Lecture – 2

Instruction Set Architectures

Dr. Soner Onder

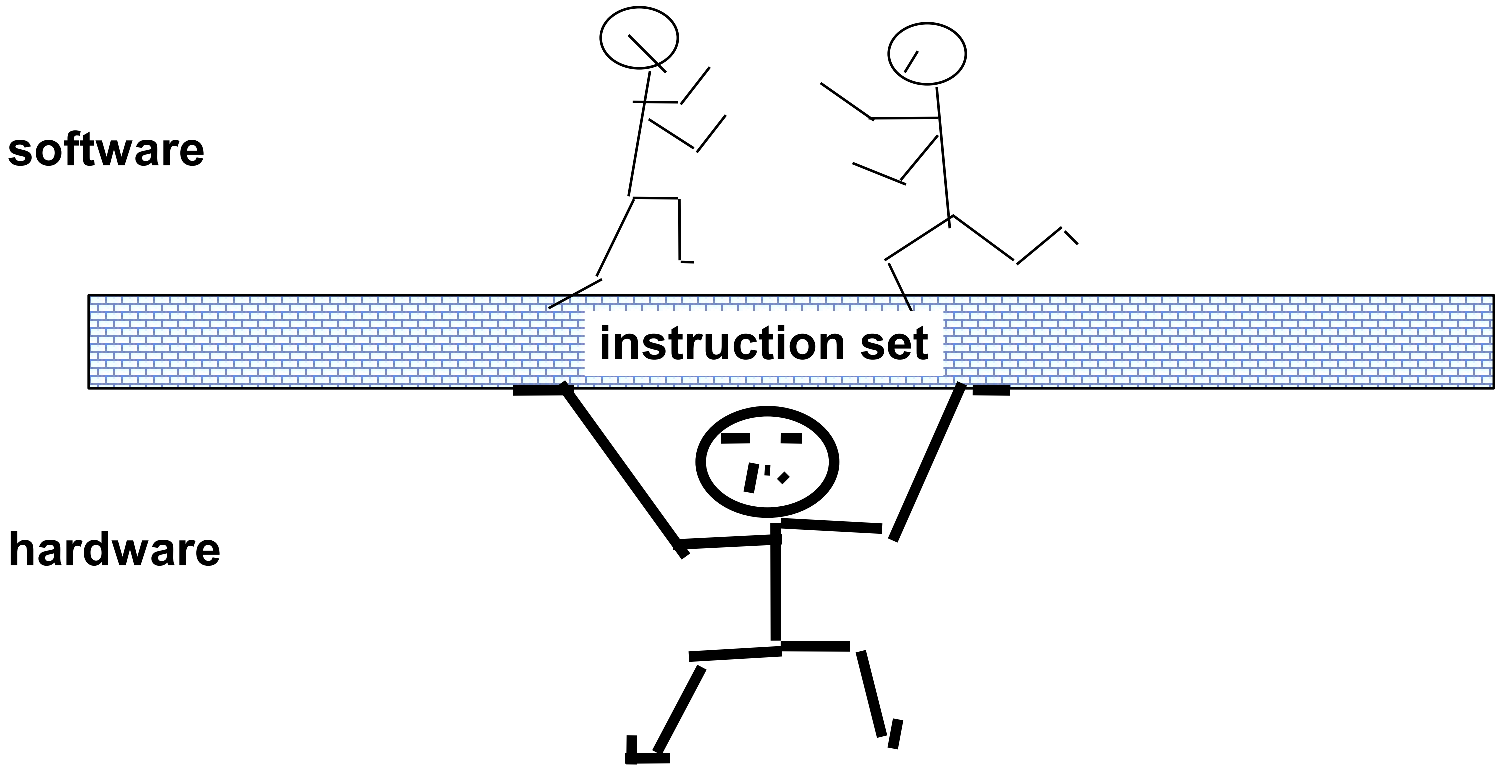
CS 4431

Michigan Technological University

Instruction Set Architecture (ISA)

- 1950s to 1960s: Computer Architecture Course Computer Arithmetic
- 1970 to mid 1980s: Computer Architecture Course Instruction Set Design, especially ISA appropriate for compilers
- 1990s: Computer Architecture Course Design of CPU, memory system, I/O system, Multiprocessors

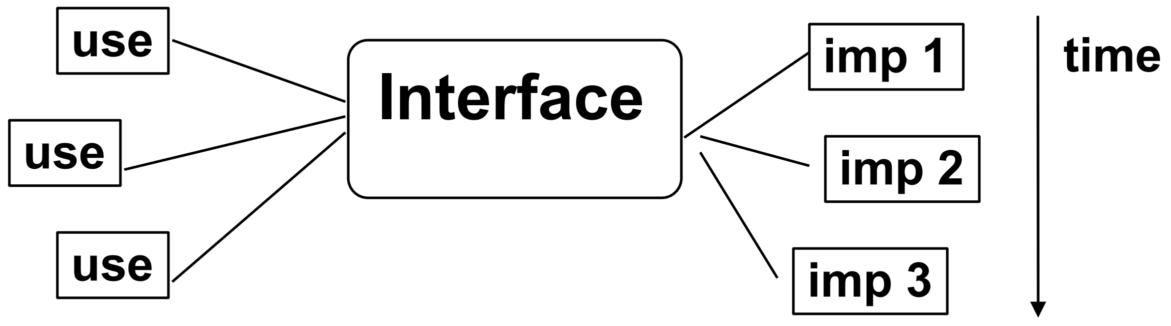
Instruction Set Architecture (ISA)



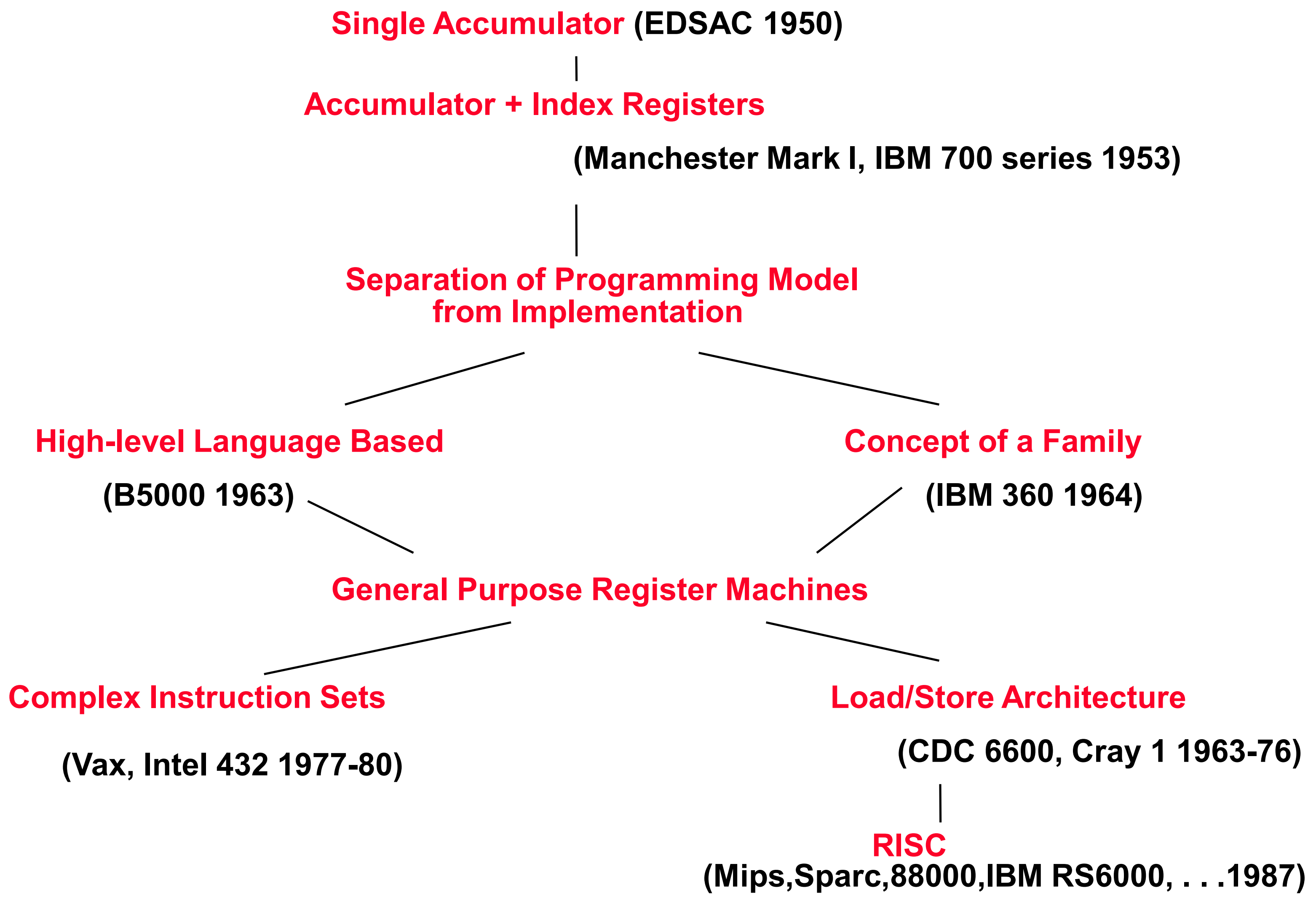
Interface Design

A good interface:

- **Lasts through many implementations (portability, compatability)**
- **Is used in many differeny ways (generality)**
- **Provides convenient functionality to higher levels**
- **Permits an efficient implementation at lower levels**



Evolution of Instruction Sets


Single Accumulator (EDSAC 1950)
Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)
Separation of Programming Model from Implementation
High-level Language Based
(B5000 1963)
Concept of a Family
(IBM 360 1964)
General Purpose Register Machines
Complex Instruction Sets
(Vax, Intel 432 1977-80)
Load/Store Architecture
(CDC 6600, Cray 1 1963-76)
RISC
(Mips,Sparc,88000,IBM RS6000, . . .1987)

Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they in turn influence these

Design Space of ISA

Five Primary Dimensions

- Number of explicit operands(0, 1, 2, 3)
- Operand Storage Where besides memory?
- Effective Address How is memory location specified?
- Type & Size of Operands byte, int, float, vector, . . .
  How is it specified?
- Operations add, sub, mul, . . .
  How is it specified?

Other Aspects

- Successor How is it specified?
- Conditions How are they determined?
- Encoding Fixed or variable? Wide?
- Parallelism

ISA Metrics

Aesthetics:

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type
- Completeness
  - Support for a wide range of operations and target applications
- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined
  - Resource needs easily determined

Ease of compilation (programming?)

Ease of implementation

Scalability

Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

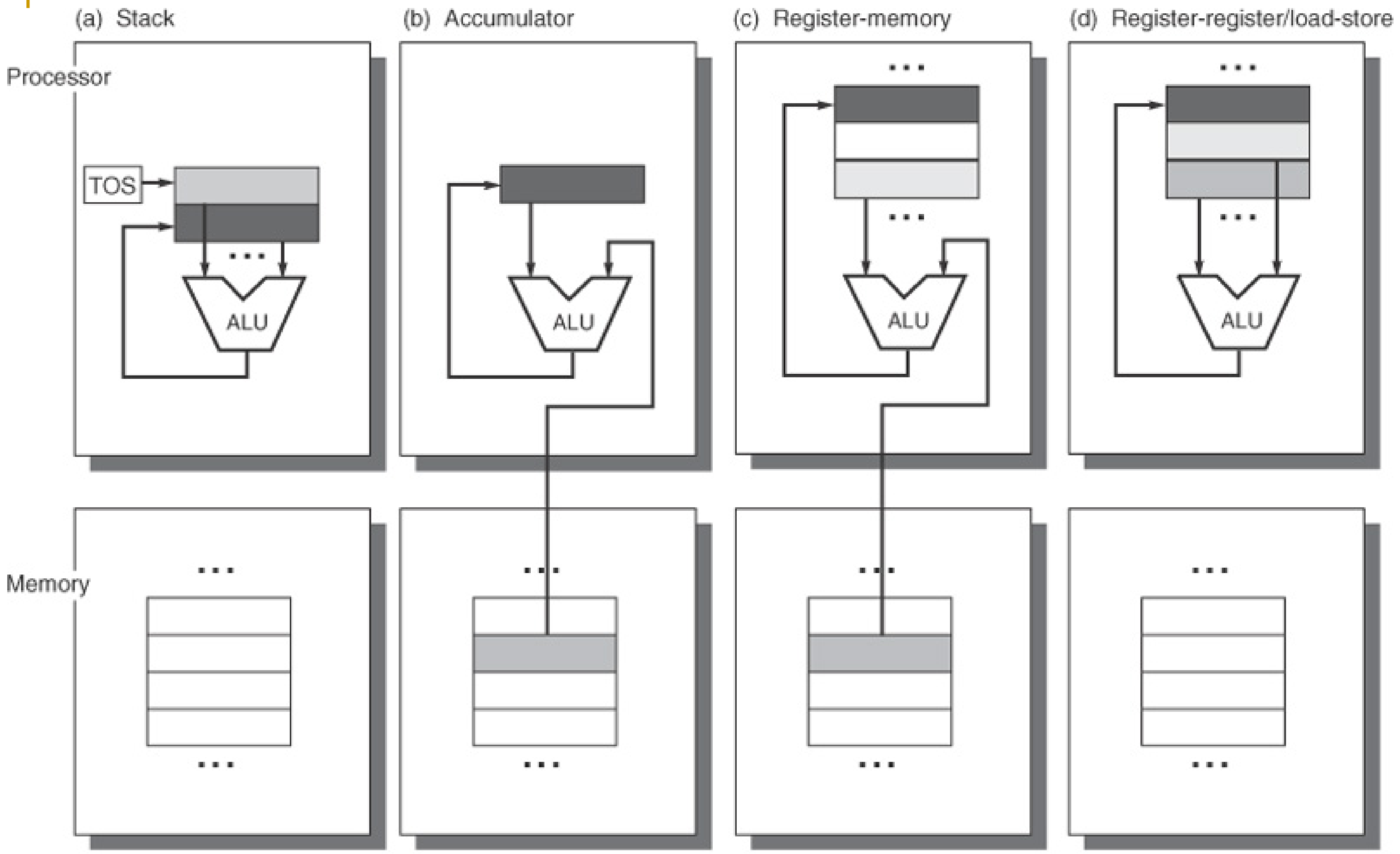
0 address	add	tos ← tos + next

General Purpose Register:

2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)

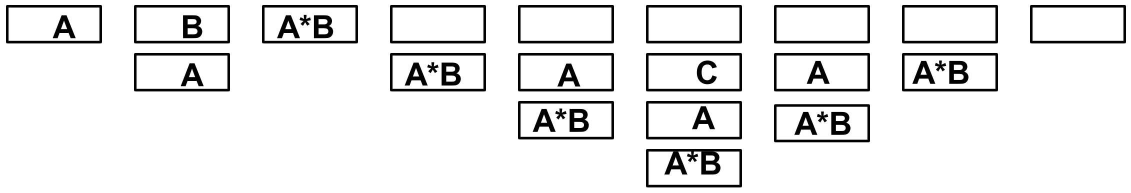
Load/Store:

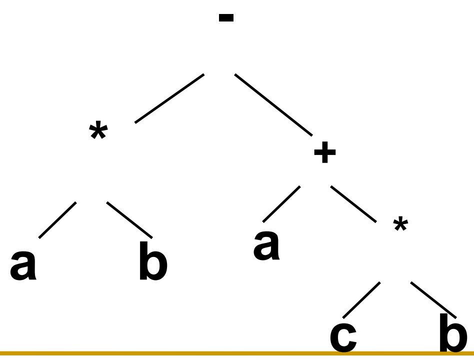
3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra


(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store
Processor
TOS
ALU
Memory
. . .

Stack Machines

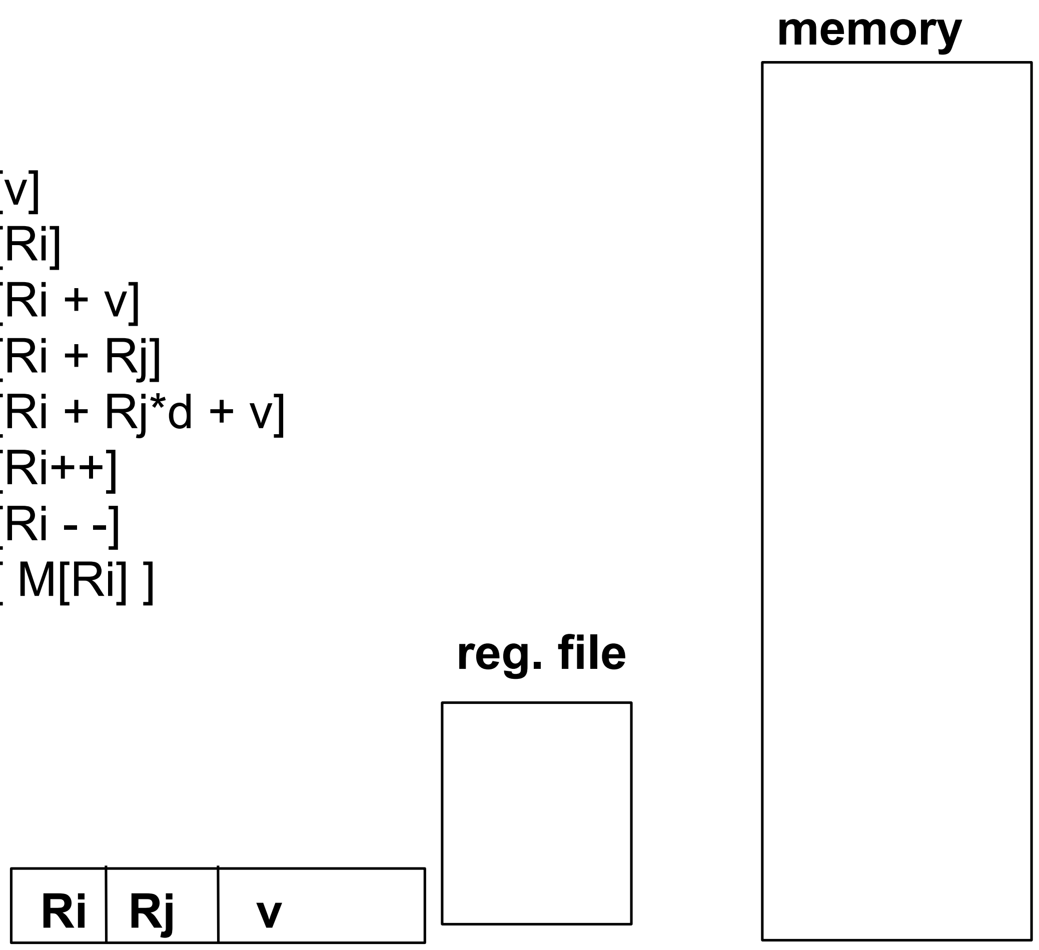
- Instruction set:
  +, -, *, /, . . .
  push A, pop A
- Example: a*b - (a+c*b)
  push a
  push b
  *
  push a
  push c
  push b
  *
  +
  -

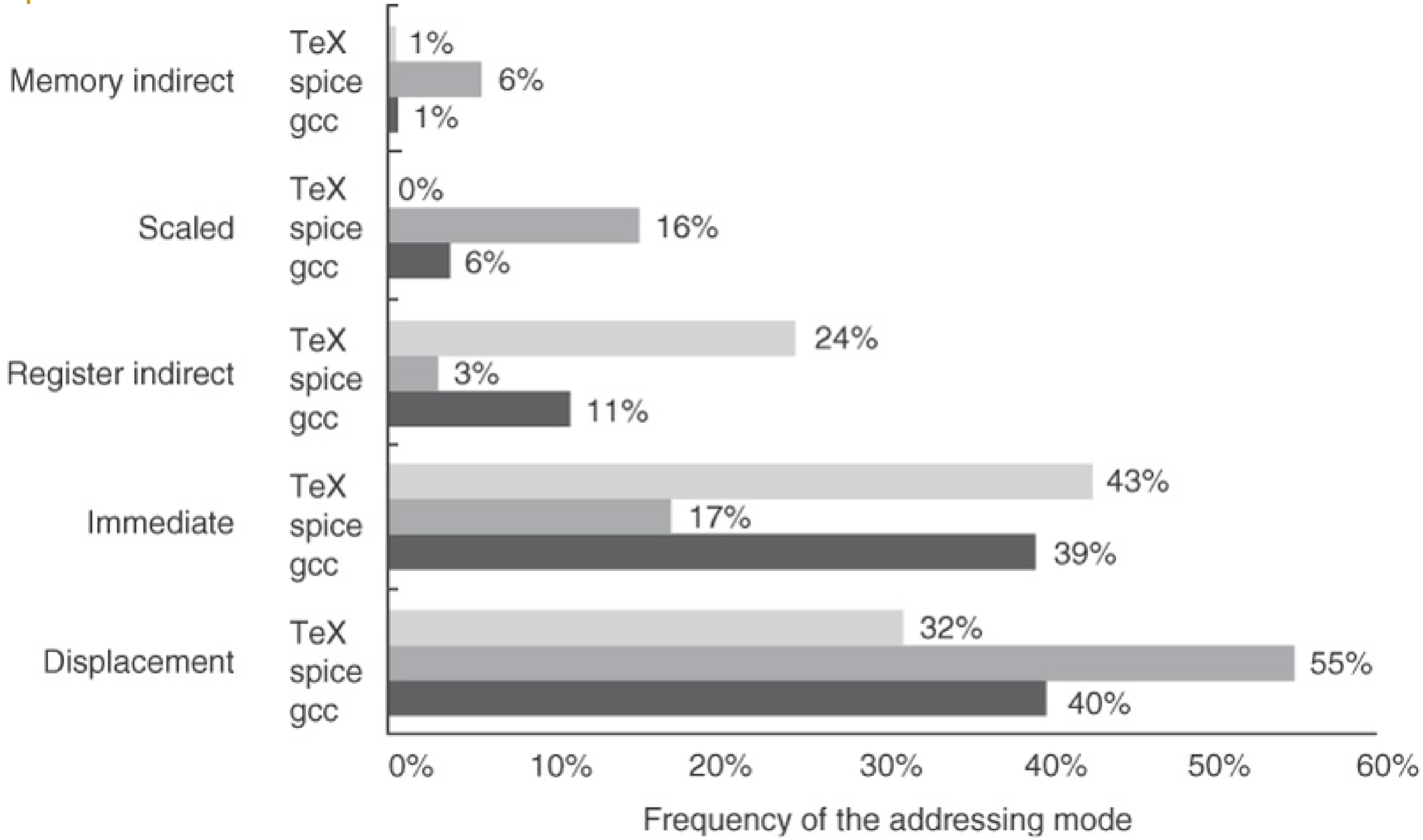




Kinds of Addressing Modes

Register direct	Ri
Immediate (literal)	v
Direct (absolute)	M[v]
Register indirect	M[Ri]
Base+Displacement	M[Ri + v]
Base+Index	M[Ri + Rj]
Scaled Index	M[Ri + Rj*d + v]
Autoincrement	M[Ri++]
Autodecrement	M[Ri - -]
Memory Indirect	M[M[Ri]]
[Indirection Chains]	

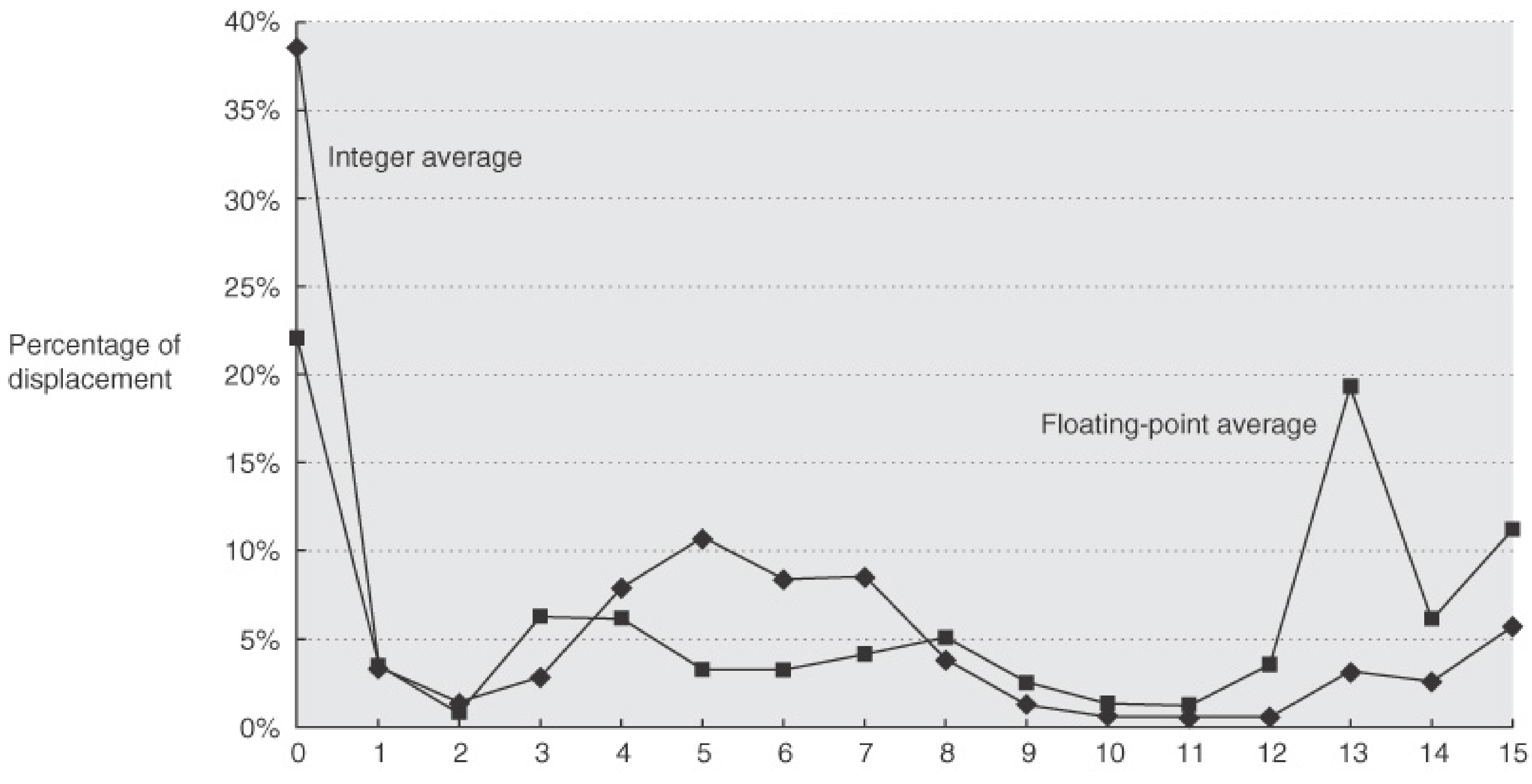


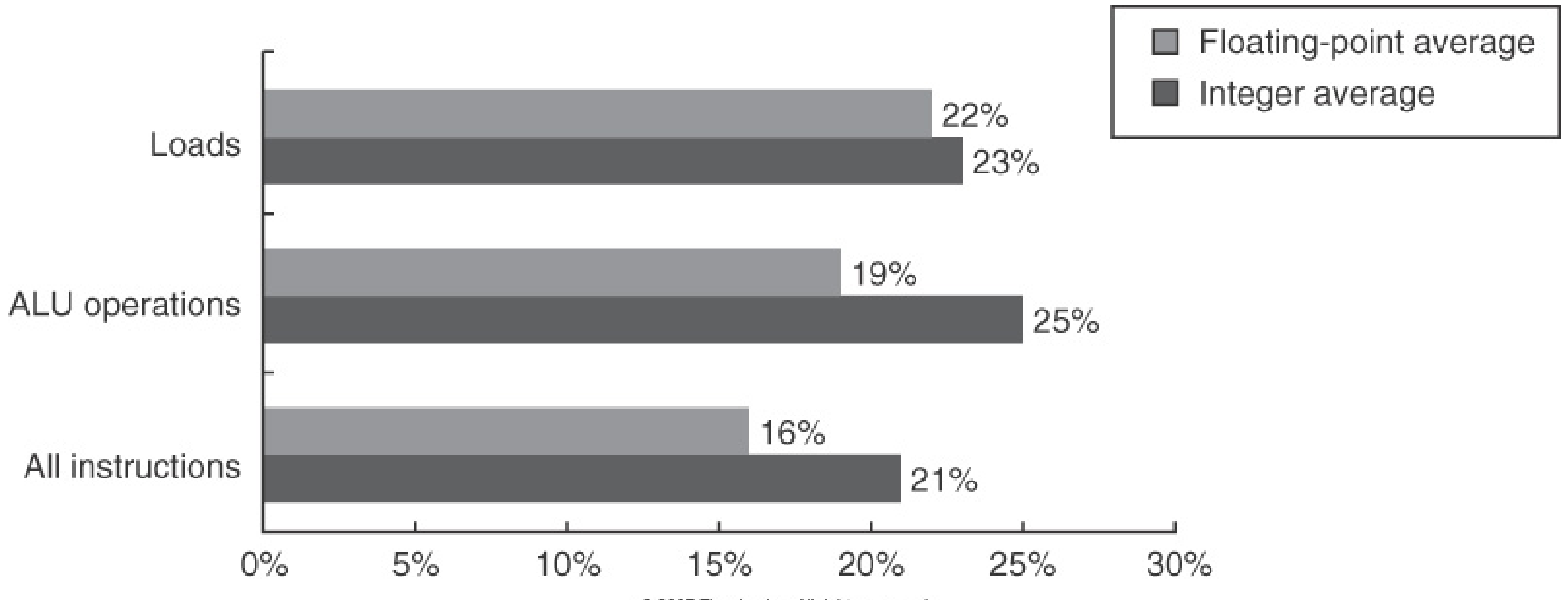

Memory indirect
Scaled
Register indirect
Immediate
Displacement
TeX
spice
gcc
1%
6%
1%
0%
16%
6%
24%
3%
11%
43%
17%
39%
32%
55%
40%
0%
10%
20%
30%
40%
50%
60%
Frequency of the addressing mode

A "Typical" RISC

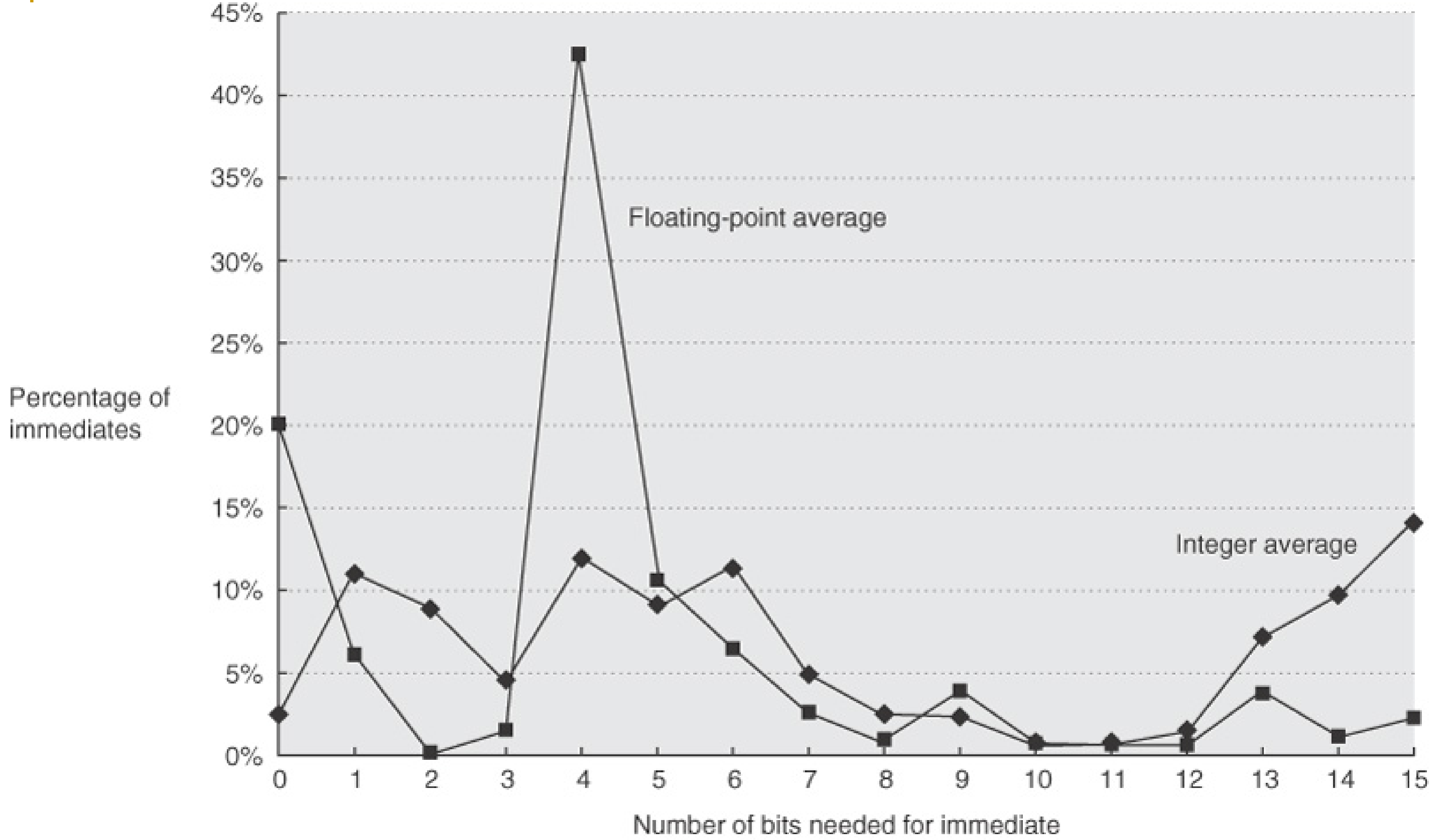
- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

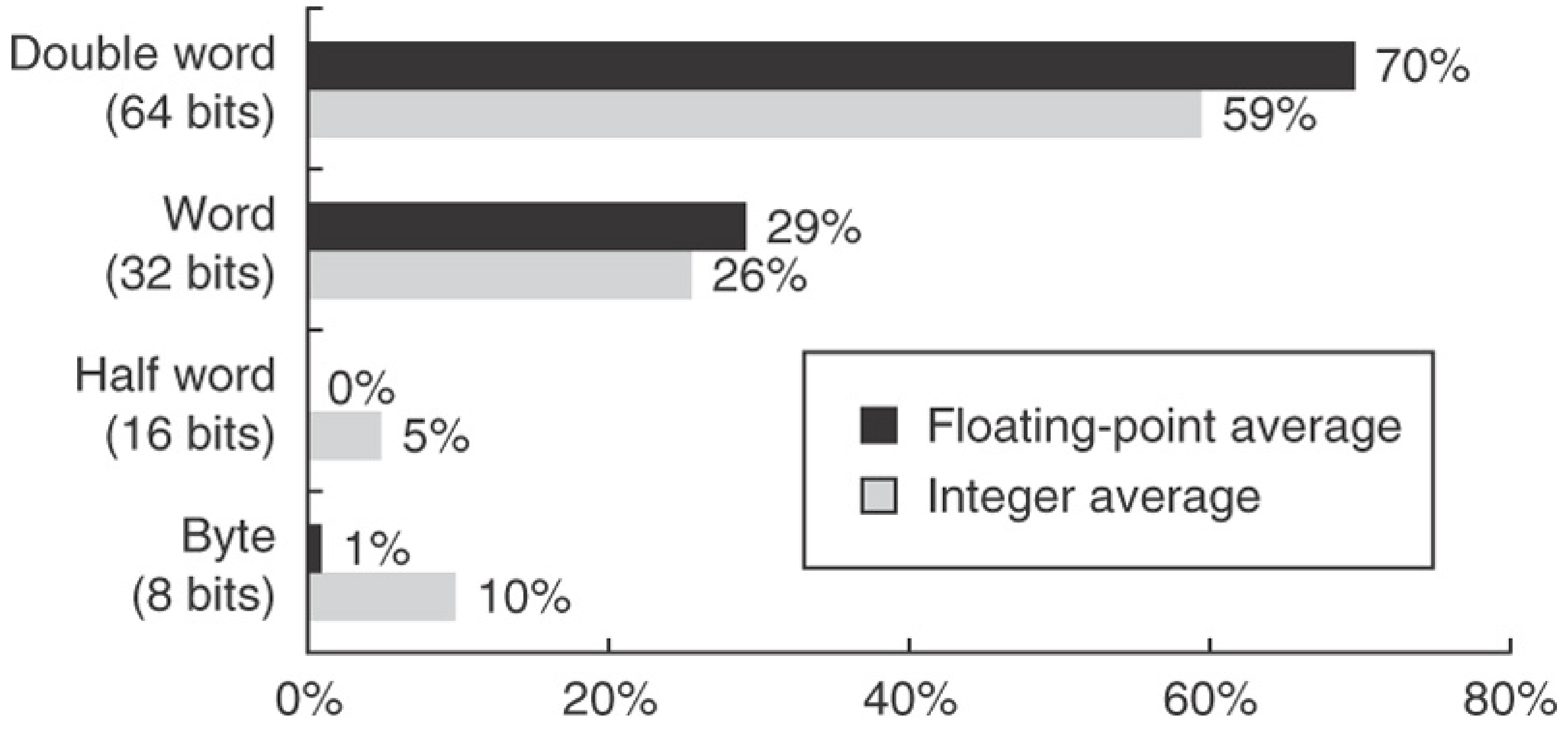
see: SPARC, MIPS, MC88100, AMD2900, i960, i860
PARisc, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3


40%
35%
30%
25%
20%
15%
10%
5%
0%
Percentage of displacement
Integer average
Floating-point average
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Number of bits of displacement

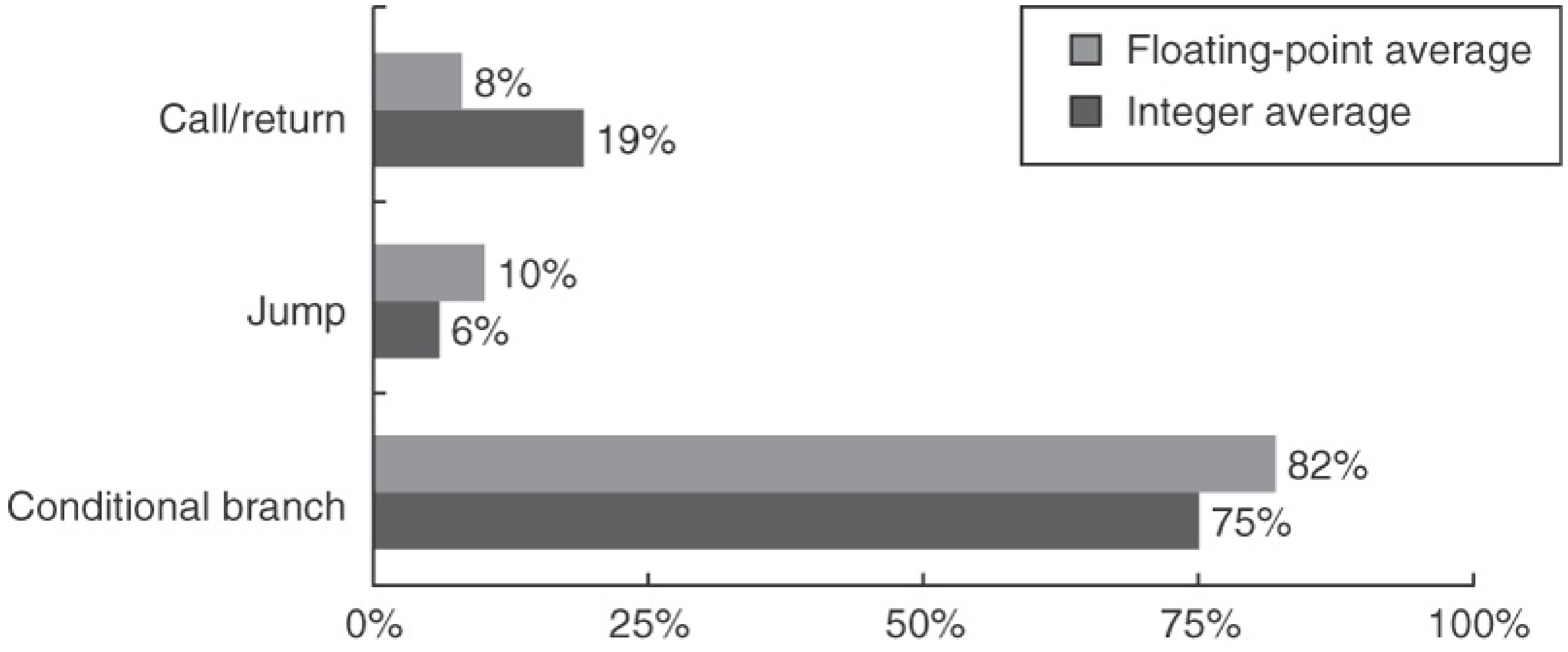


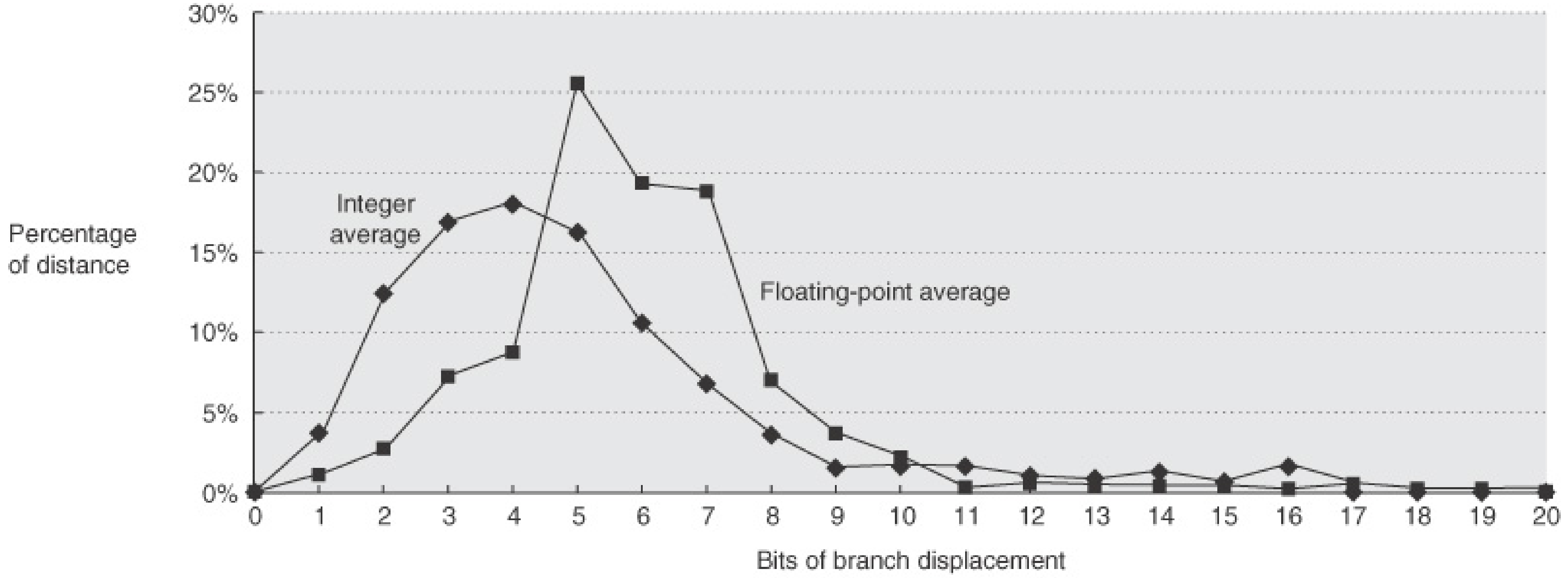
Operations that need an immediate operand

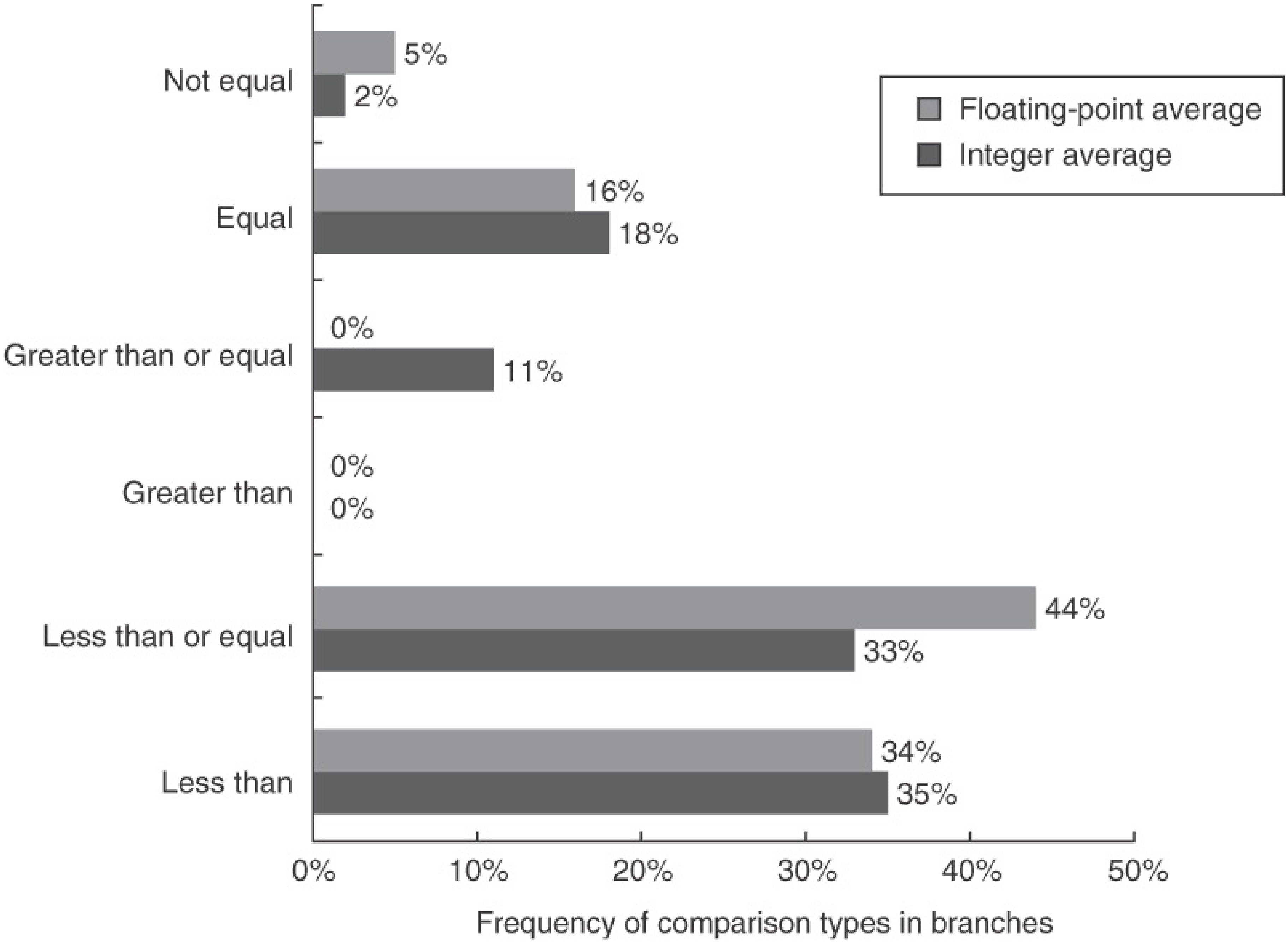

45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
Percentage of immediates
Floating-point average
Integer average
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Number of bits needed for immediate

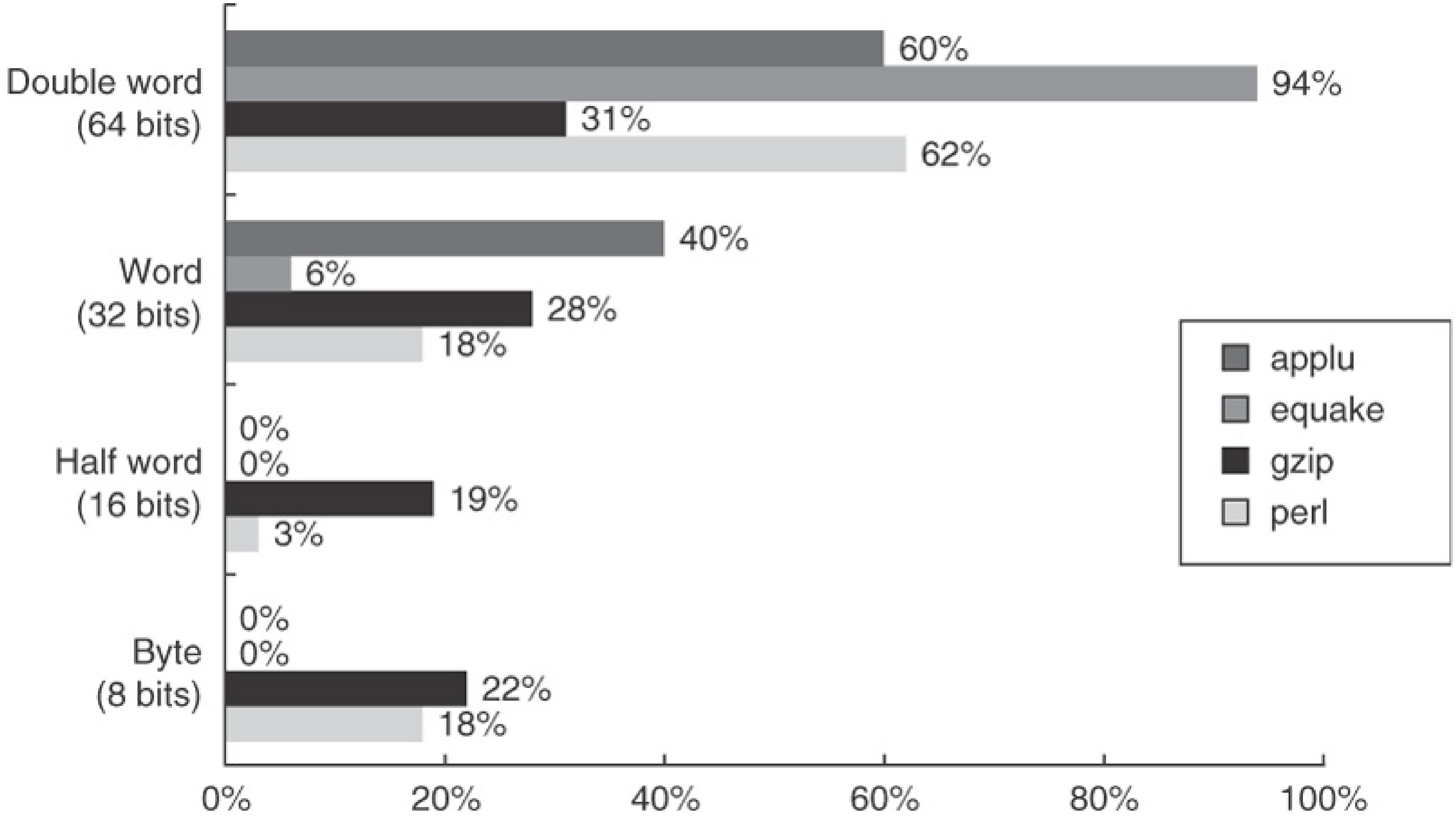


Distribution of data accesses by size for benchmark programs

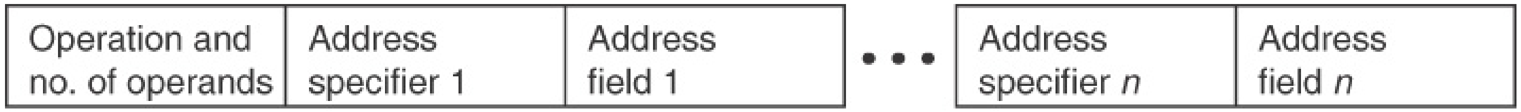

Floating-point average
Integer average
Call/return
8%
19%
Jump
10%
6%
Conditional branch
82%
75%
0%
25%
50%
75%
100%
Frequency of branch instructions


30%
25%
20%
15%
10%
5%
0%
Percentage of distance
Integer average
Floating-point average
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Bits of branch displacement

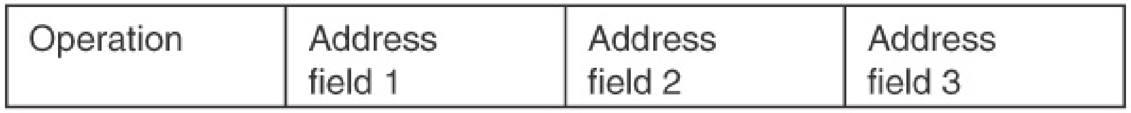

Not equal
5%
2%
Equal
16%
18%
Greater than or equal
0%
11%
Greater than
0%
0%
Less than or equal
44%
33%
Less than
34%
35%
Floating-point average
Integer average
0%
10%
20%
30%
40%
50%
Frequency of comparison types in branches


Double word (64 bits)
60%
94%
31%
62%
Word (32 bits)
40%
6%
28%
18%
Half word (16 bits)
0%
0%
19%
3%
Byte (8 bits)
0%
0%
22%
18%
applu
equake
gzip
perl
0%
20%
40%
60%
80%
100%

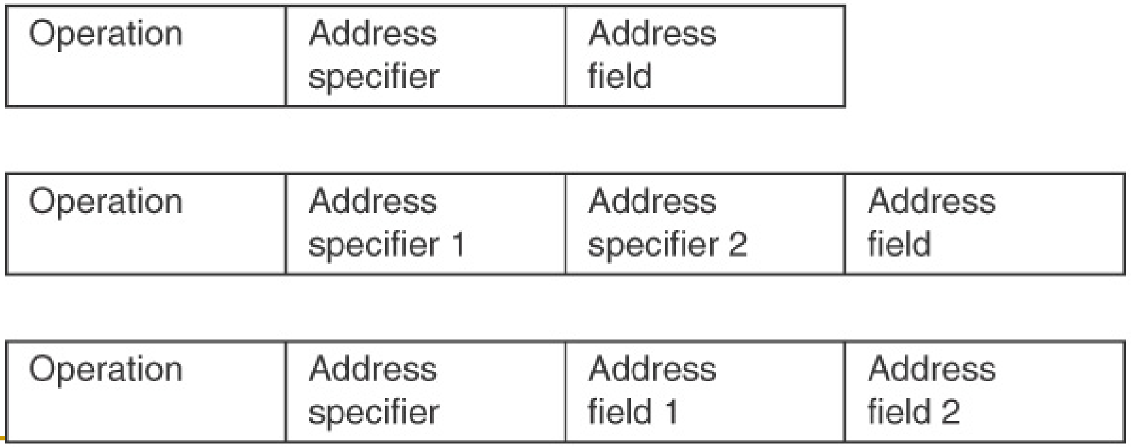
Variations of Instruction Encoding



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

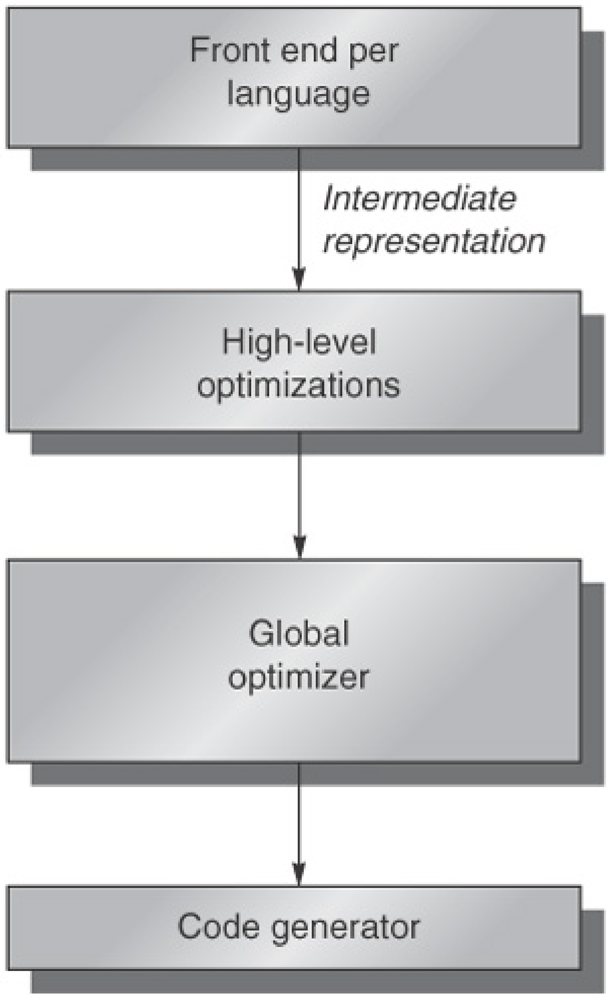


(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

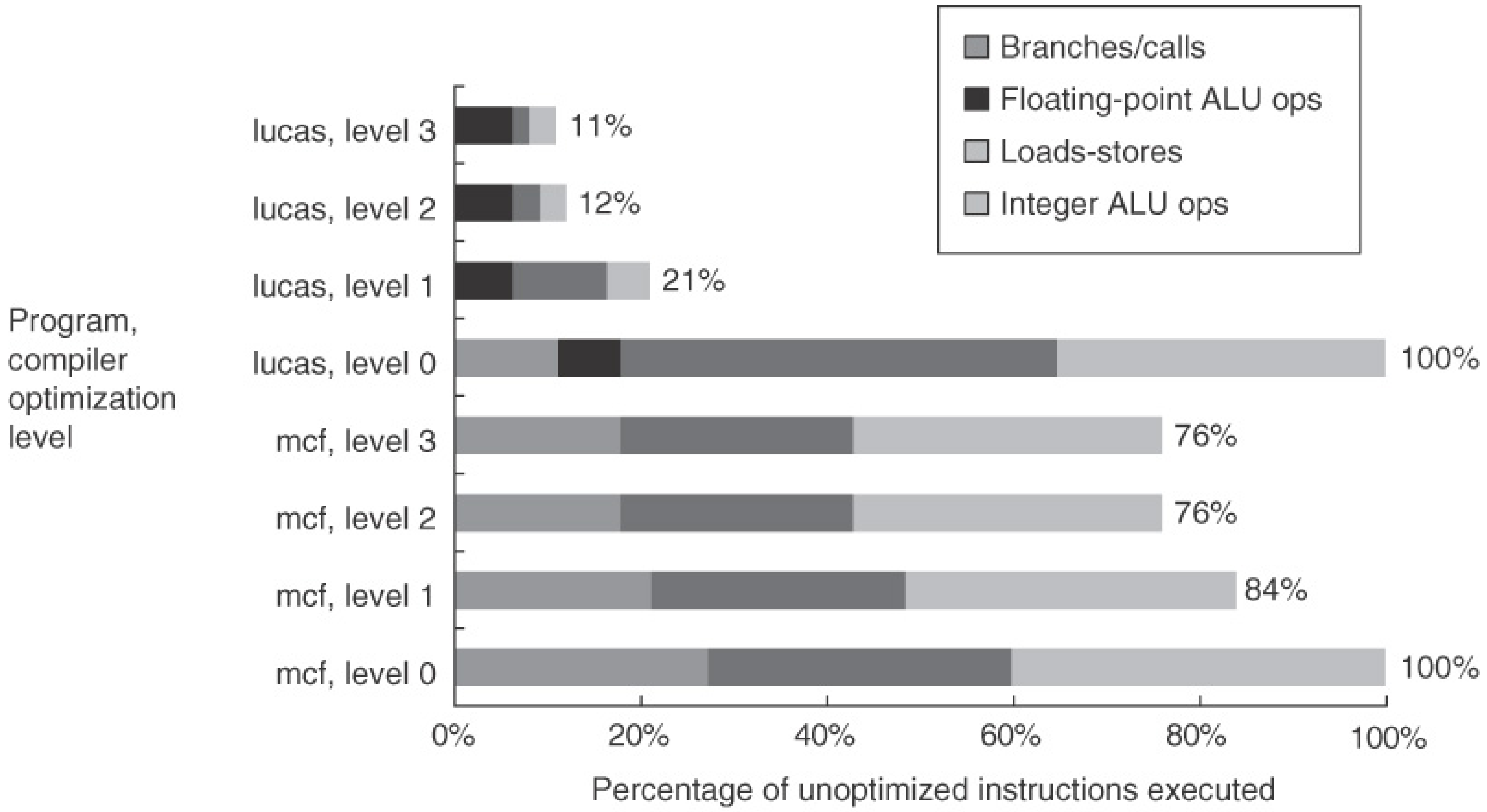


(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

State-of-the Art Compilers



Dependencies	Function
Language dependent; machine independent	Transform language to common intermediate form
Somewhat language dependent; largely machine independent	For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)	Including global and local optimizations + register allocation
Highly machine dependent; language independent	Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler


Branches/calls
Floating-point ALU ops
Loads-stores
Integer ALU ops
lucas, level 3
11%
lucas, level 2
12%
lucas, level 1
21%
lucas, level 0
100%
mcf, level 3
76%
mcf, level 2
76%
mcf, level 1
84%
mcf, level 0
100%
Program, compiler optimization level
0%
20%
40%
60%
80%
100%
Percentage of unoptimized instructions executed

Example: MIPS

Register-Register

31 26	25 21	20 16	15 11	10 6	5 0
Op	Rs1	Rs2	Rd		Opx

Register-Immediate

31 26	25 21	20 16	15 0
Op	Rs1	Rd	immediate

Branch

31 26	25 21	20 16	15 0
Op	Rs1	Rs2/Opx	immediate

Jump / Call

31 26	25 0
Op	target

Overview of MIPS

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

R	op	rs	rt	rd	shamt	funct

I	op	rs	rt	16 bit address

J	op	26 bit address

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- Instructions:

  `bne $t4,$t5,Label` Next instruction is at Label if `$t4`
  ° `$t5`

  `beq $t4,$t5,Label` Next instruction is at Label if `$t4 =`
  `$t5`

  `j Label` Next instruction is at Label

- Formats:

op	**rs**	**rt**	**16 bit address**

op	26 bit Address

– **Addresses are not 32 bits**
  — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

```
bne $t4,$t5,Label
```
Next instruction is at Label if $t4° $t5

```
beq $t4,$t5,Label
```
Next instruction is at Label if $t4=$t5

- Formats:

I	op	rs	rt	16 bit address

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

Summary of MIPS

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy

    instructions from 1 to 54 bytes long!

PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3`
    `#$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3)`
    `#$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?

- Others:
  - load multiple/store multiple
  - a special counter register "`bc Loop`"
    decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the "golden handcuffs" of compatibility

"adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

*"what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective"*

Tips for Helping the Compiler Writer

- Provide regularity
  - How does it affect the architecture?
- Provide primitives, not solutions
  - Why is it hard?
- Simplify tradeoffs among alternatives
  - How does this affect architecture?
- Provide instructions that bind the quantities known at compile time as constants.
  - How does it help with compiler/Hw interaction?

WELCOME TO HOUGHTON