EEL 6764 Principles of Computer Architecture
Instruction Set Principles

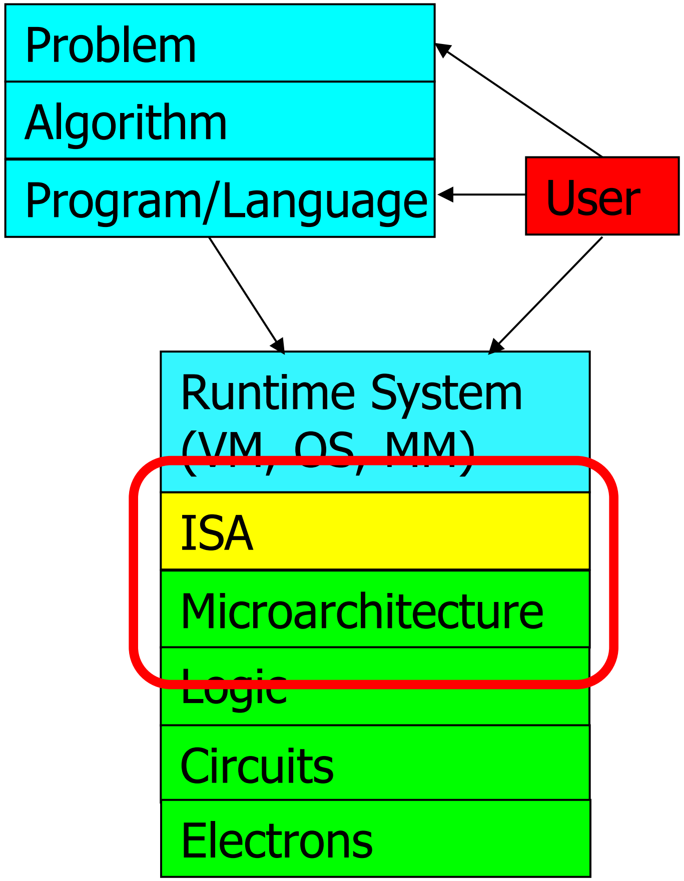
Dr Hao Zheng
Computer Sci. & Eng.
U of South Florida

Reading

- Computer Architecture: A Quantitative Approach
  - → Appendix A
- Computer Organization and Design: The Hardware/Software Interface
  - → Chapter 2

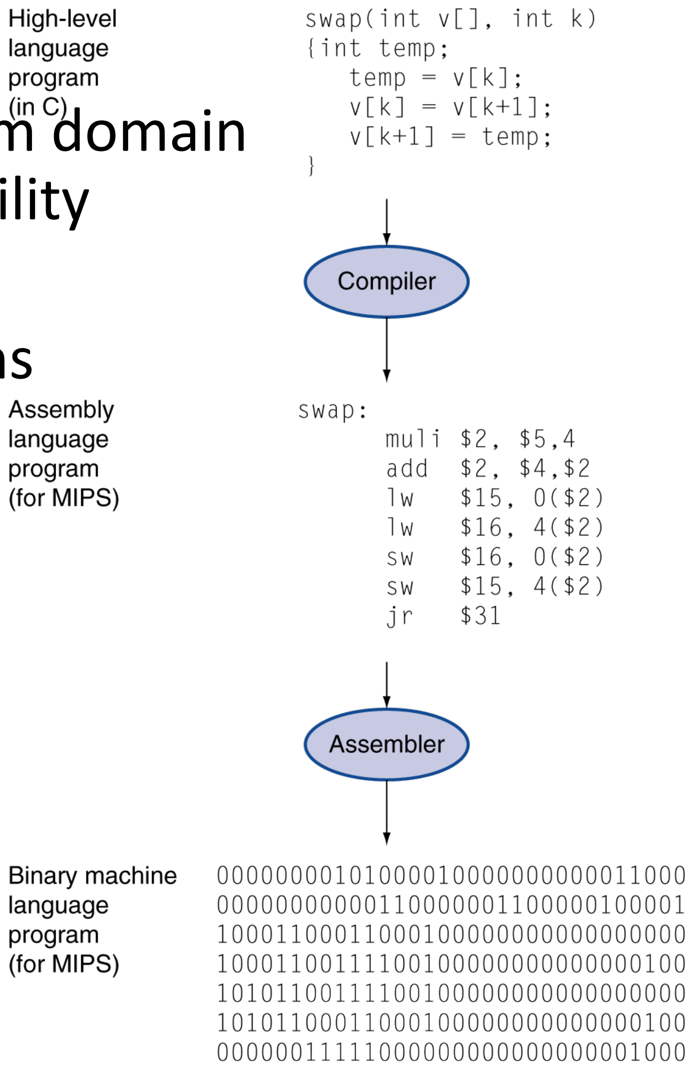
Abstractions

- Abstraction helps us deal with complexity
  - → Hide lower-level detail
- Instruction set architecture (ISA)
  - → The hardware/software interface
  - → Defines storage, operations, etc
- Implementation
  - → The details underlying the interface
  - → An ISA can have multiple implementations



Levels of Program Code

- High-level language
  - → Level of abstraction closer to problem domain
  - → Provides for productivity and portability
- Assembly language
  - → Textual representation of instructions
- Hardware representation
  - → Machine code - Binary digits (bits)
  - → Encoded instructions and data
- Compiler
  - → Translate HL prgm to assembly
- Assembler
  - → Translate assembly to machine code


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

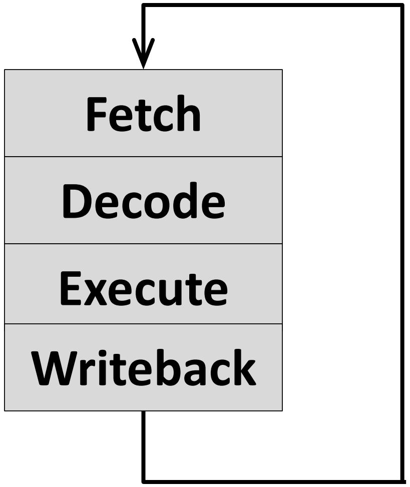
```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Program Execution Model

- A computer is just a FSM
  - → States stored in registers, memory, PC, etc
  - → States changed by instruction execution
- An instruction is executed in
  - → **Fetch** an instruction into CPU from memory
  - → **Decode** it to generate control signals
  - → **Execute** it (add, mult, etc)
  - → **Write back** output to reg or memory



- *Programs* and *data* coexist in memory
  - → How to distinguish program from data?

What Makes a Good ISA?

- **Programmability**
  - → Who does assembly programming these days?
- **Performance/Implementability**
  - → Easy to design high-performance implementations?
  - → Easy to design low-power implementations?
  - → Easy to design low-cost implementations?
- **Compatibility**
  - → Easy to maintain as languages, programs evolve
  - → x86 (IA32) generations: 8086, 286, 386, 486, Pentium, Pentium-II, Pentium-III, Pentium4, Core2, Core i7, ...

Performance

- **Execution time = IC * CPI * cycle time**
- **IC:** instructions executed to finish program
  - → Determined by program, compiler, ISA
- **CPI:** number of cycles needed for each instruction
  - → Determined by compiler, ISA, u-architecture
- **Cycle time:** inverse of clock frequency
  - → Determined by u-architecture & technology
- Ideally optimize all three
  - → Their optimizations often against each other
  - → Compiler plays a significant role

Instruction Granularity

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - → Big heavyweight instructions (lots of work per instruction)
  - + Low “inst/program” (IC)
  - – Higher “cycles/inst” and “seconds/cycle” (CPI)
    - ➤ We have the technology to get around this problem
- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - → Minimalist approach to an ISA: simple inst only
  - + Low “cycles/inst” and “seconds/cycle”
  - – Higher “inst/program”, but hopefully not as much
    - ➤ Rely on compiler optimizations

Classifying Architectures

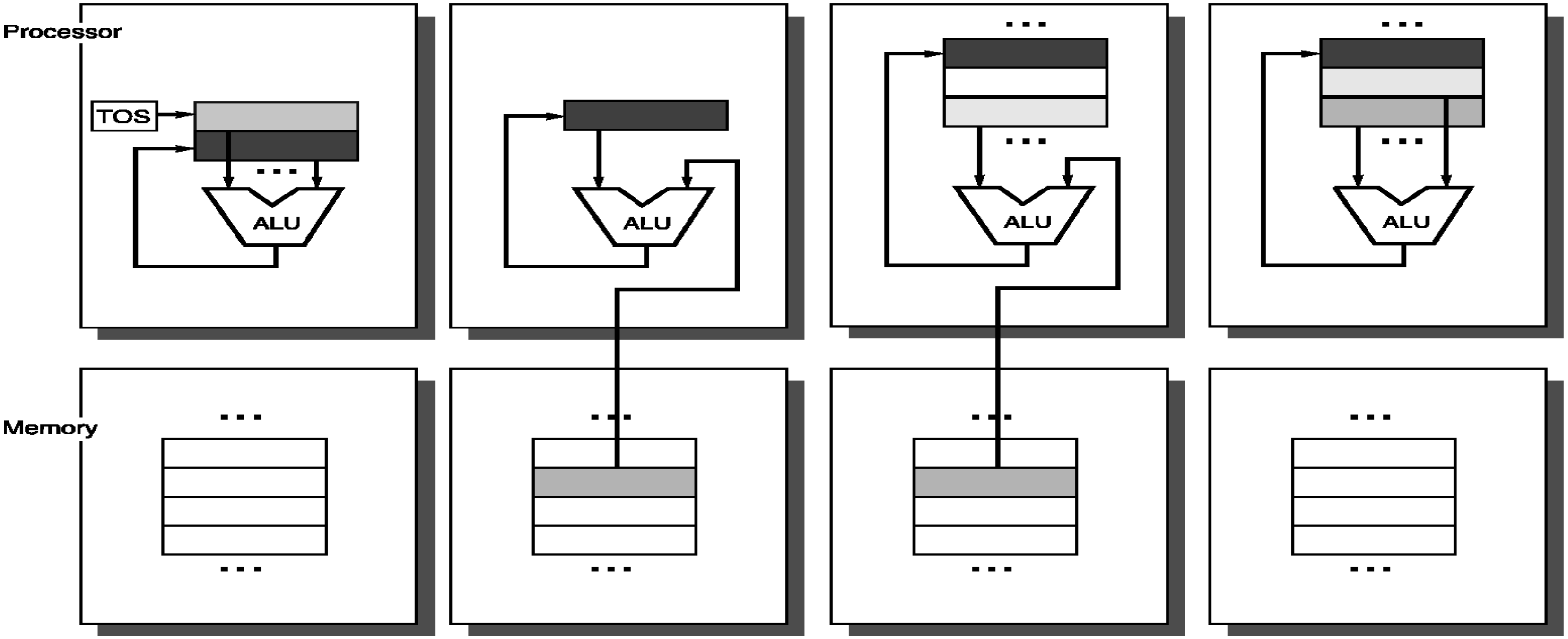
- One important classification scheme is by the type of addressing modes supported.

- Stack architecture: Operands implicitly on top of a stack. (Early machines, Intel floating-point.)
- Accumulator architecture: One operand is implicitly an accumulator (a special register).
  - → early machines
- General-purpose register arch.: Operands may be any of a large (typically 10s-100s) # of registers.
  - → Register-memory architectures: One op may be memory.
  - → Load-store architectures: All ops are registers, except in special load and store instructions.

Illustrating Architecture Types

Assembly for **C:=A+B**:

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R1,B`	`Load R2,B`
`Add`	`Store C`	`Store C,R1`	`Add R3,R1,R2`
`Pop C`			`Store C,R3`



Number of Registers

- Registers have advantages
  - → faster than memory, good for compiler optimization, hold variables
- have as many as possible?
  - → **No**
- One reason that registers are faster: there are
  - → **fewer of them** – small is fast (hardware truism)
- Another: they are **directly addressed**
  - → More registers, means more bits per register in instruction
  - → Thus, fewer registers per instruction or larger instructions
- More registers means **more saving/restoring**
  - → Across function calls, traps, and context switches
- Trend toward more registers:
  - → 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)

Number of Operands

- A further classification is by the max. number of operands, and # of operands that can be memory: e.g.,
  - → 2-operand (e.g. a += b)

➢ src/dest(reg), src(reg)	
➢ src/dest(reg), src(mem)	IBM 360, x86, 68k
➢ src/dest(mem), src(mem)	VAX

  - → 3-operand (e.g. a = b+c)

➢ dest(reg), src1(reg), src2(reg)	MIPS, PPC, SPARC, &c.
➢ dest(reg), src1(reg), src2(mem)	
➢ dest(mem), src1(mem), src2(mem)	VAX

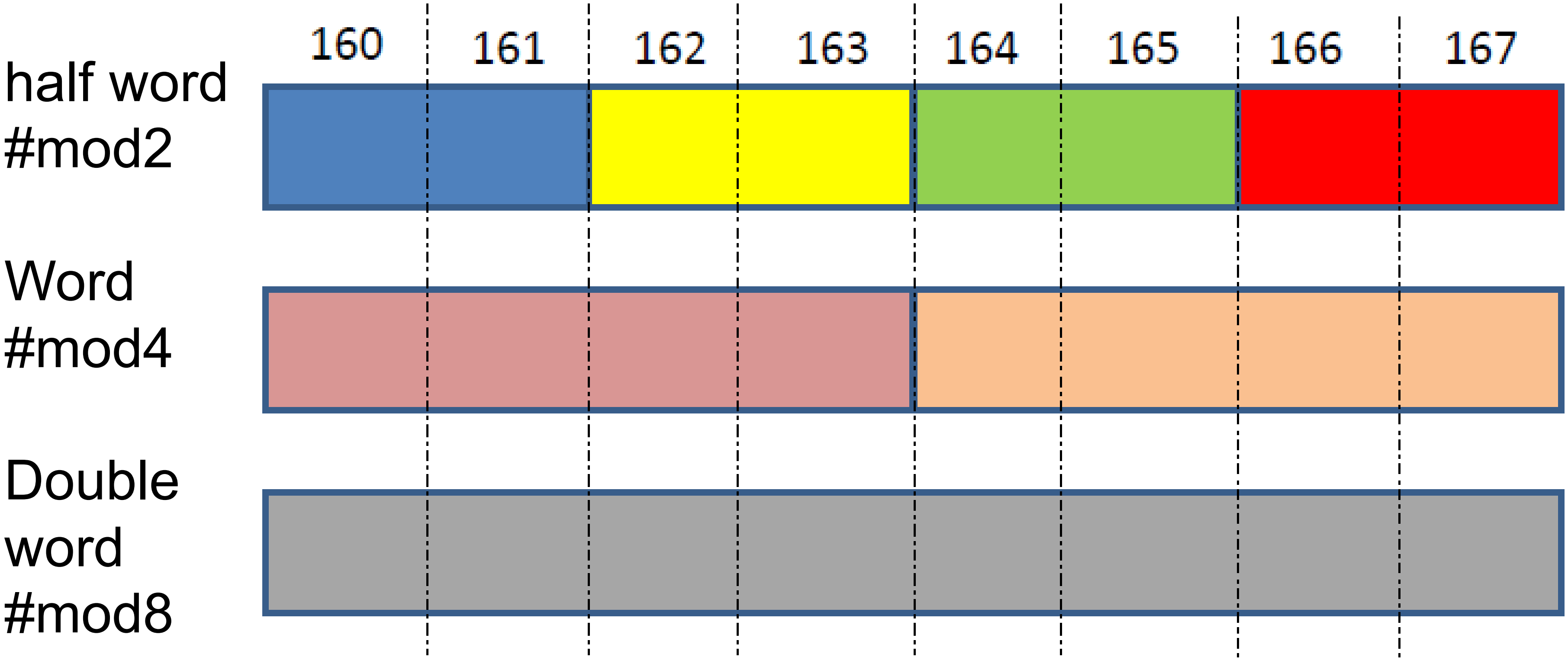
- Classifications
  - → register-register (load-store)
  - → register-memory
  - → memory-memory

Memory Addressing

- Byte Addressing
  - → Each byte has a unique address
- Other addressing units
  - → Half-word: 16-bit (or 2 bytes)
  - → Word: 32-bit (or 4 bytes)
  - → Double word : 64-bit (or 8 bytes)
  - → Quad word: 128-bit (or 16 bytes)
- Two issues
  - → **Alignment** specifies whether there are any boundaries for word addressing
  - → **Byte order** (Big Endian vs. Little Endian)
    - ➤ specifies how multiple bytes within a word are mapped to memory addresses

Memory Addressing

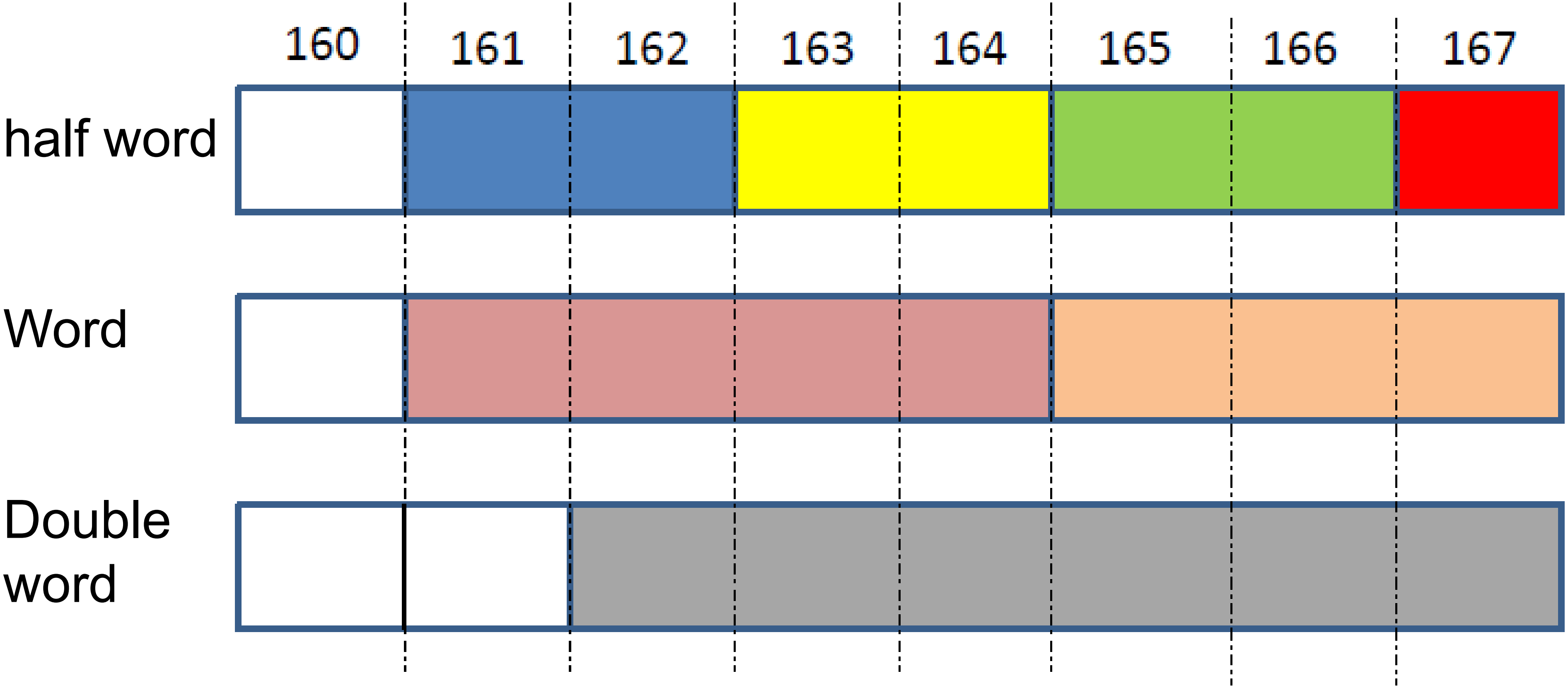
- Alignment
  - → Must half word, words, double words begin mod 2, mod 4, mod 8 boundaries



Aligned if *Addr* ***mod*** *size = 0*

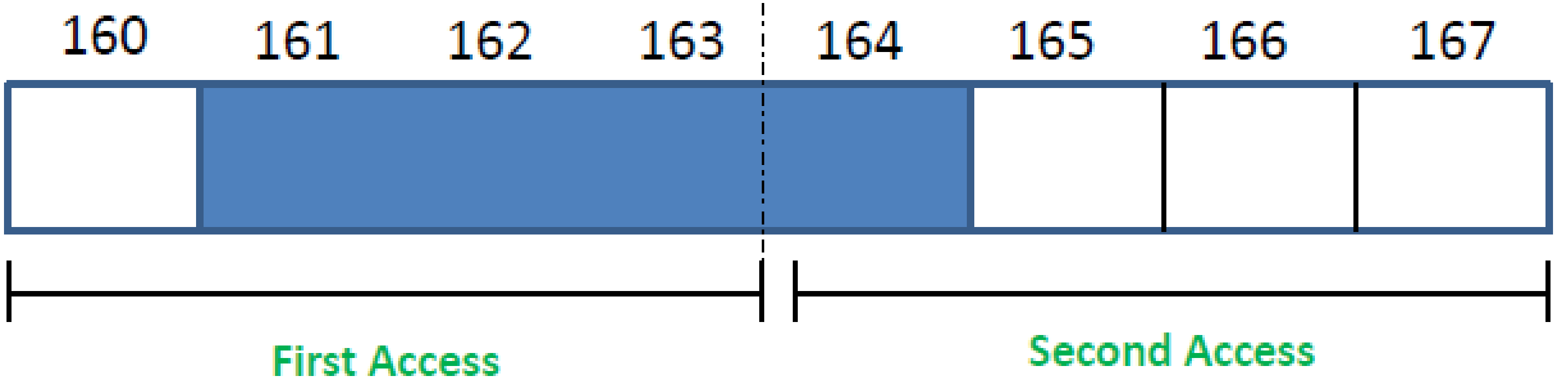
Memory Addressing

- Alignment
  - → Or there no alignment restrictions



Memory Addressing

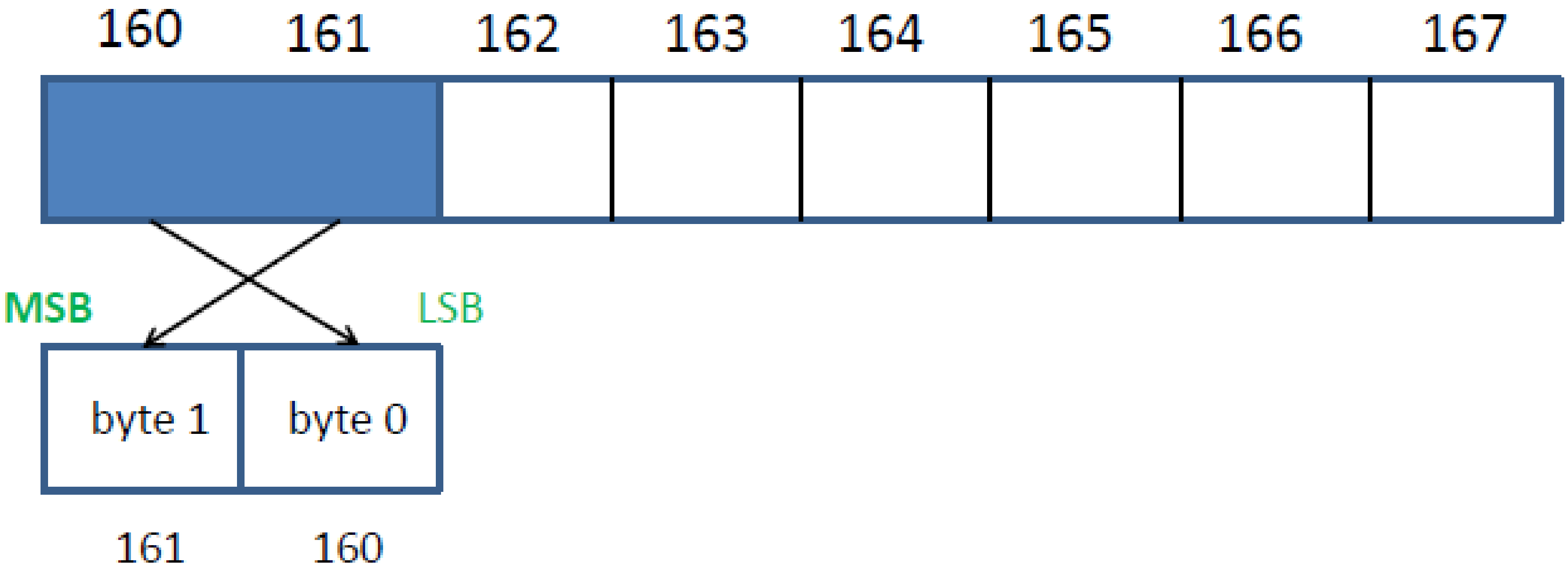
- Non-aligned memory references may cause multiple memory accesses



- Consider a system in which memory reads return 4 bytes and a reference to a word spans a 4-byte boundary: two memory accesses are required
- Complicates memory and cache controller design
- Assemblers typically force alignment for efficiency

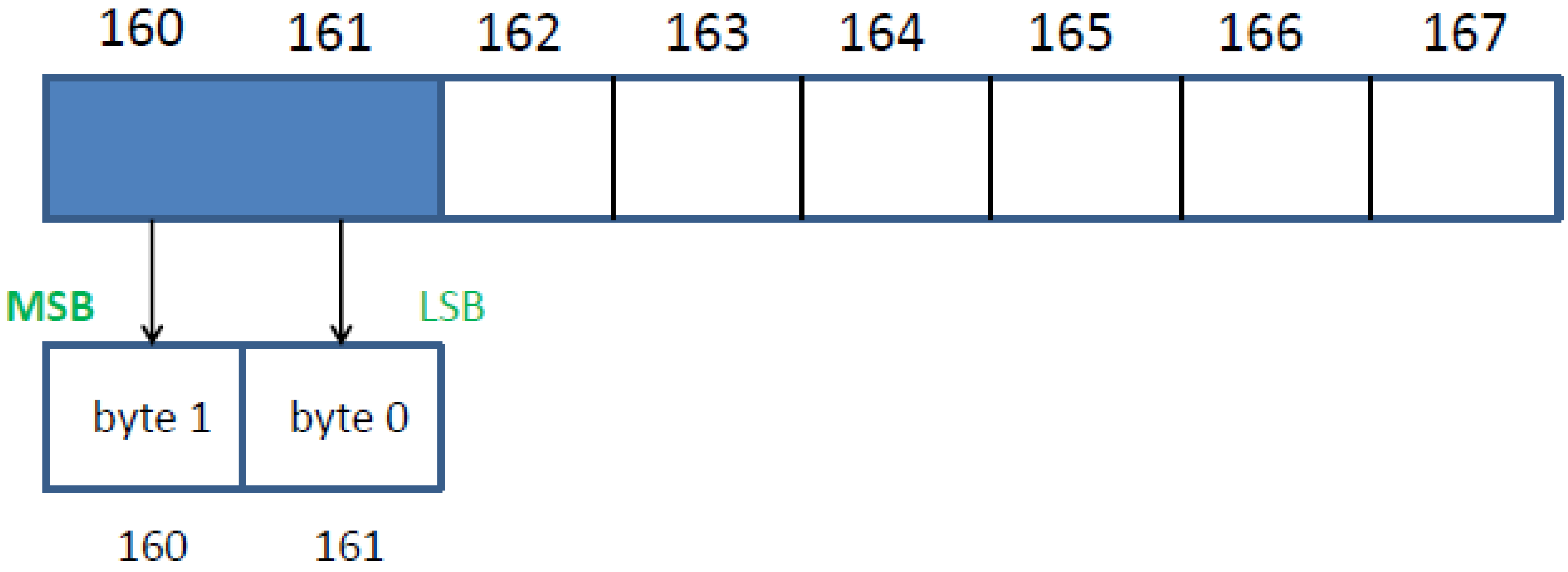
Byte Ordering – Little Endian

- The least significant byte within a word (or half word or double word) is stored in the smallest address



Byte Ordering – Big Endian

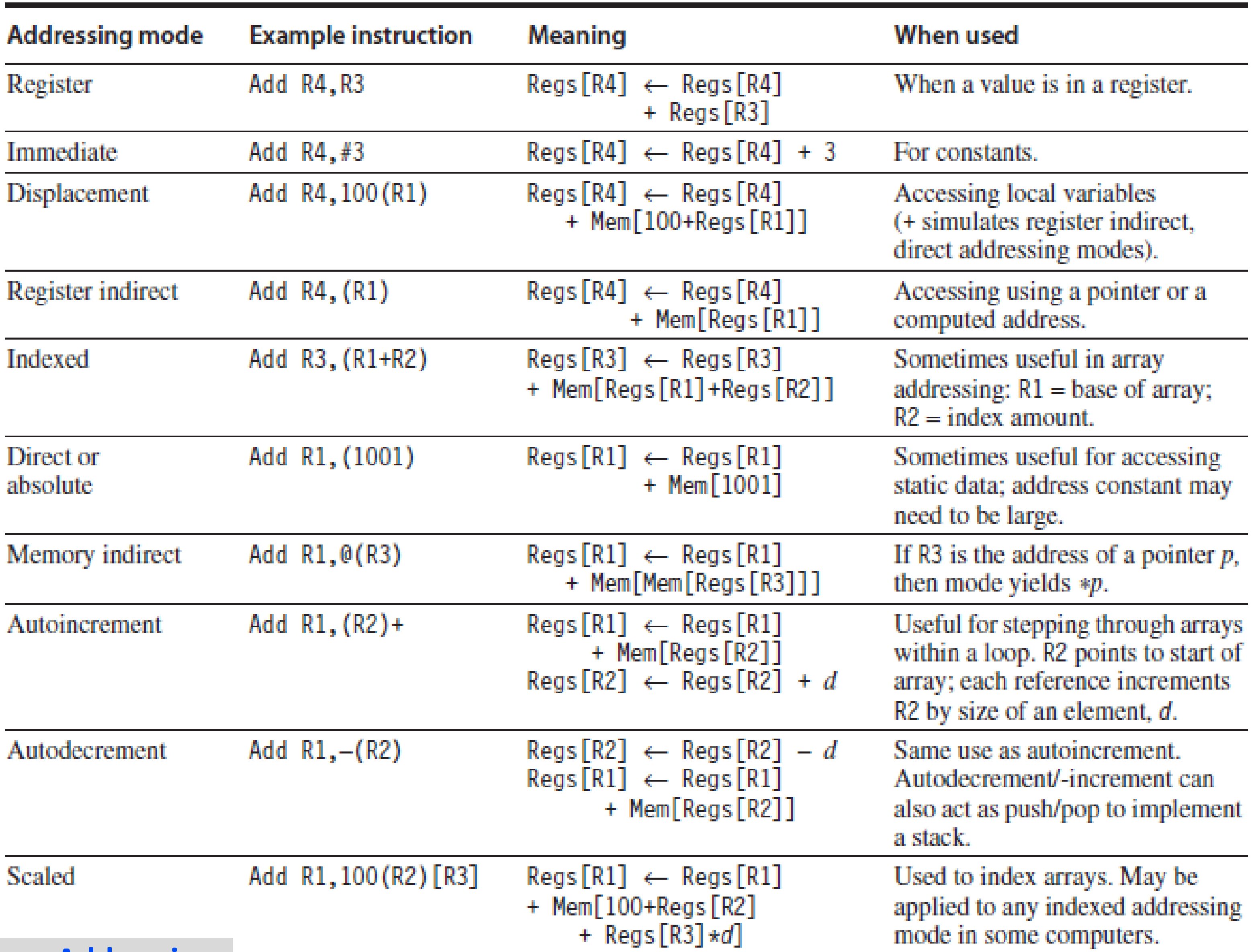
- The most significant byte within a word (or half word or double word) is stored in the smallest address



Byte Order in Real Systems

- Big Endian: Motorola 68000, Sun Sparc, PDP-11
- Little Endian: VAX, Intel IA32
- Configurable: MIPS, ARM
- No difference within a single machine

Addressing Modes – How to Find Operands



Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4] ← Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4,#3	Regs[R4] ← Regs[R4] + 3	For constants.
Displacement	Add R4,100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1+R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer *p*, then mode yields **p*.

Autoincrement	Add R1,(R2)+	Regs[R1] ← Regs[R1]
		+ Mem[Regs[R2]]
		Regs[R2] ← Regs[R2] + *d*
			Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.

Autodecrement	Add R1,–(R2)	Regs[R2] ← Regs[R2] – *d*
		Regs[R1] ← Regs[R1]
		+ Mem[Regs[R2]]
			Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.

Scaled	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]**d*]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Addressing Modes

- Addressing modes can reduce instruction counts but at a cost of added CPU design complexity and/or increase average CPI
- Example (usage of auto-increment mode):
  - → With auto-increment mode:

    ```
    Add R1, (R2)+
    ```

  - → Without auto-increment mode

    ```
    Add R1, (R2)
    Add R2, #1
    ```

- Example (usage of displacement mode):
  - → With displacement mode:

    ```
    Add R4, 100(R1)
    ```

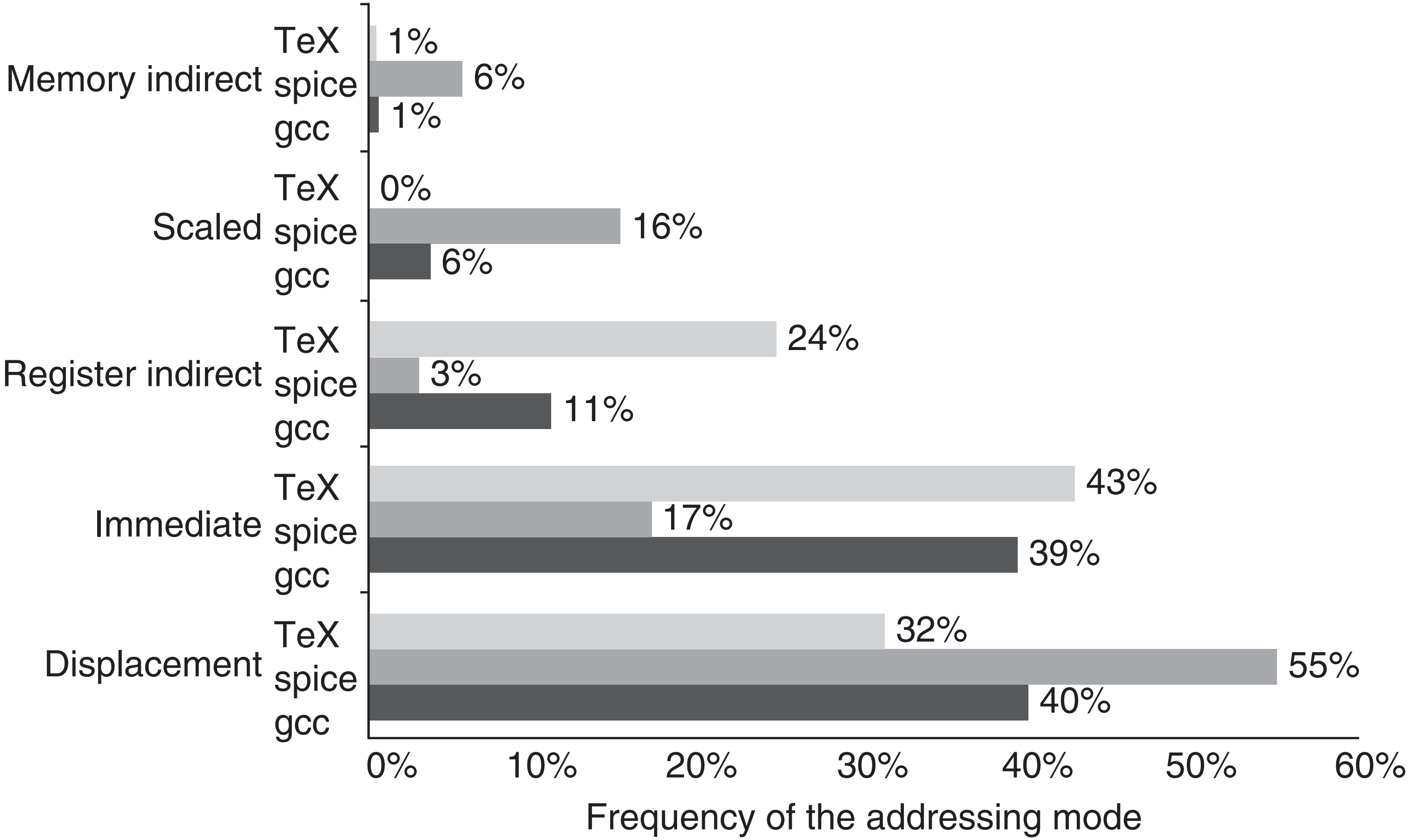
  - → Without displacement mode

    ```
    Add R1, #100
    Add R4, (R1)
    Sub R1, #100
    ```

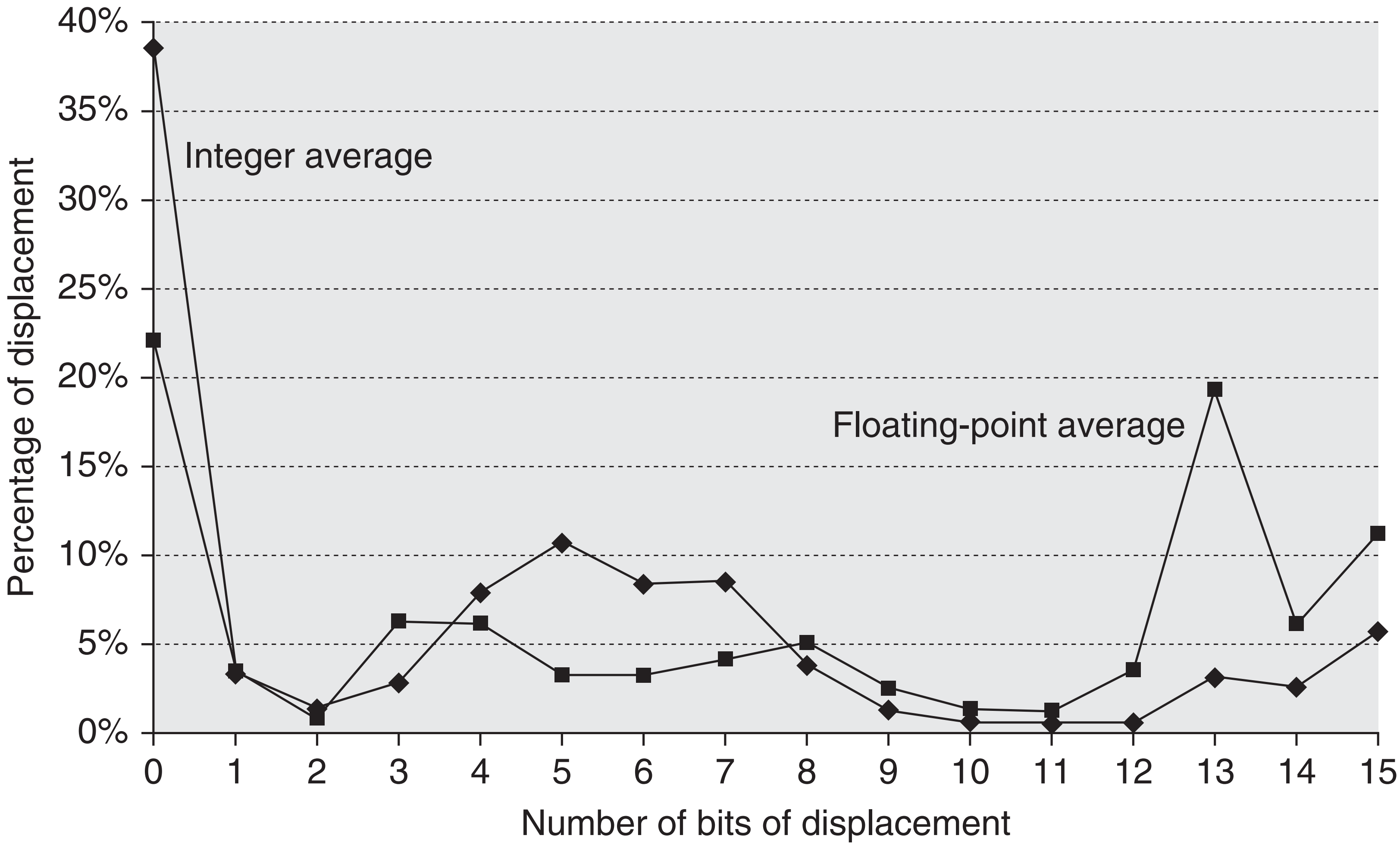
Which Addressing Modes to Support

- Support frequently used modes

  → *Make common case fast!*



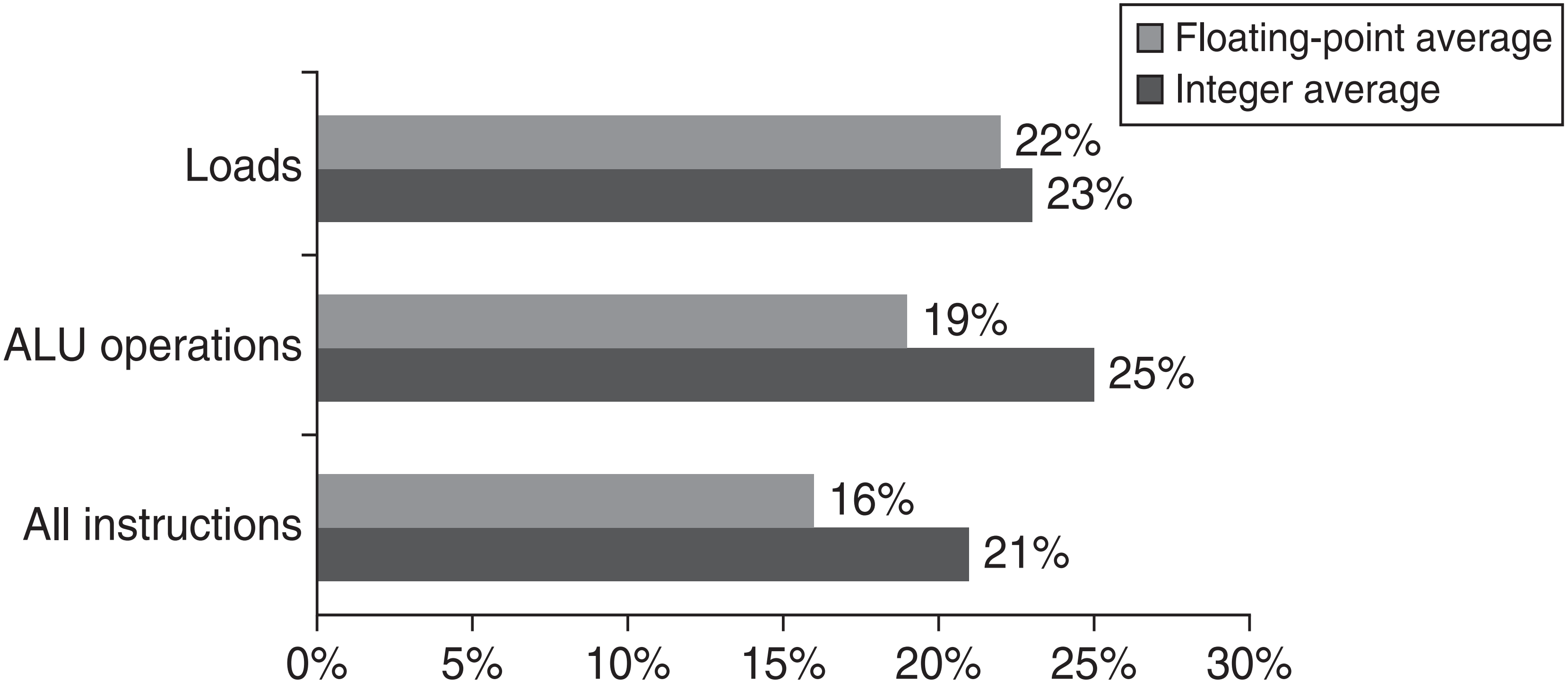
Displacement Value Distribution



add R4 100(R1) – 16 bits to be sufficient

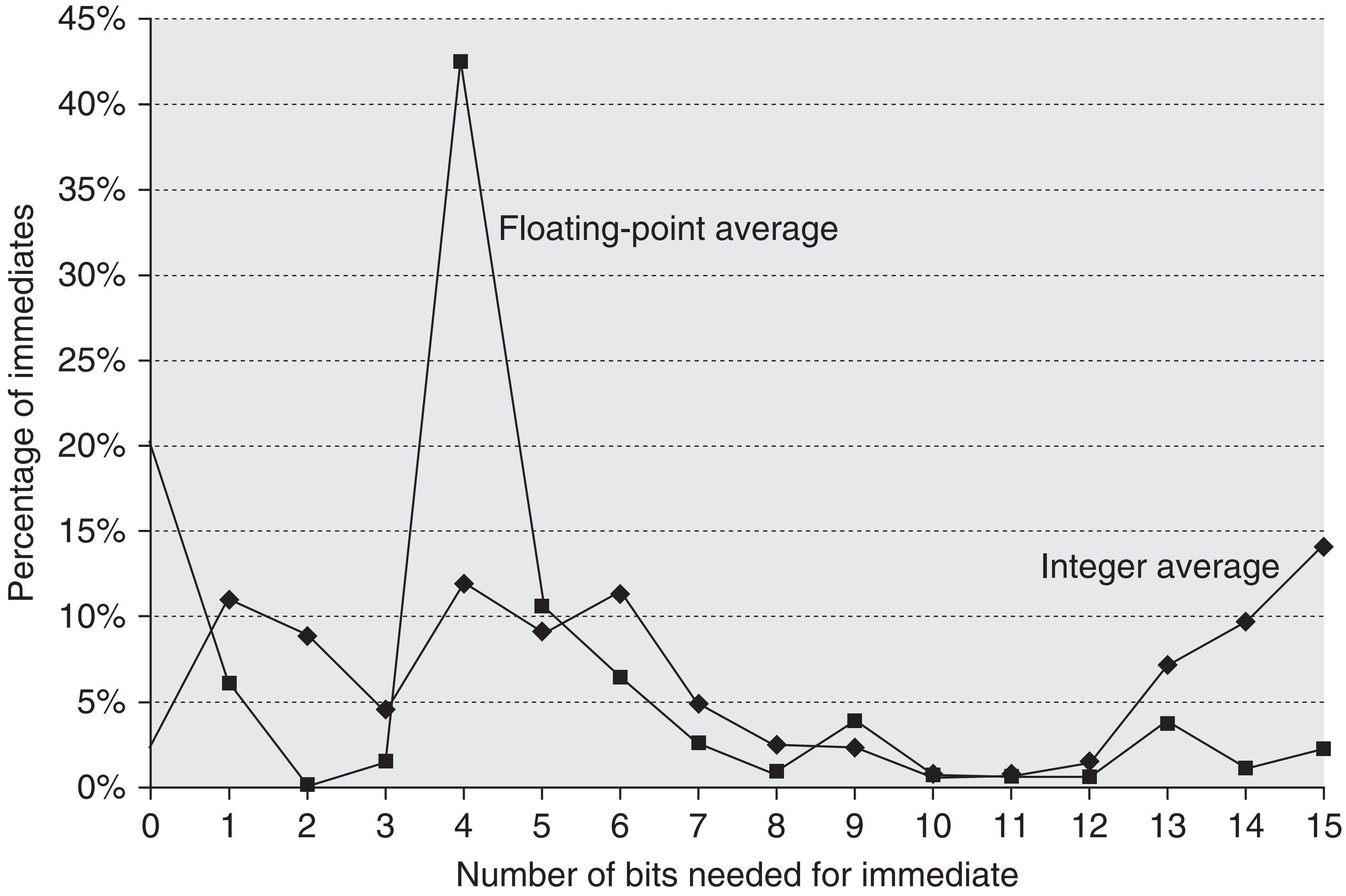
SPEC CPU 2000 on Alpha

Popularity of Immediate Operands



add R4 **#3**

Distribution of Immediate Values



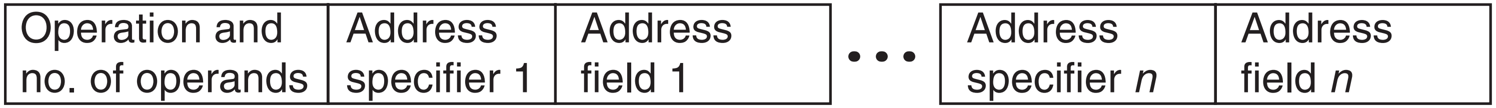
add R4 **#3** – 16 bits to be sufficient

SPEC CPU 2000 on Alpha

Other Issues

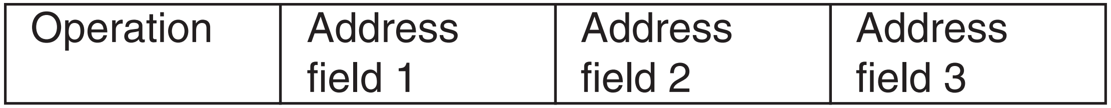
- How to specify type of size of operands (A.4)
  - → Mainly specified in opcode – no separate tags for operands
- Operations to support (A.5)
  - → simple instructions are used the most
- Control flow instructions (A.6)
  - → Branch, call/return more popular than jump
  - → Target address is typically PC-relative & register indirect
  - → Address displacements are usually <= 12 bits
  - → How to implement conditions for branches

Instruction Encoding



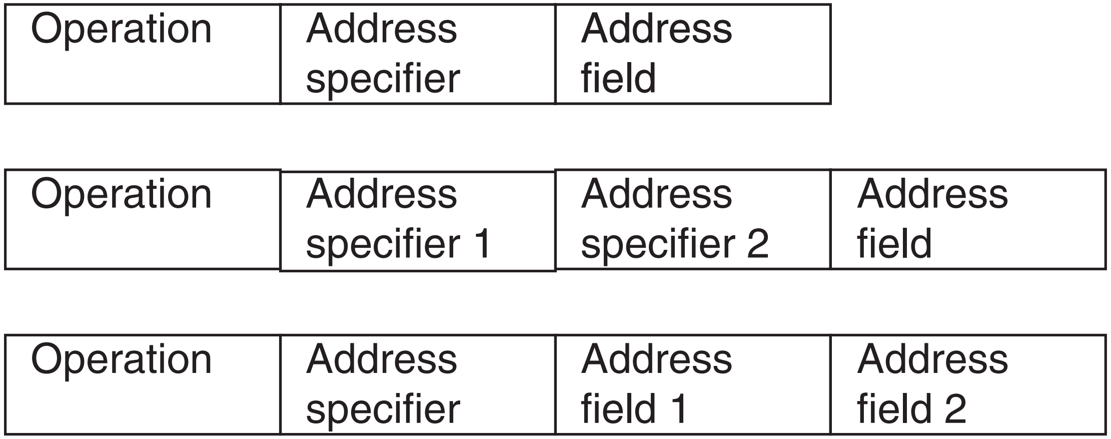
Operation and no. of operands | Address specifier 1 | Address field 1 | • • • | Address specifier *n* | Address field *n*

(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)

Instruction Encoding

- Affects code size and implementation
- OpCode – Operation Code
  - → The instruction (e.g., “add”, “load”)
  - → Possible variants (e.g., “load byte”, “load word”…)
- Oprands – source and destination
  - → Register, memory address, immediate
- Addressing Modes
  - → Impacts code size
    1. Encode as part of opcode (common in load-store architectures which use a few number of addressing modes)
    2. Address specifier for each operand (common in architectures which support may different addressing modes)

Fixed vs Variable Length Encoding

- Fixed Length
  - → Simple, easily decoded
  - → Larger code size

- Variable Length
  - → More complex, harder to decode
  - → More compact, efficient use of memory
    - ➤ Fewer memory references
    - ➤ Advantage possibly mitigated by use of cache
  - → Complex pipeline: instructions vary greatly in both size and amount of work to be performed

Instruction Encoding

- Tradeoff between variable and fixed encoding is size of program versus ease of decoding

- Must balance the following competing requirements:
  - → Support as many registers and addressing modes as possible
  - → Impact of size of the # of registers and addressing mode fields on the average instruction size
  - → Desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation
    - ➤ Multiple of bytes than arbitrary # of bits

- Many desktop and server choose fixed-length instructions
  - → ?

Putting it Together

- Use general-purpose registers with load-store arch
- Addressing modes: displacement, immediate, register indirect
- Data size: 8-, 16-, 32-, and 64-bit integer, 64-bit floating
- Simple instructions: load, store, add, subtract, …
- Compare: =, /=, <
- Fixed instruction for performance, variable instruction for code size
- At least 16 registers

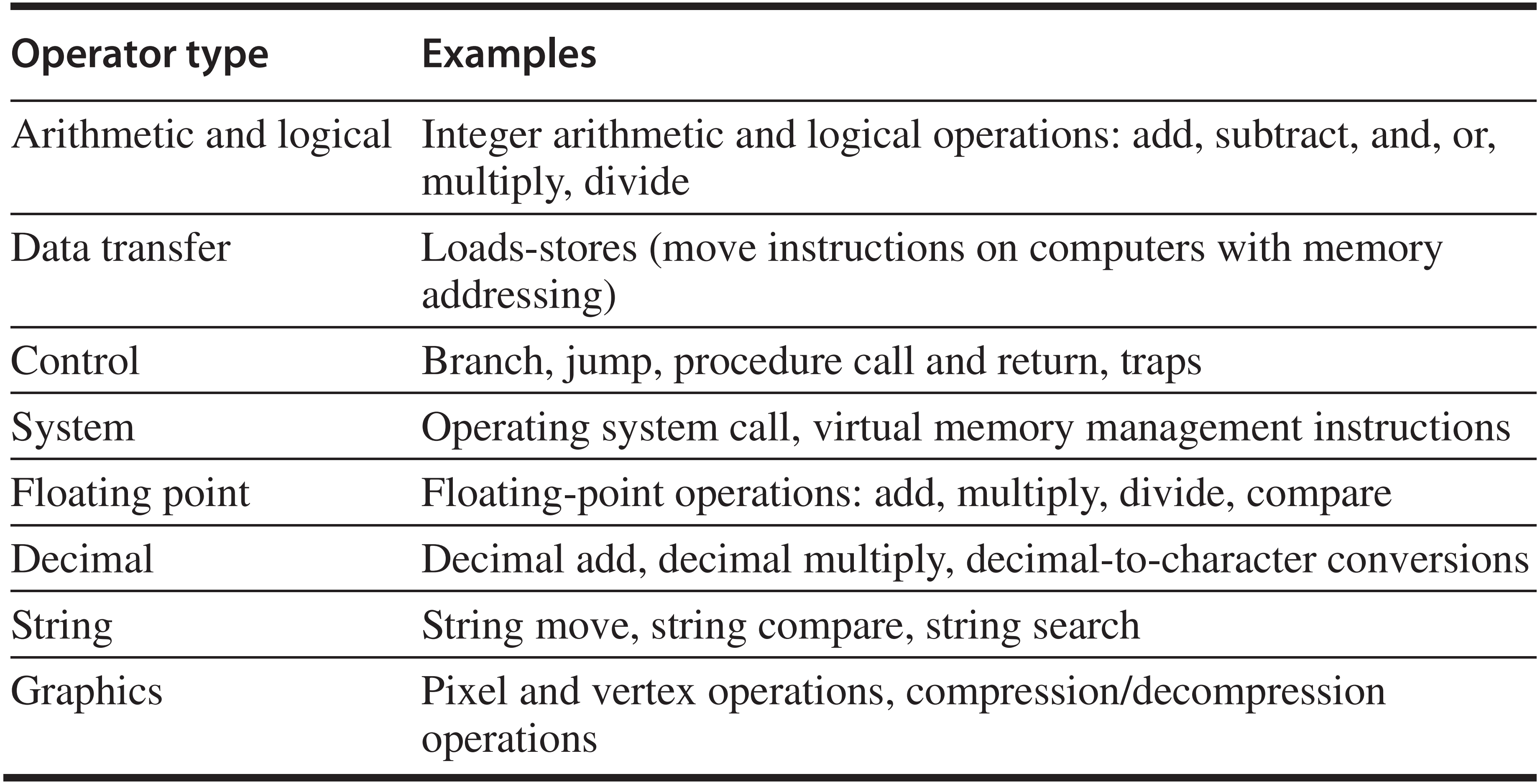
- Read section A9 to get an idea of MIPS ISA.
  - → Useful for understanding following discussions on pipelining

Pitfalls

- Designing “high-level” instruction set features to support a high-level language structure
  - → They do not match HL needs, or
  - → Too expensive to use
  - → Should provide primitives for compiler
- Innovating at instruction set architecture alone without accounting for compiler support
  - → Often compiler can lead to larger improvement in performance or code size

Backup

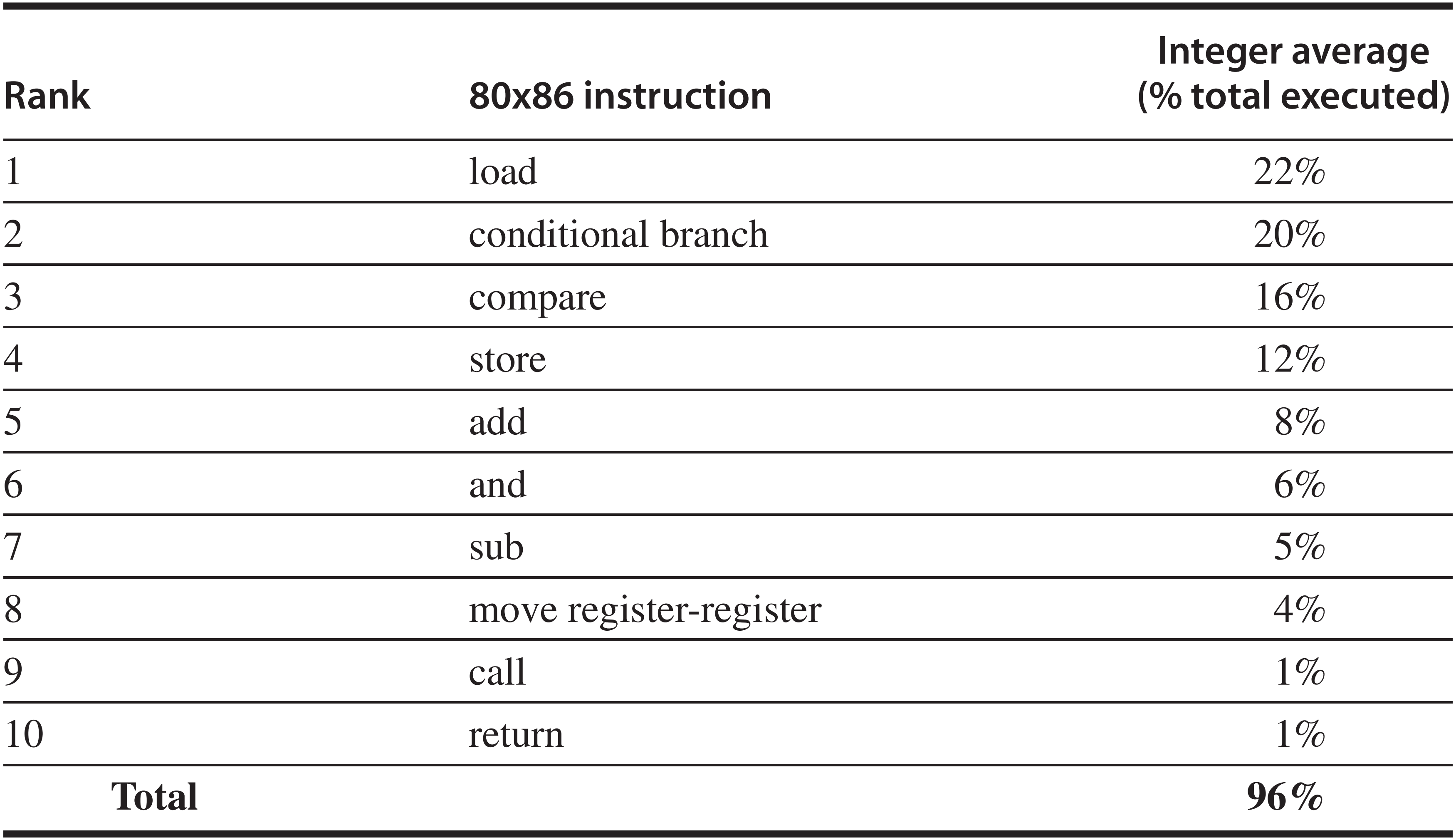
Types of Instructions



Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Operations supported by most ISAs

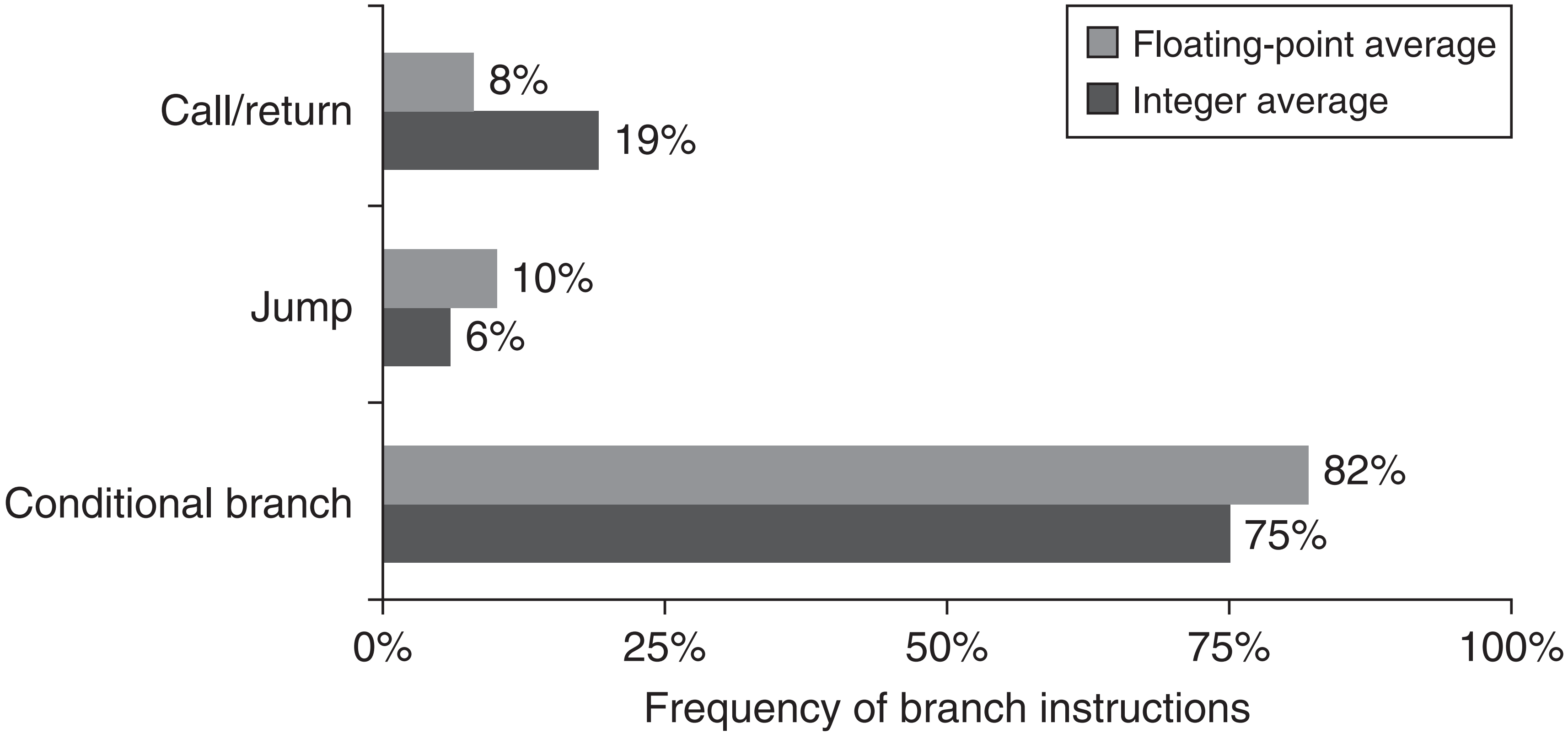
Instruction Distribution



Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		**96%**

Simple instructions dominate!

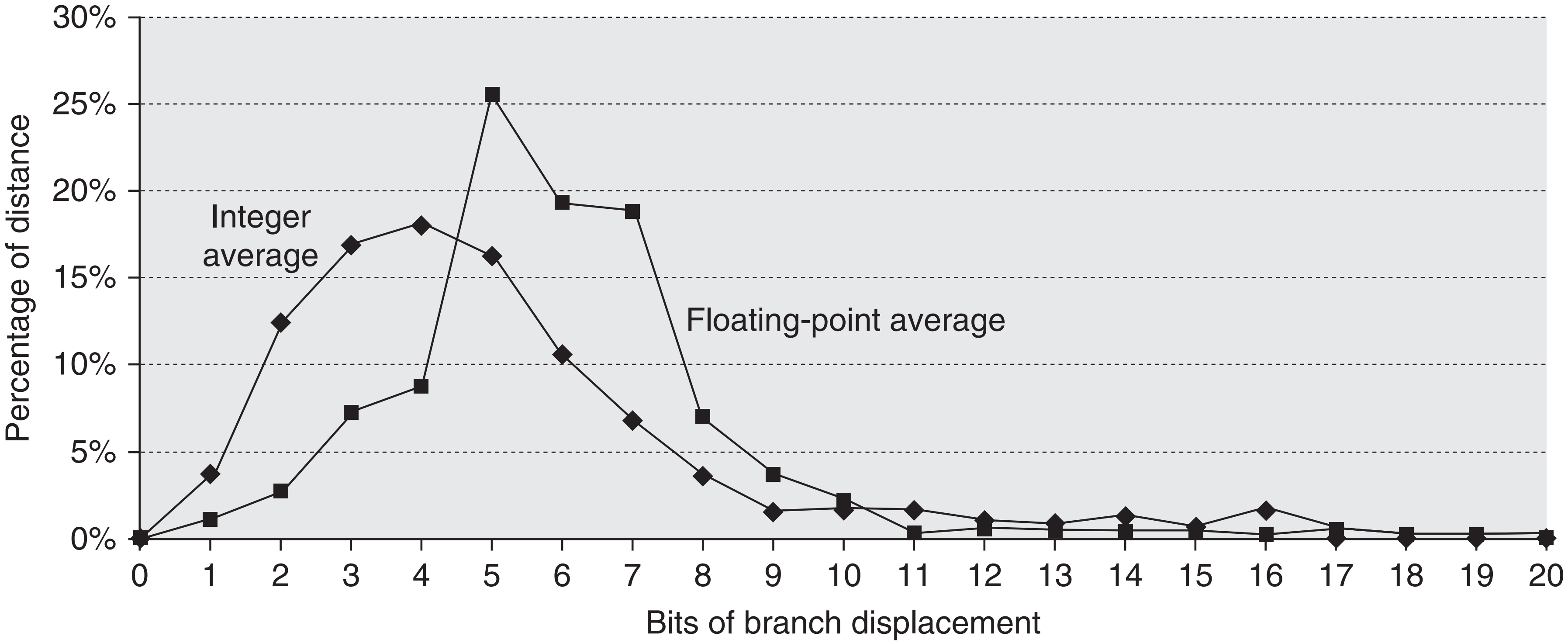
Control Flow Instructions



Conditional branches dominate!

SPEC CPU 2000 on Alpha

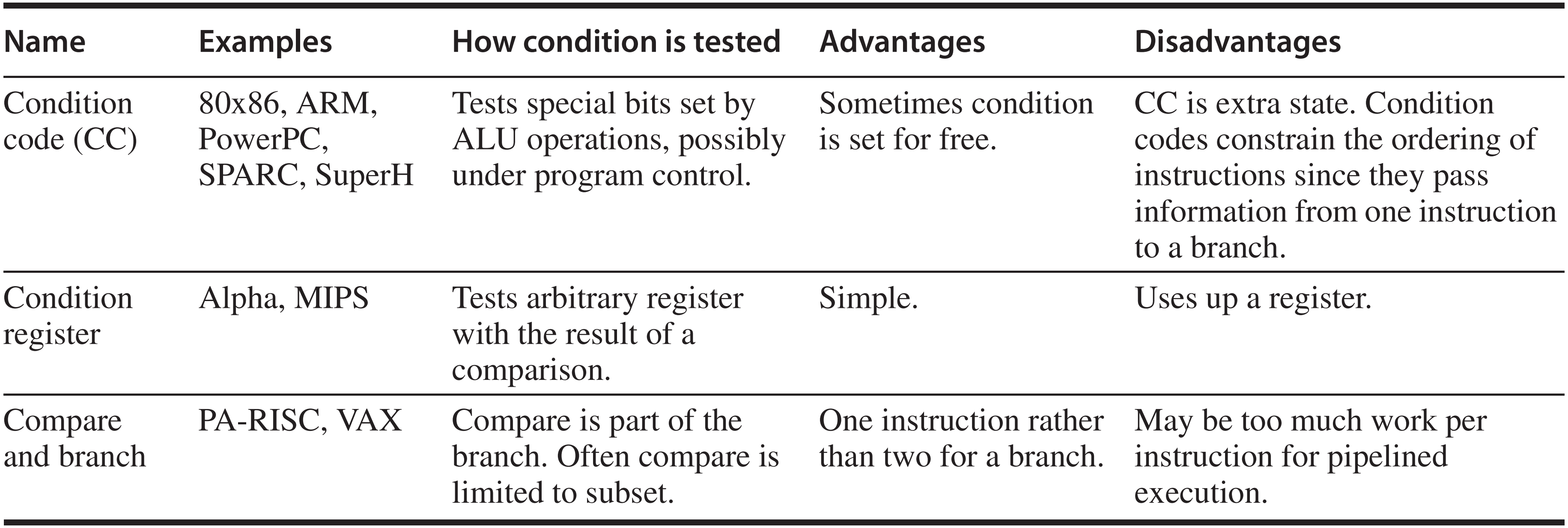
Conditional Branch Distances



4-8 bits can encode 90% branches!

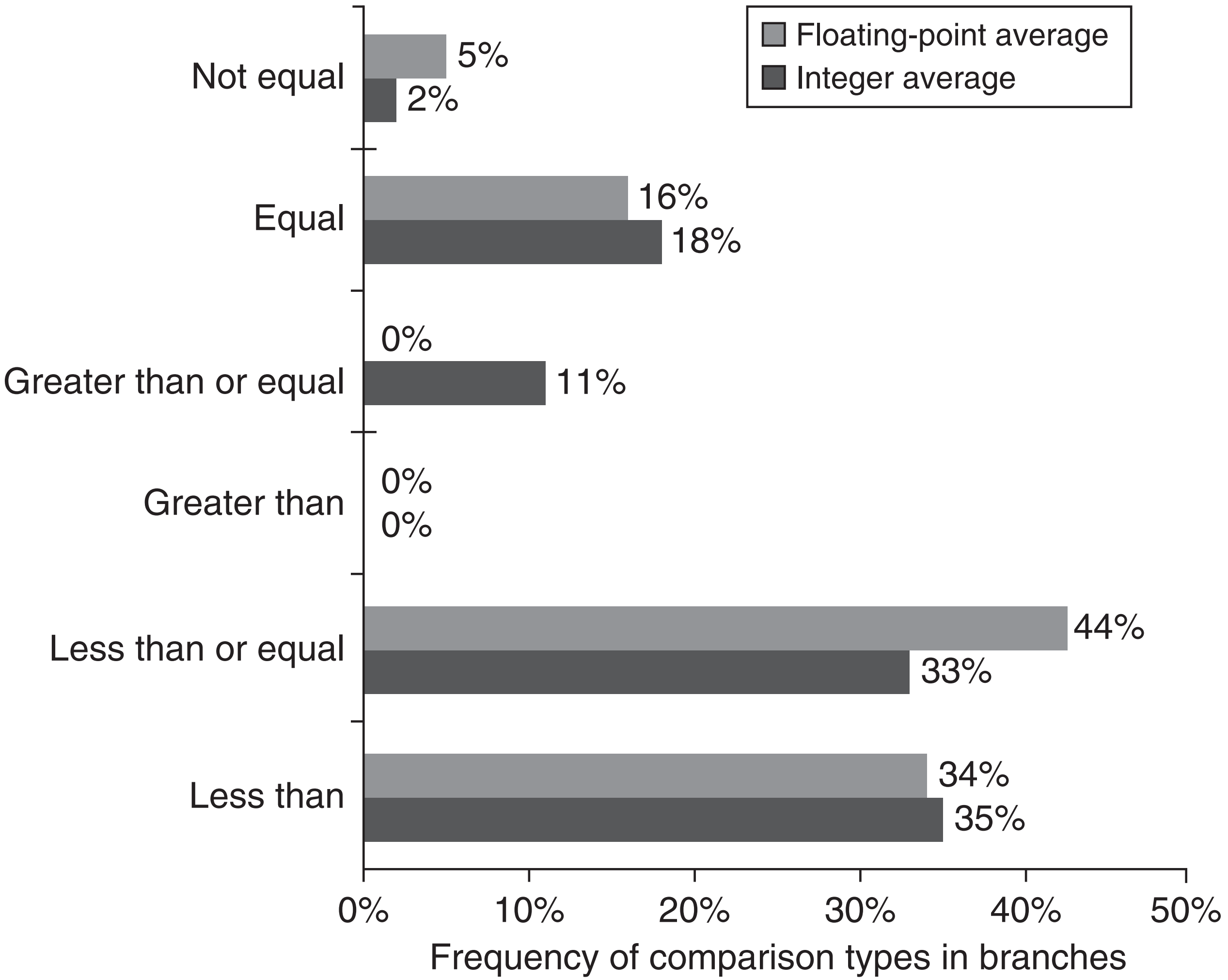
SPEC CPU 2000 on Alpha

Branch Condition Evaluation



Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

Types of Comparisons



SPEC CPU 2000 on Alpha