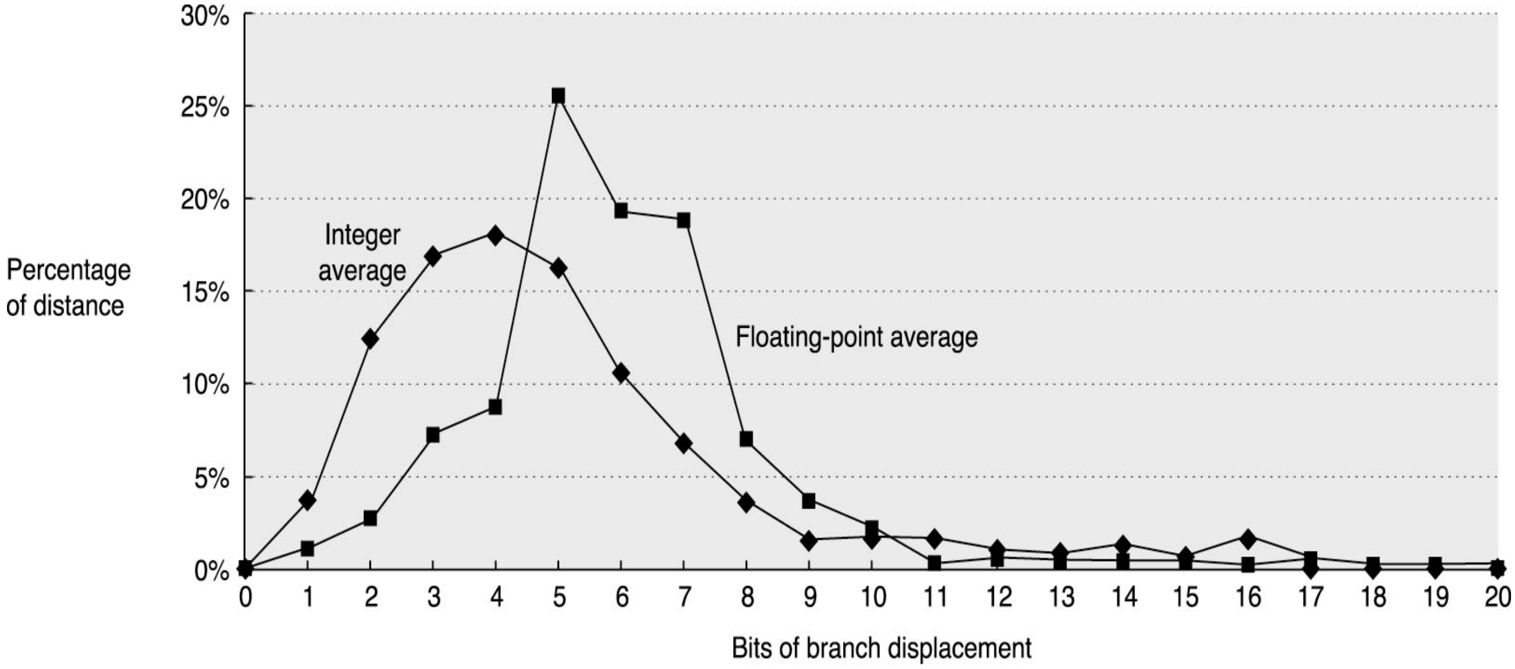
Relative Frequency of Control Instructions



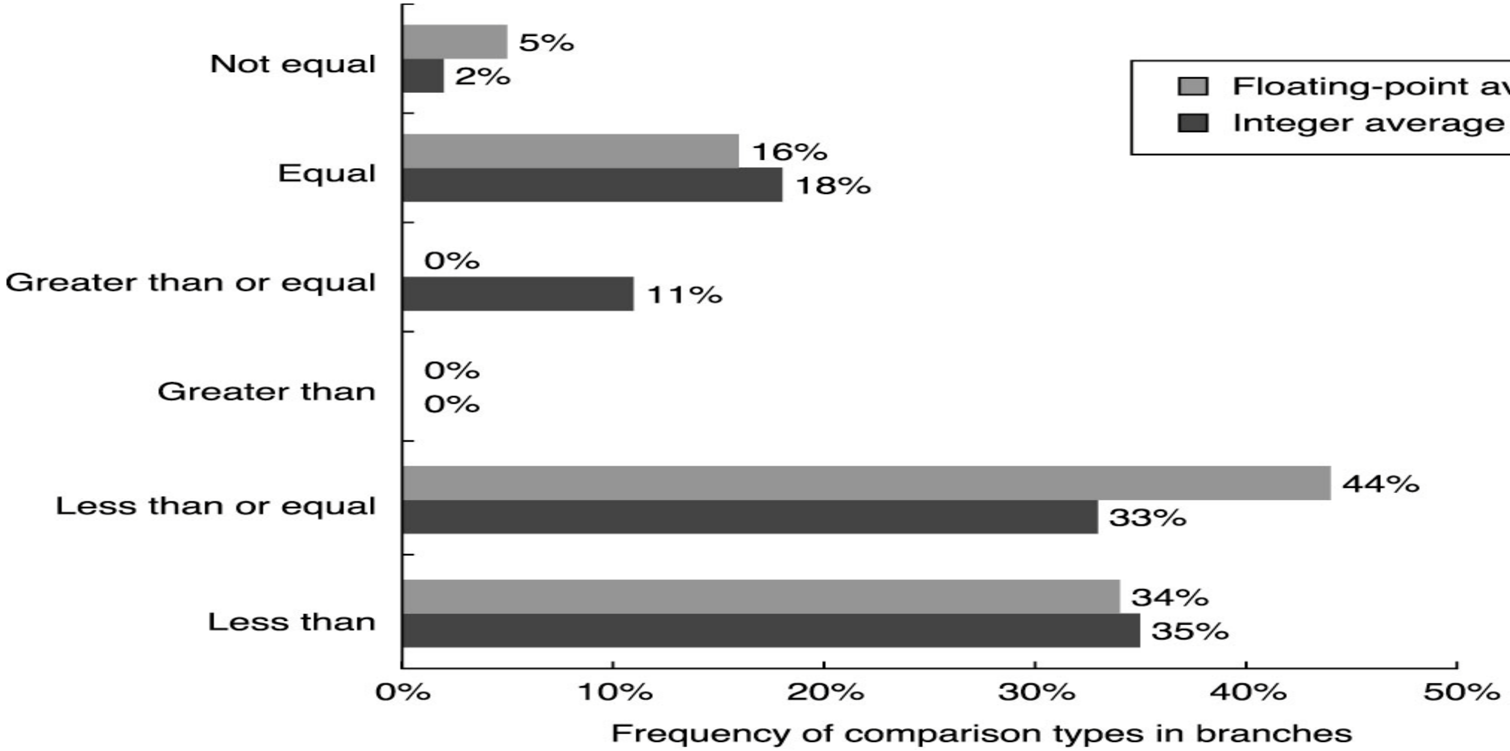
Control instructions (contd.)

- 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)



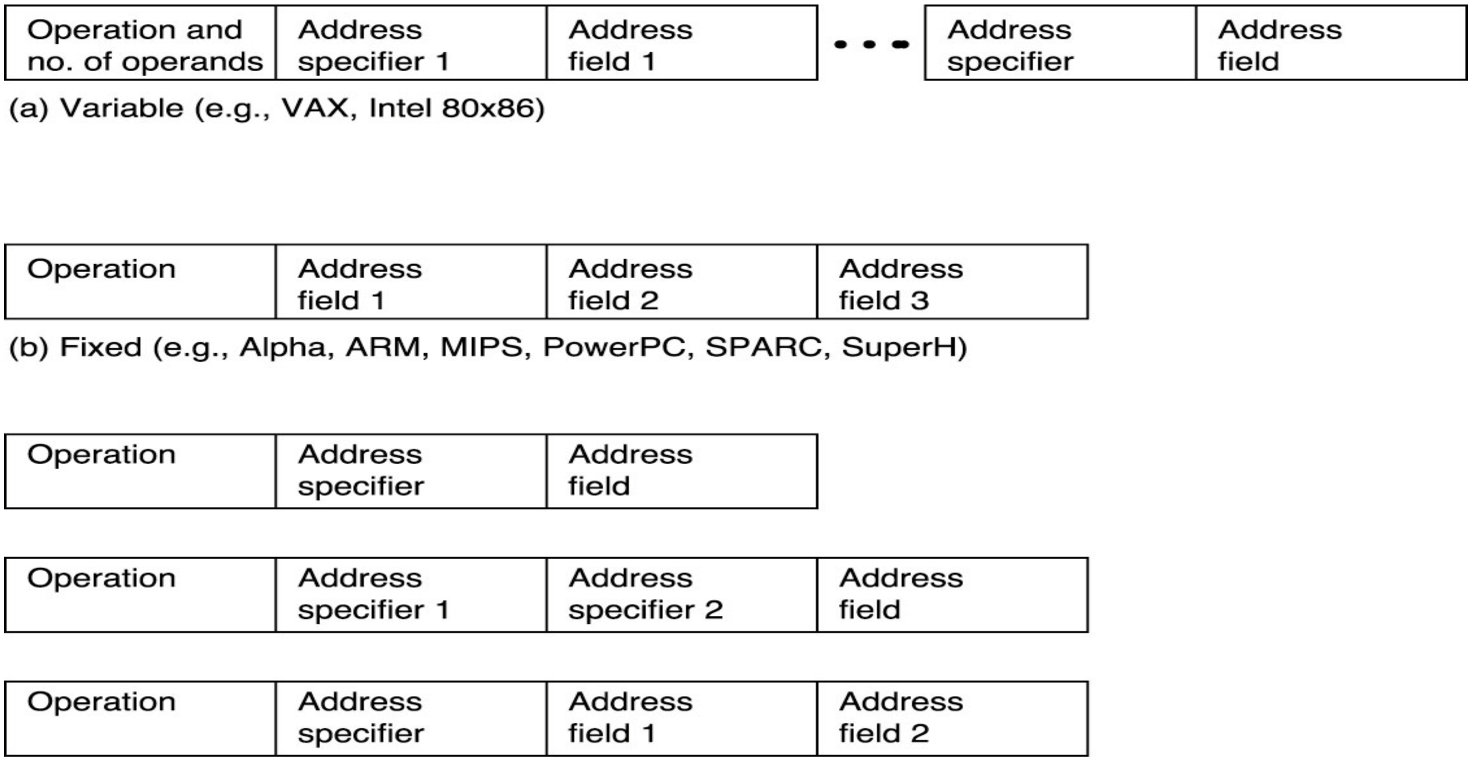
Frequency of Different Types of Compares in Conditional Branches



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;**

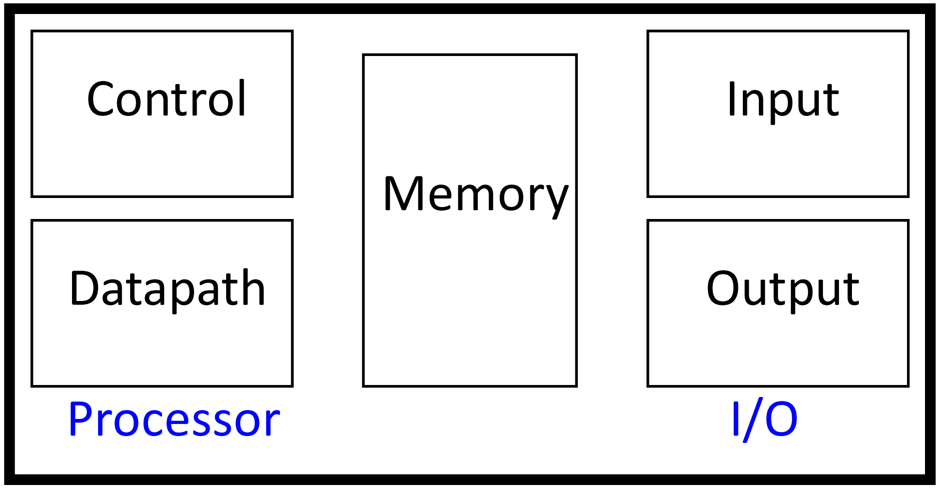
DLX code:

```
add a, b, c
add a, a, d
```

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

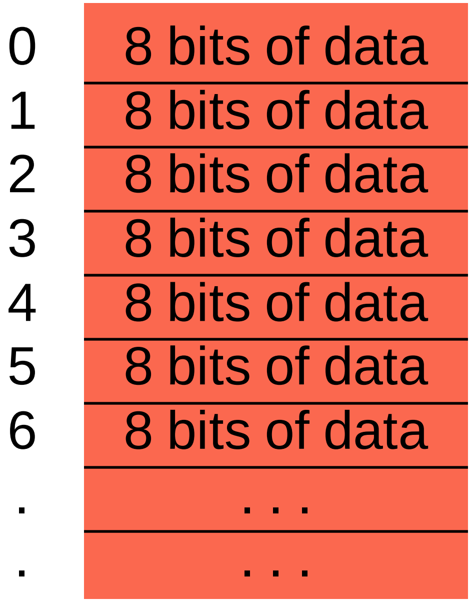
Registers vs. Memory

- Arithmetic instructions operands must be registers
  - **32 registers provided**
- Compiler associates variables with registers.
- What about programs with lots of variables? Must use memory.



Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.



0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
.	. . .
.	. . .

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For DLX, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
.	...
.	...

...

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32} – 1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32} – 4$
- Words are aligned

  i.e., what are the least 2 significant bits of a word address?

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

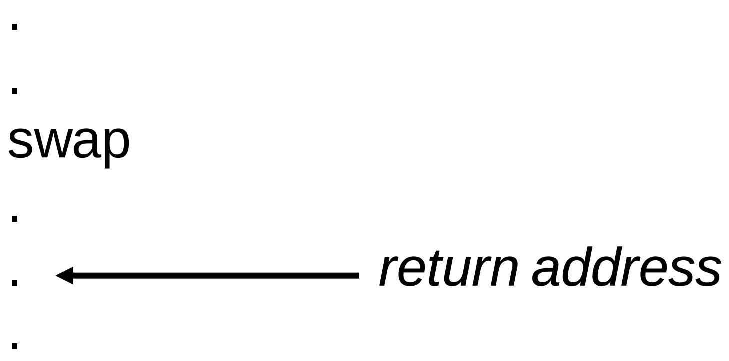
- Can we figure out the code?

```
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

What Happens?



.

.

swap

.

.

.

return address

- When the program reaches swap statement:
  - Jump to swap routine
    - Registers 4 and 5 contain the arguments
    - Register 31 contains the return address
  - Swap two words in memory
  - Jump back to return address to continue rest of the program

Memory and Registers

byte addr.	
0	Word 0
4	Word 1
8	Word 2
12	
.	
4n	v[0] (Word n)
.	v[1] (Word n+1)
.	
.	
4n+4k	v[k] (Word n+k)
.	v[k+1] (Word n+k+1)

Register 0	
Register 1	
Register 2	
Register 3	
Register 4	4n
Register 5	k
.	
.	
Register 31	*jump addr.*

Machine Language

- Instructions, like registers and words of data, are also 32 bits lon
  - Example: `add R1, R2, R3`
- Instruction Format:

000000	00001	00010	00011	00000	100000
op	rs1	rs2	rd		funct

- *Can you guess what the field names stand for?*

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

Addresses in Branches and Jumps

- Instructions:

`bnez R4, R5, Label` Next instruction is at Label

if **R4 ≠ R5**

`beqz R4, R5, Label` Next instruction is at Label

if **R4 = R5**

`j Label` Next instruction is at Label

- Formats:

op	rs	rd	16 bit rel. address

I

op	26 bit absolute address

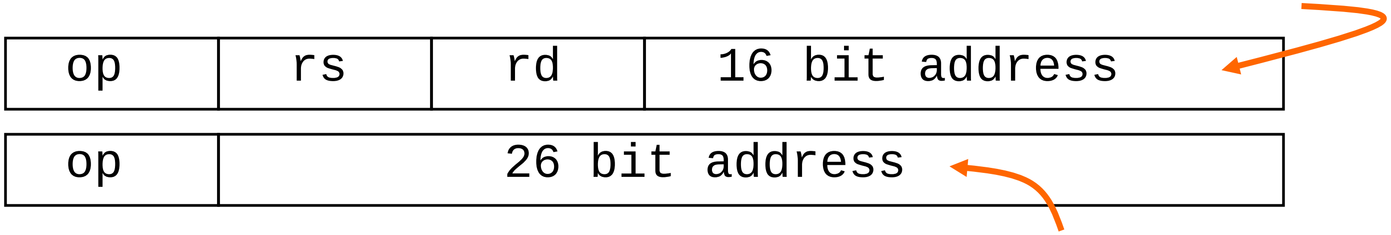
J

Addresses in Branches

- Instructions:

  `bnez R4,Label` Next instruction is at Label if R4 ≠ **0**
  `beqz R4,Label` Next instruction is at Label if R4 = 0

- Formats:



op	rs	rd	16 bit address

op	26 bit address

2^{26} = 64 Mwords

- Relative addressing
  - with respect to PC (program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MBytes (maximum jump 64 Mwords)

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate – *we will see performance measures later*
- 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!

Instruction Set

Register-Register Instructions

Opcode	Rs1	Rs2	Rd		func

0 5 10 15 20 25 31

Arithmetic and Logical Instruction

- ADD Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1] + Reg[Rs2]
- SUB Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1] - Reg[Rs2]
- AND Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1] and Reg[Rs2]
- OR Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1] or Reg[Rs2]
- XOR Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1] xor Reg[Rs2]

•SUB Rd, Rs1, Rs2 Regs[Rd] <= Reg[Rs1]-Reg[Rs2]

DLX Instruction Set

ADD Rd, Rs1, Rs2	Rd ← Rs1 + Rs2 (overflow – exception)	R	000_000 000_100
SUB Rd, Rs1, Rs2	Rd ← Rs1 - Rs2 (overflow – exception)	R	000_000 000_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) (5 lsb of Rs2 are significant)	R	000_000 001_100
SRL Rd, Rs1, Rs2	Rd ← Rs1 >> Rs2 (logical) (5 lsb of Rs2 are significant)	R	000_000 001_110
SRA Rd, Rs1, Rs2	Rd ← Rs1 >> Rs2 (arithmetic) (5 lsb of Rs2 are significant)	R	000_000 001_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 Rd(16:32) ← hex0000 (Imm: 16 bit immediate)	I	011_011
NOP	Do nothing	R	000_000 000_000

DLX Instruction Set

SEQ Rd, Rs1, Rs2	Rs1 = Rs2: Rd ← hex0000_0001
	else: Rd ← hex0000_0000
		R	000_000
			010_000

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

DLX Instruction Set

SEQI Rd, Rs1, Imm	Rs1 = Imm : Rd ← hex0000_0001
	else: Rd ← hex0000_0000
	(Imm: Sign extended 16 bit immediate)
		I	100_000

SNEI Rd, Rs1, Imm	Rs1 /= Imm : Rd ← hex0000_0001 else: Rd ← hex0000_0000	I	100_010
SLTI Rd, Rs1, Imm	Rs1 < Imm : Rd ← hex0000_0001 else: Rd ← hex0000_0000	I	100_100
SLEI Rd, Rs1, Imm	Rs1 <= Imm : Rd ← hex0000_0001 else: Rd ← hex0000_0000	I	100_110
SGTI Rd, Rs1, Imm	Rs1 > Imm : Rd ← hex0000_0001 else: Rd ← hex0000_0000	I	101_000
SGEI Rd, Rs1, Imm	Rs1 >= Imm : Rd ← hex0000_0001 else: Rd ← hex0000_0000	I	101_010

DLX Instruction Set

BEQZ Rs, Label	Rs = 0: PC ← PC+4+Label Rs /= 0: PC ← PC+4 (Label: Sign extended16 bit immediate)	I	010_000
BNEZ Rs, Label	Rs /= 0: PC ← PC+4+Label Rs = 0: PC ← PC+4	I	010_001
J Label	PC ← PC + 4 + sign_extd(imm26)	J	001_100

JAL Label	R31 ← PC + 4
	PC ← PC+ 4 + sign_extd(imm26)
		J	001_100

JAL Label	R31 ← PC + 4
	PC ← PC+ 4 + sign_extd(imm26)
		J	001_101

JR Rs	PC ← Rs	I	001_110

JALR Rs	R31 ← PC + 4
	PC ← Rs
		I	001_111

DLX Instruction Set

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

DLX Instruction Set

LWI Rd, Imm (Rs)	Rd ← M(Rs + Imm) (Imm: sign extended 16 bit) (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) (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) (Rd sign extended to 32 bit)	I	000_110
SBI Imm(Rs), Rd	M(Rs + Imm) ← Rd(24:31)	I	001_010

Thank You