EEL 5764 Computer Architecture

Sandip Ray

Department of Electrical and Computer Engineering

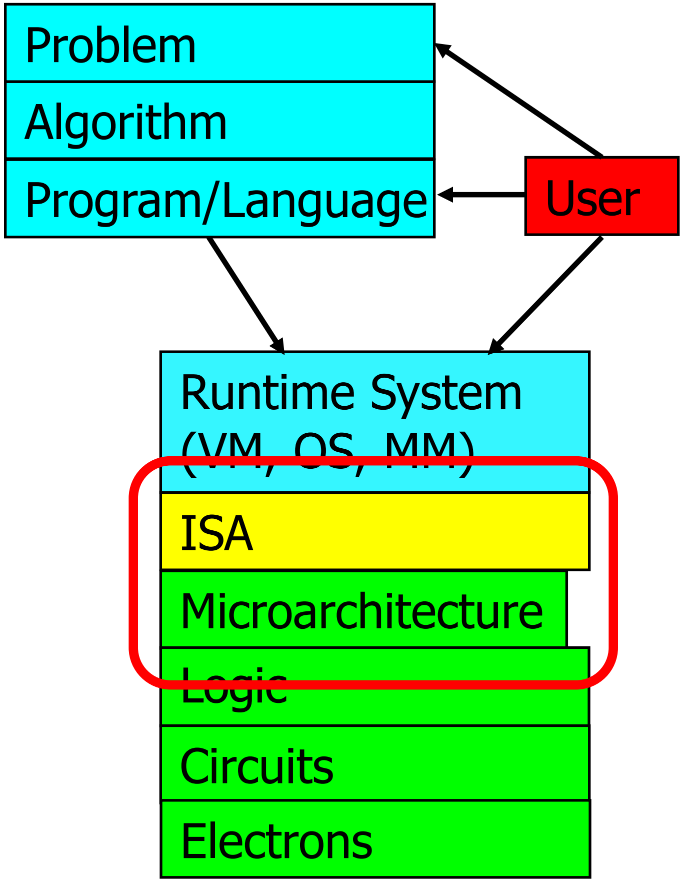
University of Florida

Lecture 16-17:

- Instruction Set Architecture
- Introduction to Pipelining

Abstractions

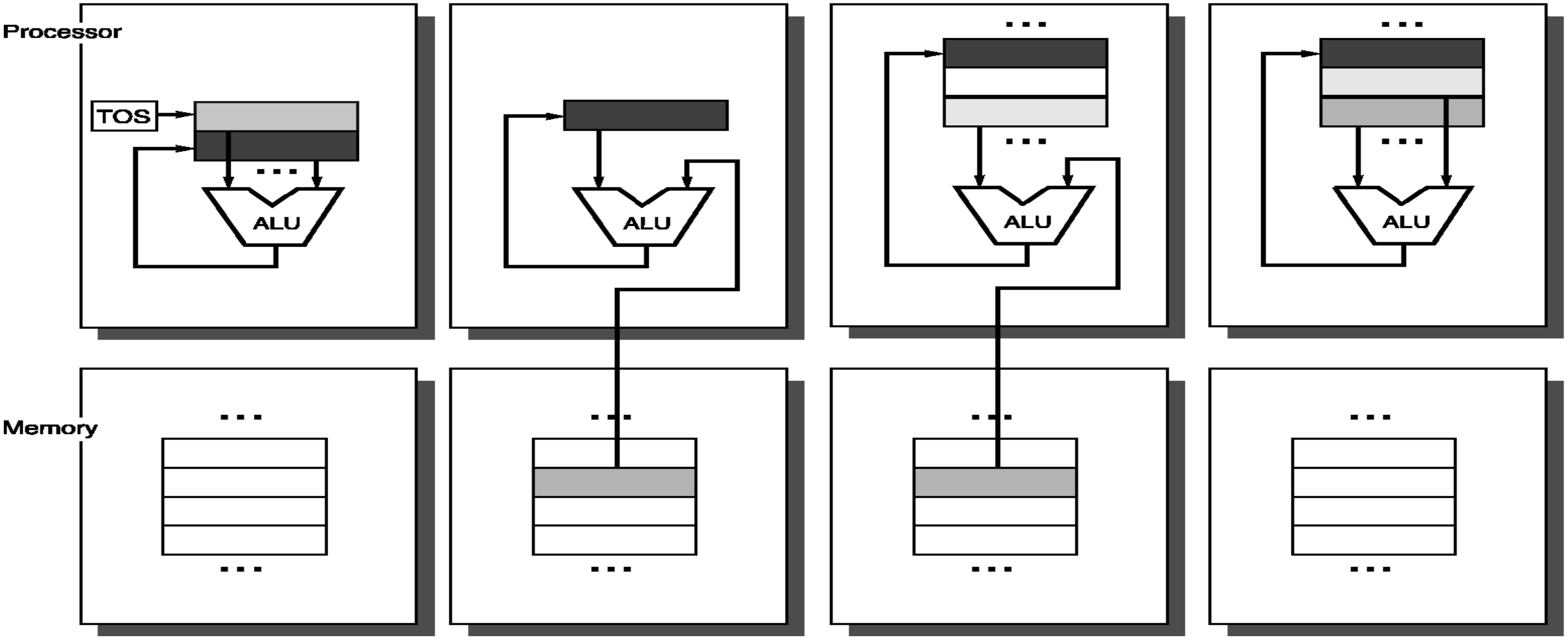
- Abstraction helps us deal with complexity
  - → Hide lower-level detail
- Instruction set architecture (ISA)
  - → The hardware/software interface
  - → Defines storage, operations, etc
- Implementation
  - → The details underlying the interface
  - → An ISA can have multiple implementations



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`

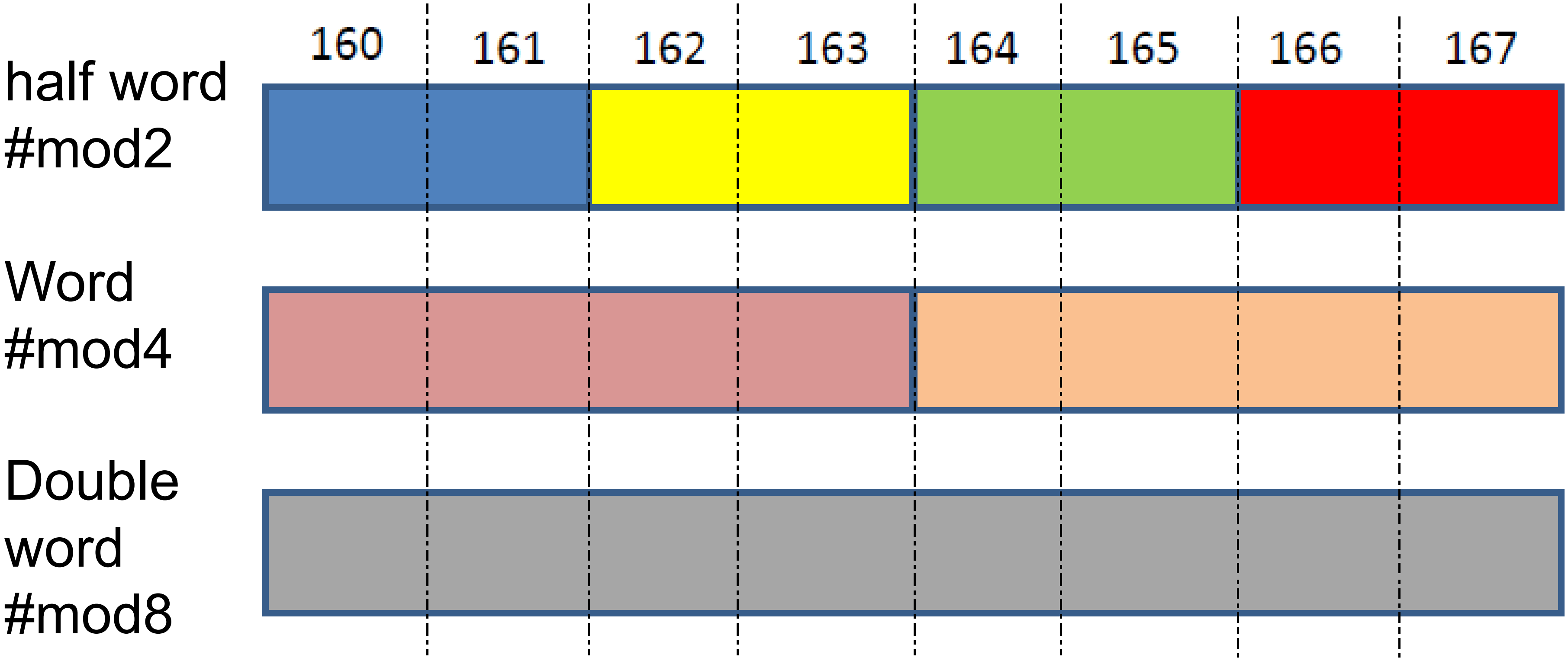


Memory Addressing

- Byte Addressing
  - → Each byte has a unique address
- 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

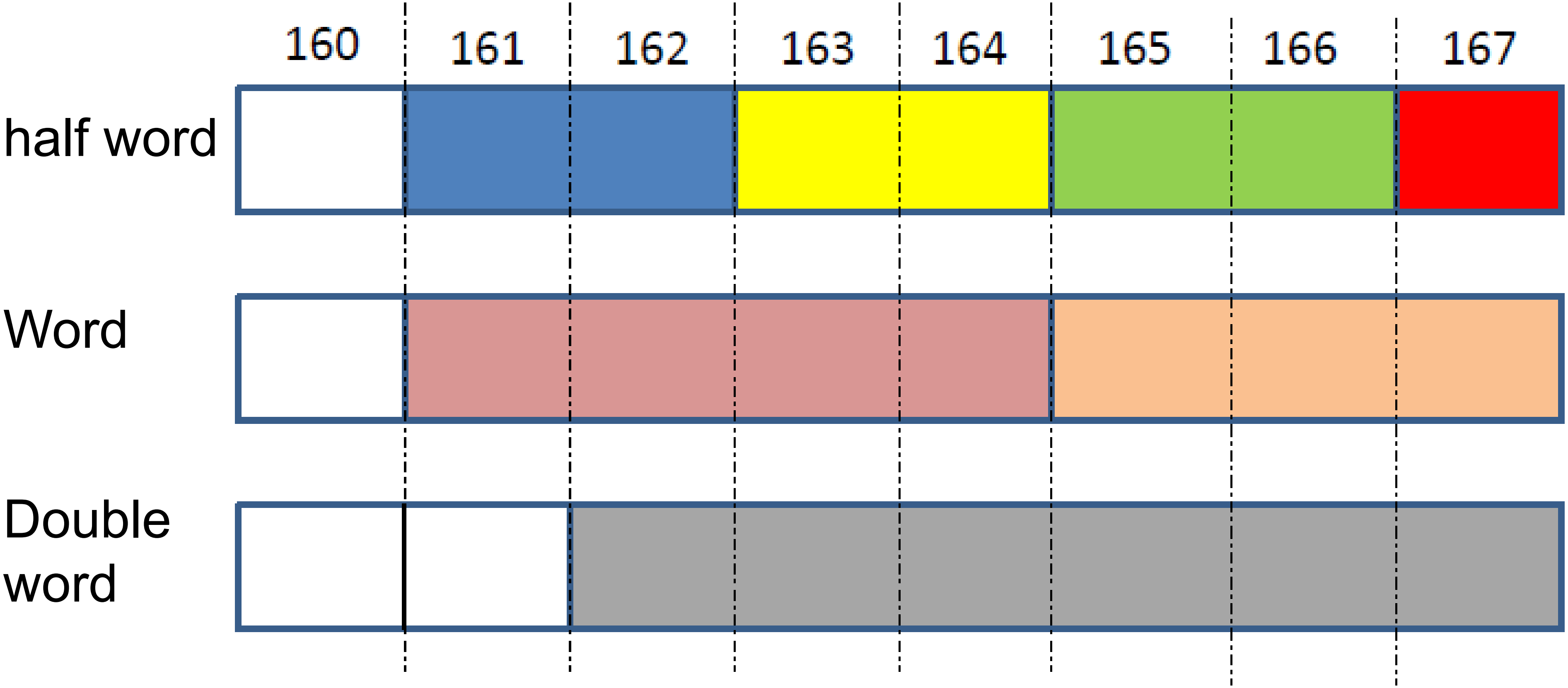
- Alignment
  - → Must half word, words, double words begin mod 2, mod 4, mod 8 boundaries



Aligned if *A mod s = 0*

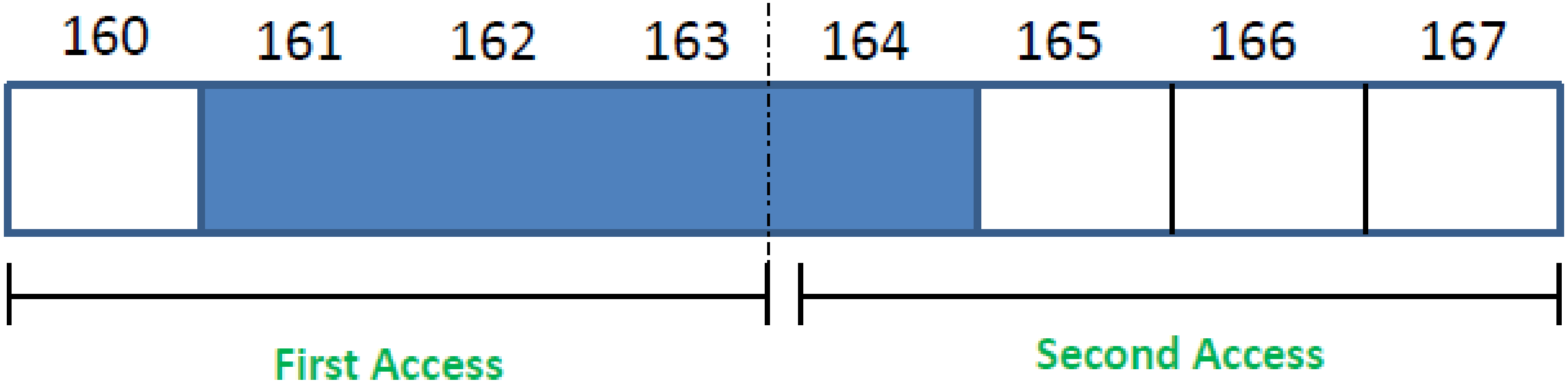
Memory Addressing

- Alignment
  → Or there no alignment restrictions



Memory Addressing

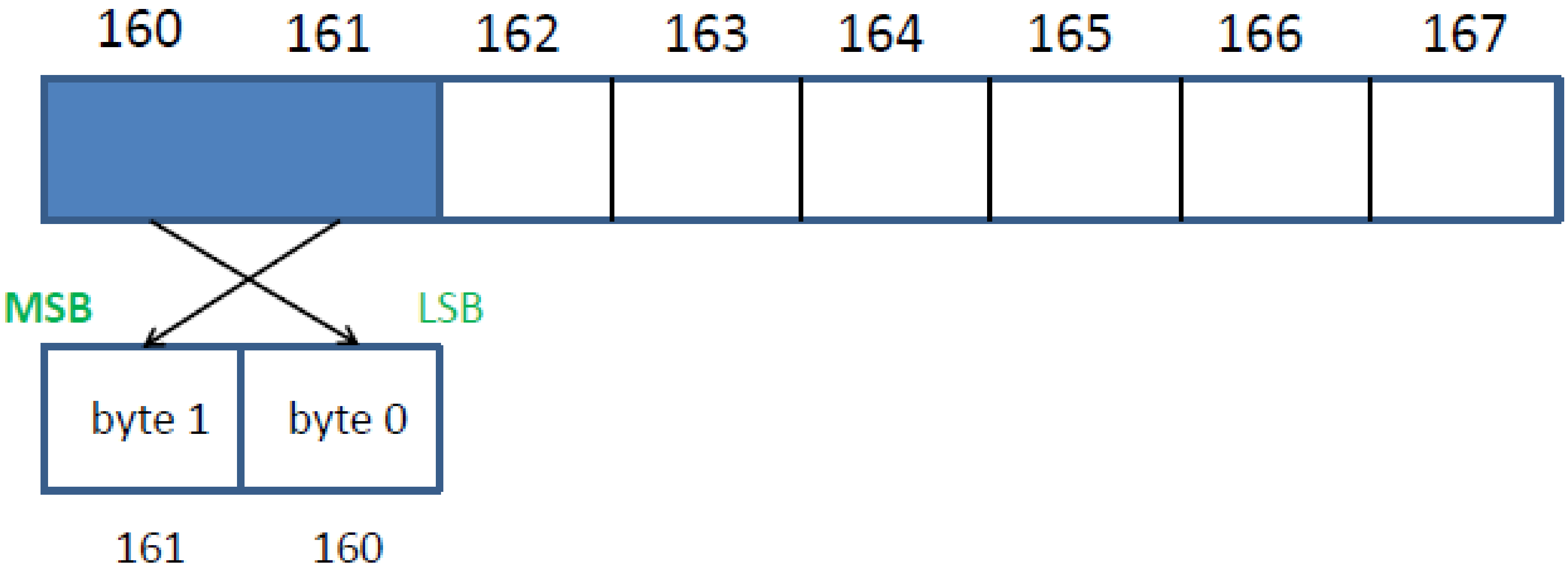
- Non-aligned memory references may cause multiple memory accesses



- 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

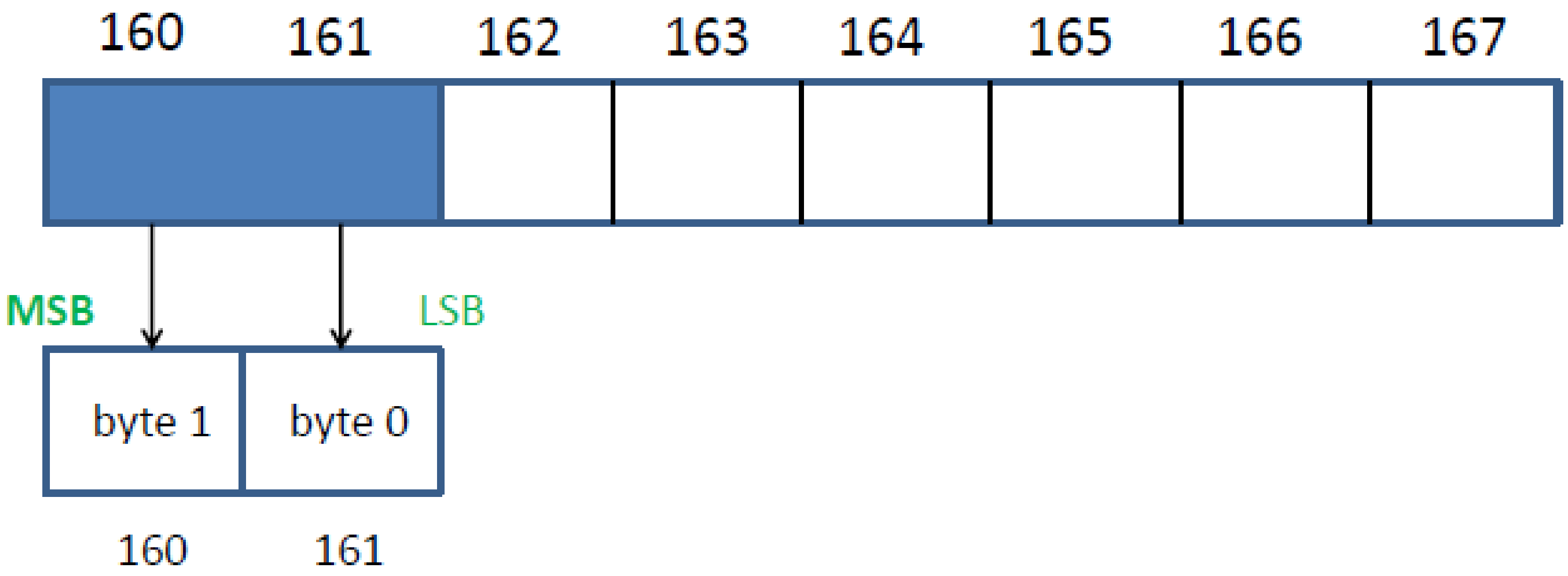
Byte Ordering – Little Endian

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



Byte Ordering – Big Endian

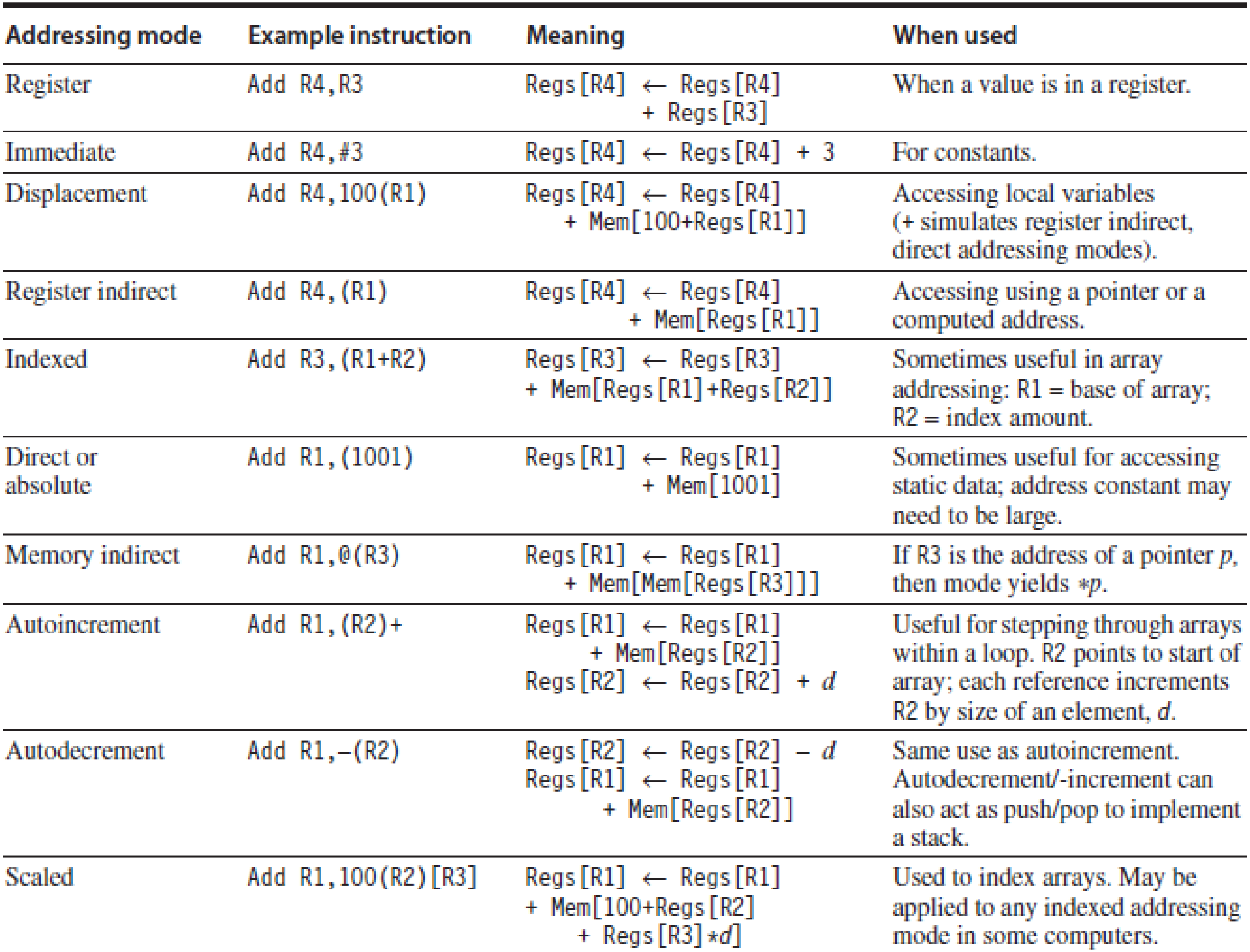
- The most significant byte within a word (or half word or double word) is stored in the smallest address



Byte Order in Real Systems

- Big Endian: Motorola 68000, Sun Sparc, PDP-11
- Little Endian: VAX, Intel IA32
- Configurable: MIPS, ARM

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:

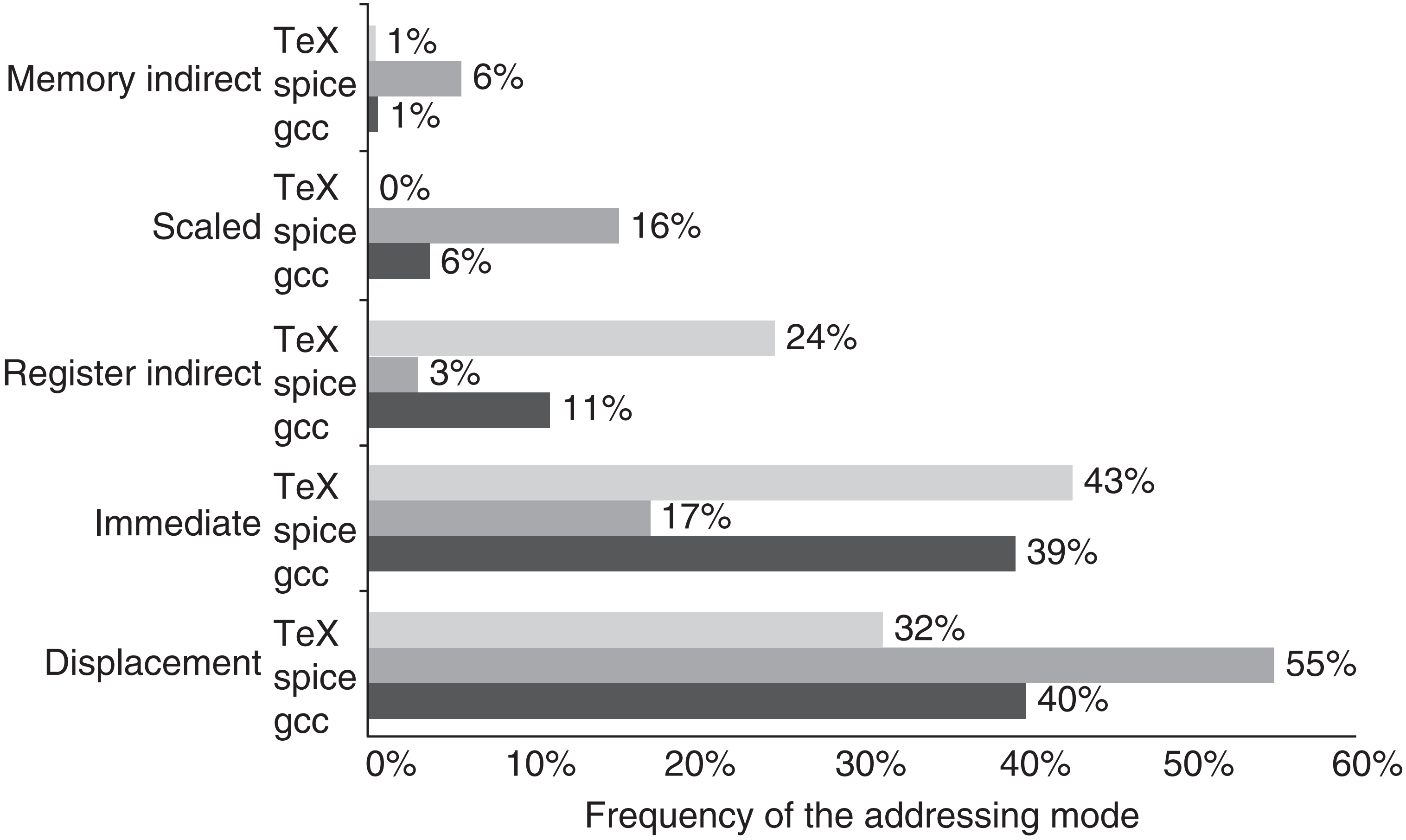
    ```
    Add R4, 100(R1)
    ```

  - → Without displacement mode

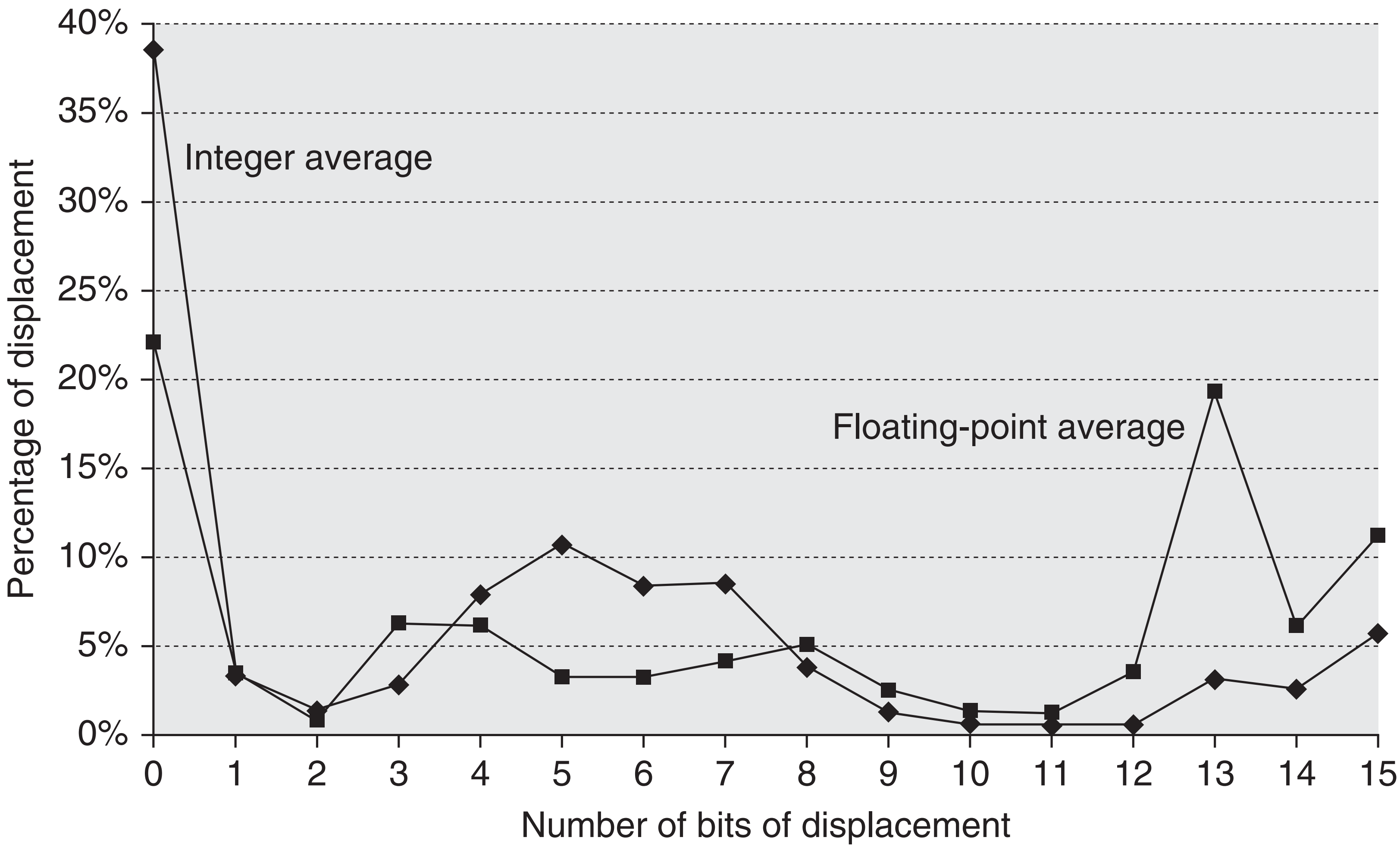
    ```
    Add R1, #100
    Add R4, (R1)
    Sub R1, #100
    ```

Which Addressing Modes to Support

- Support frequently used modes
  - → *Make common case fast!*



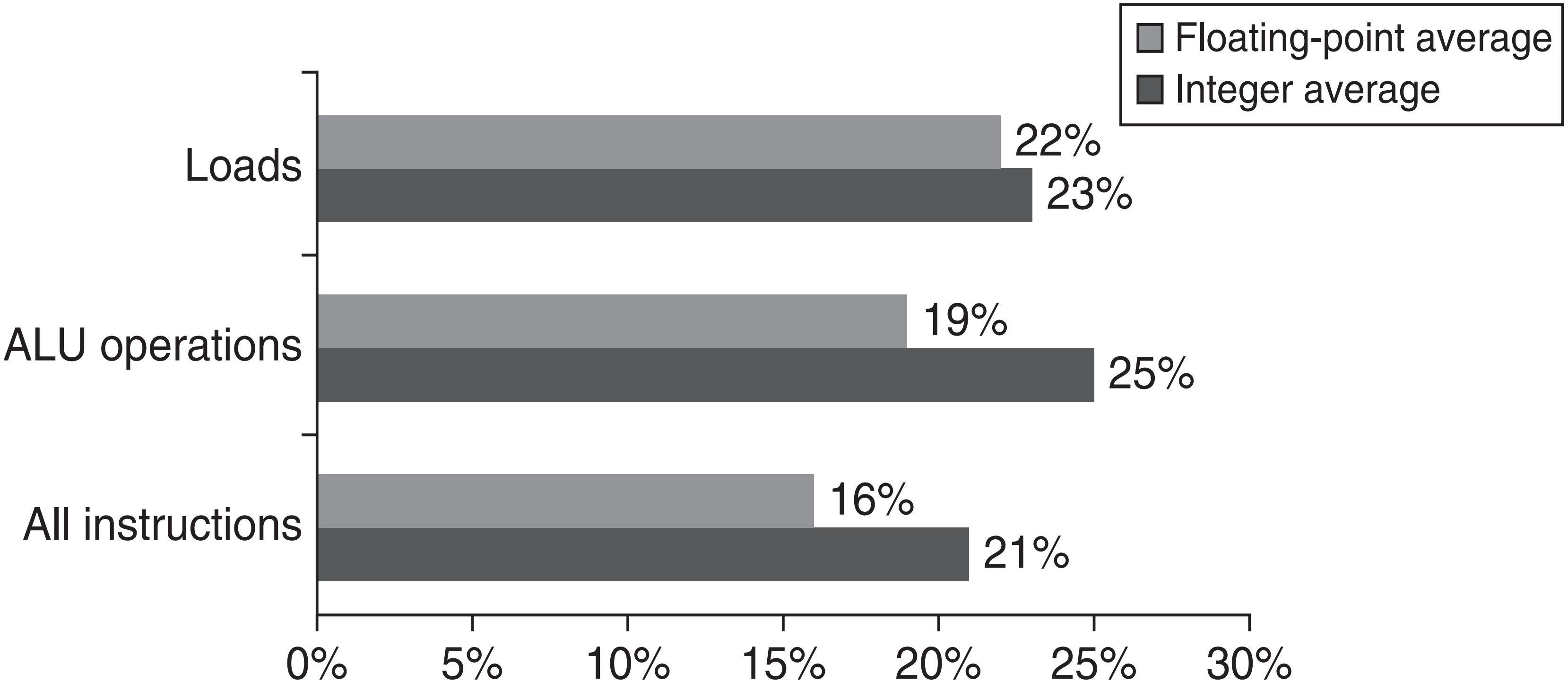
Displacement Value Distribution



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

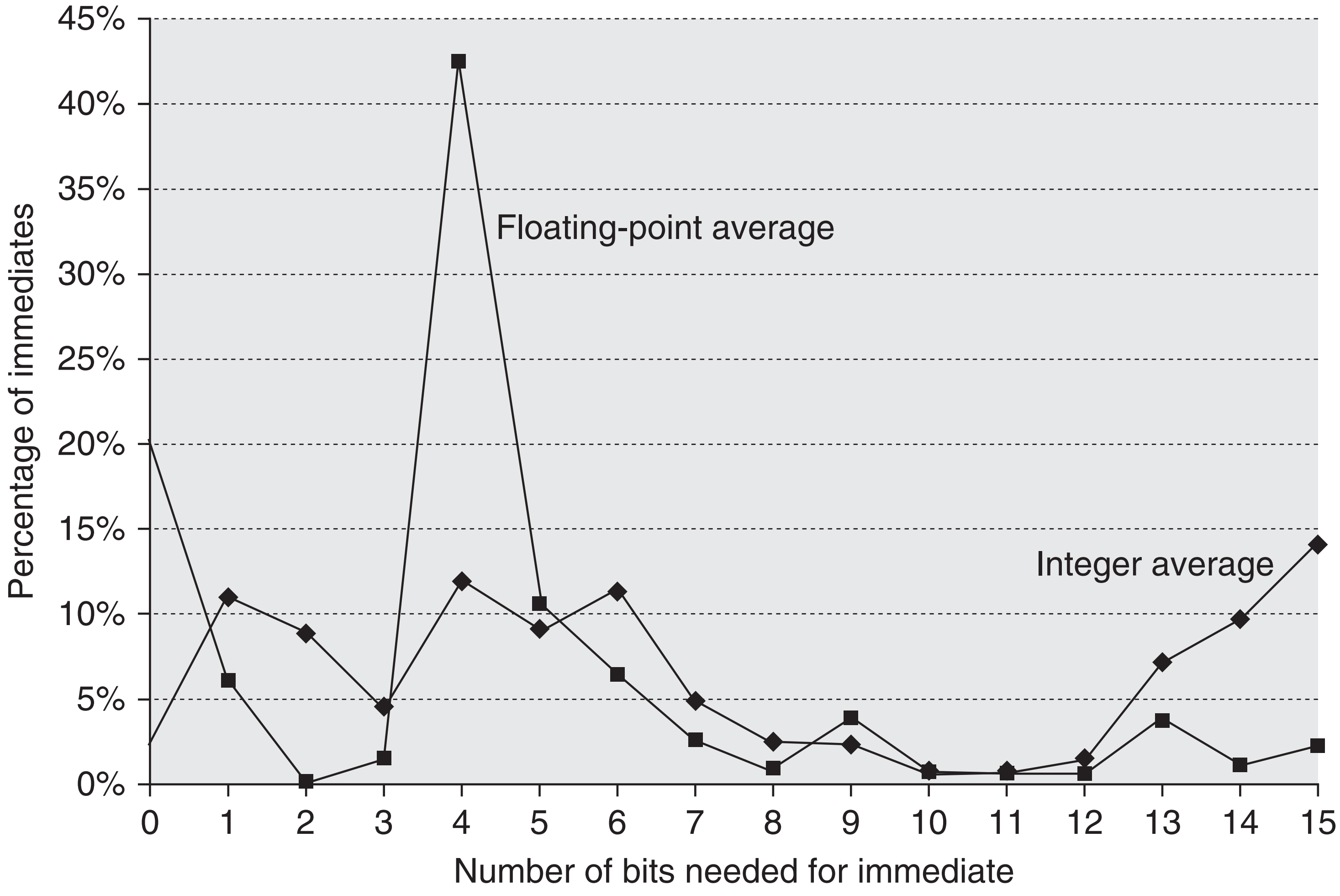
SPEC CPU 2000 on Alpha

Popularity of Immediates



add R4 **#3**

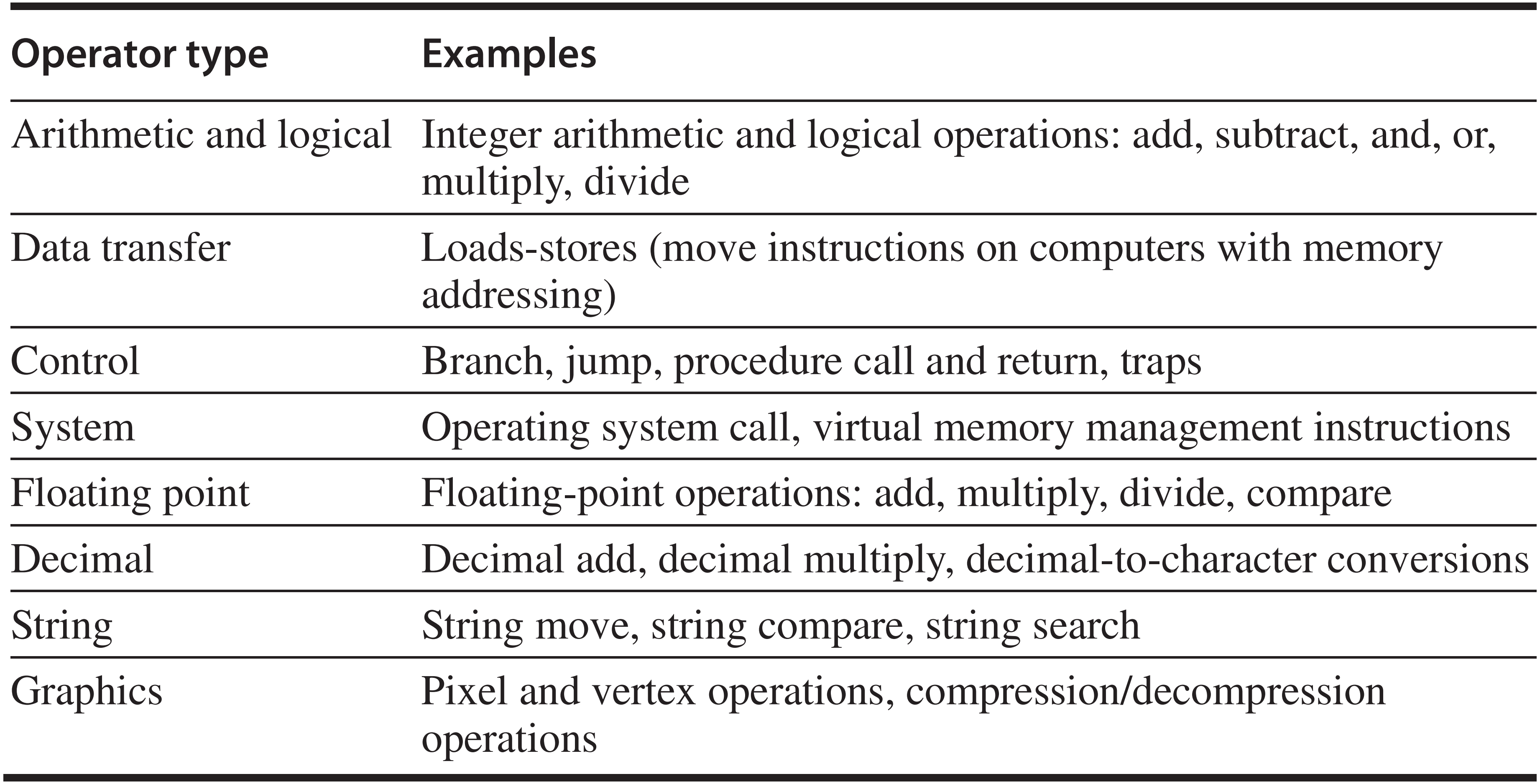
Distribution of Immediate Values



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

SPEC CPU 2000 on Alpha

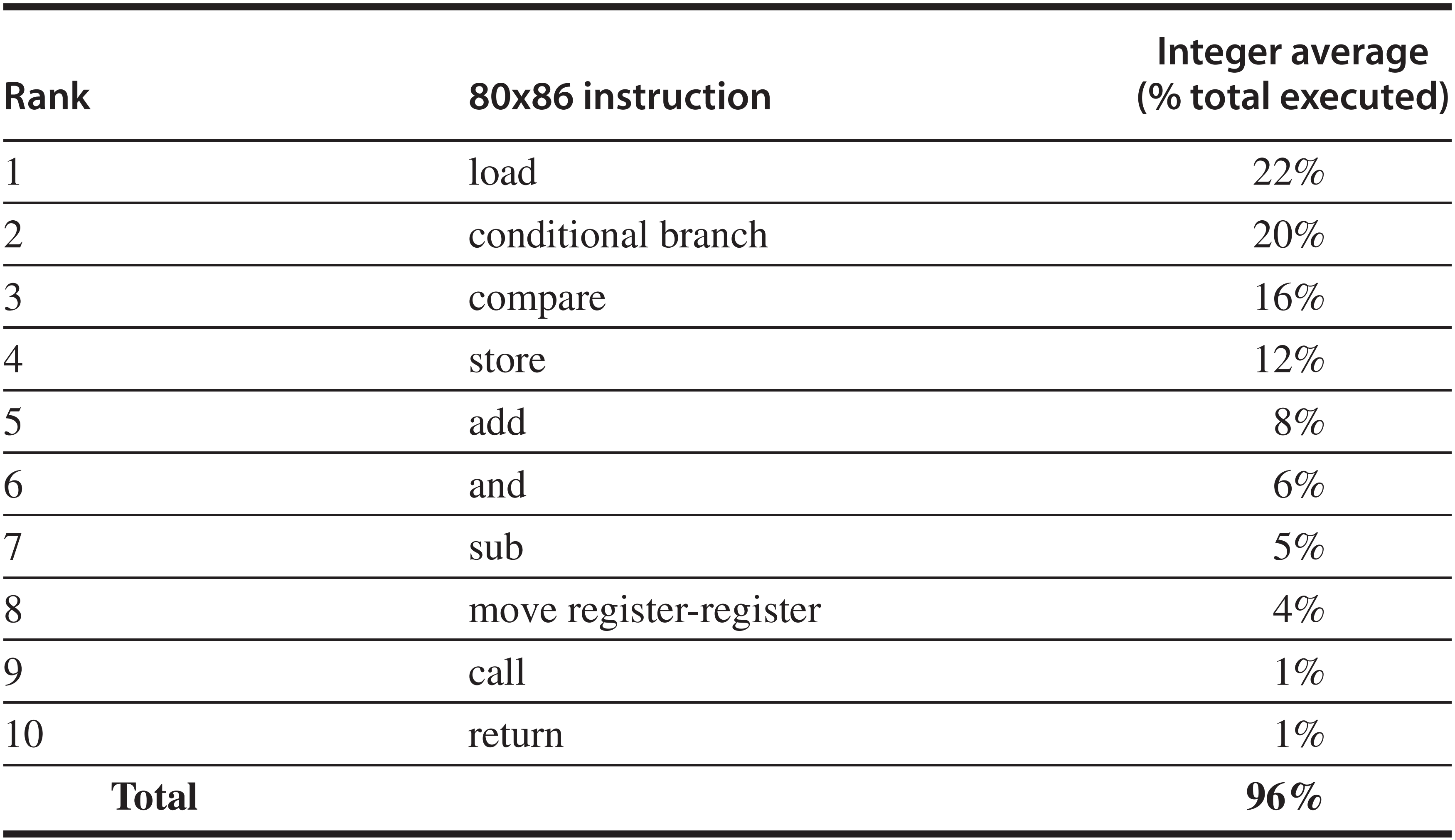
Types of Instructions



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

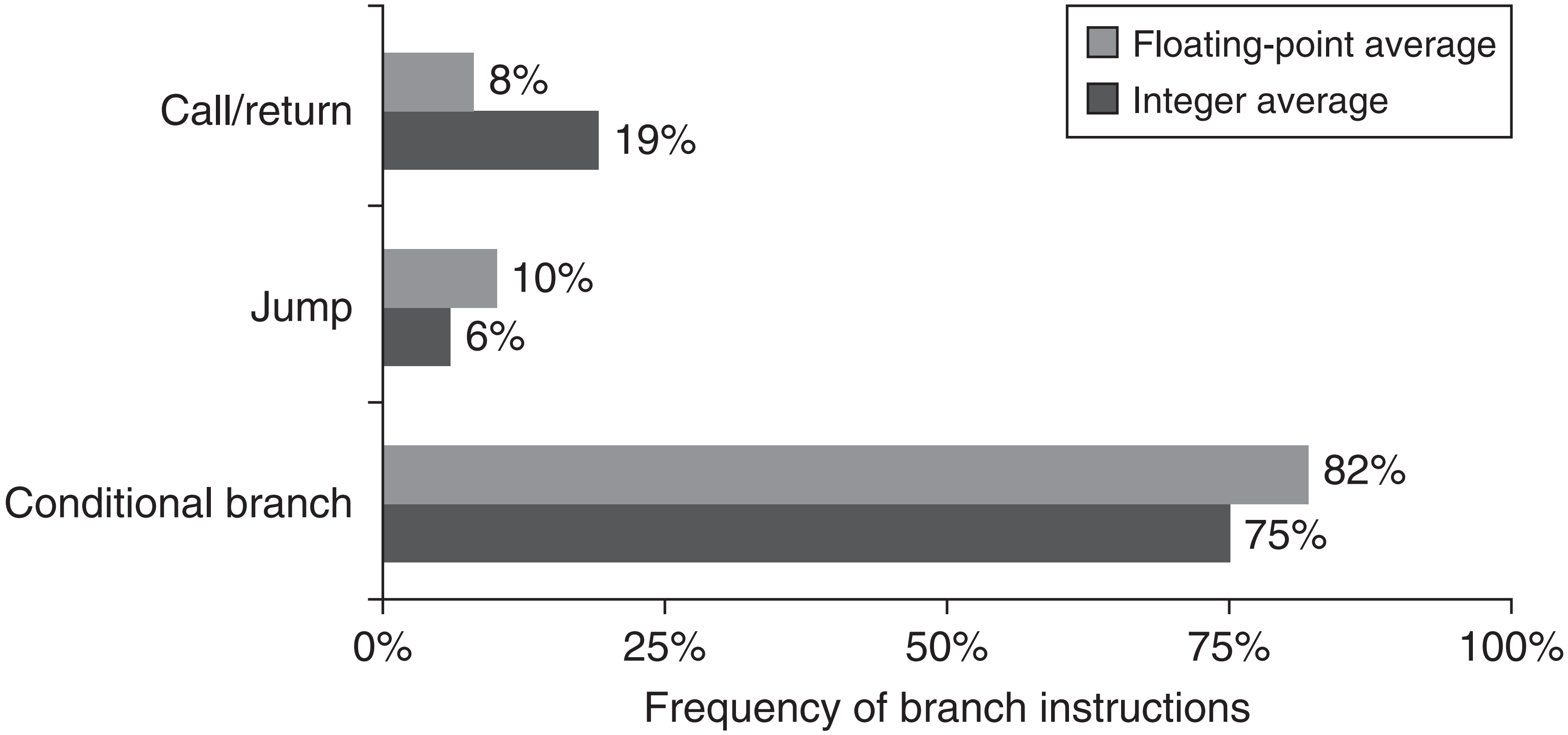
Instruction Distribution



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!

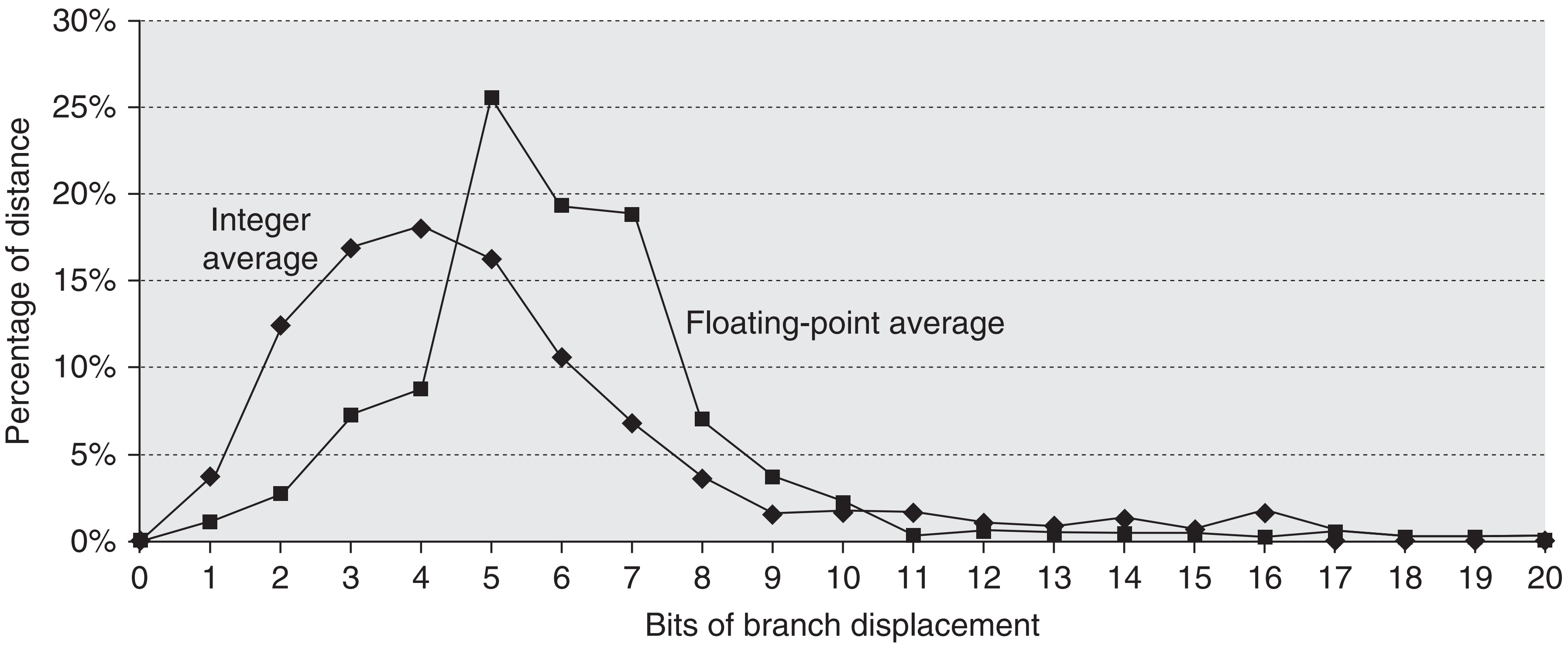
Control Flow Instructions



Conditional branches dominate!

SPEC CPU 2000 on Alpha

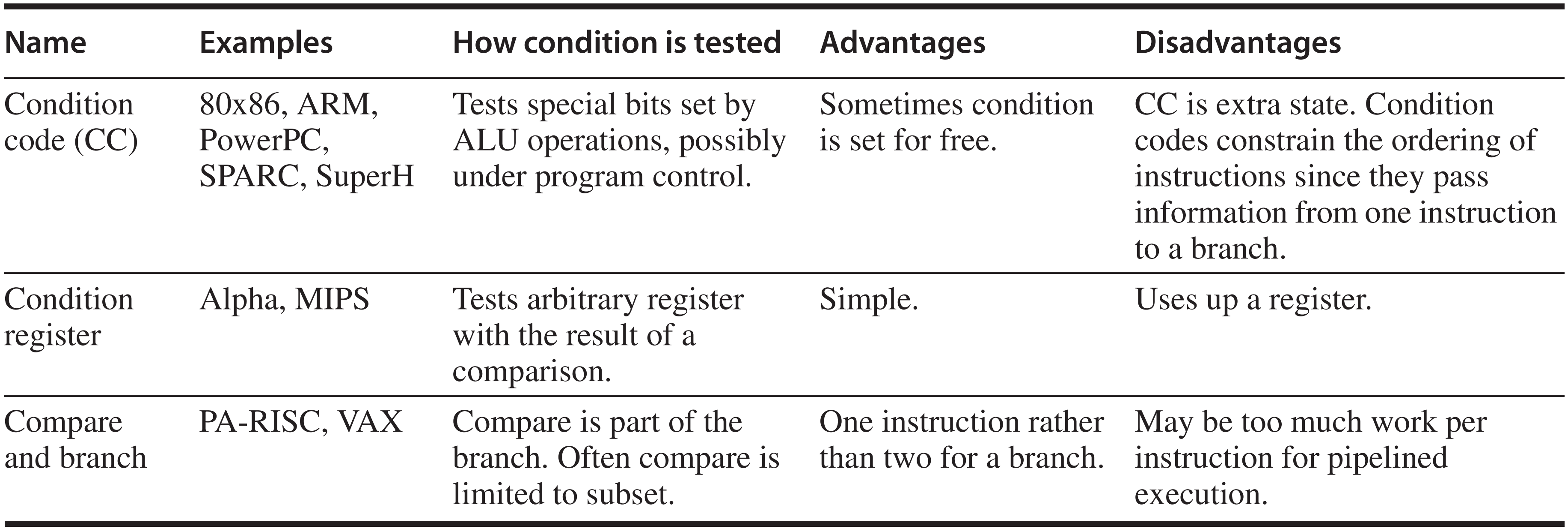
Conditional Branch Distances



4-8 bits can encode 90% branches!

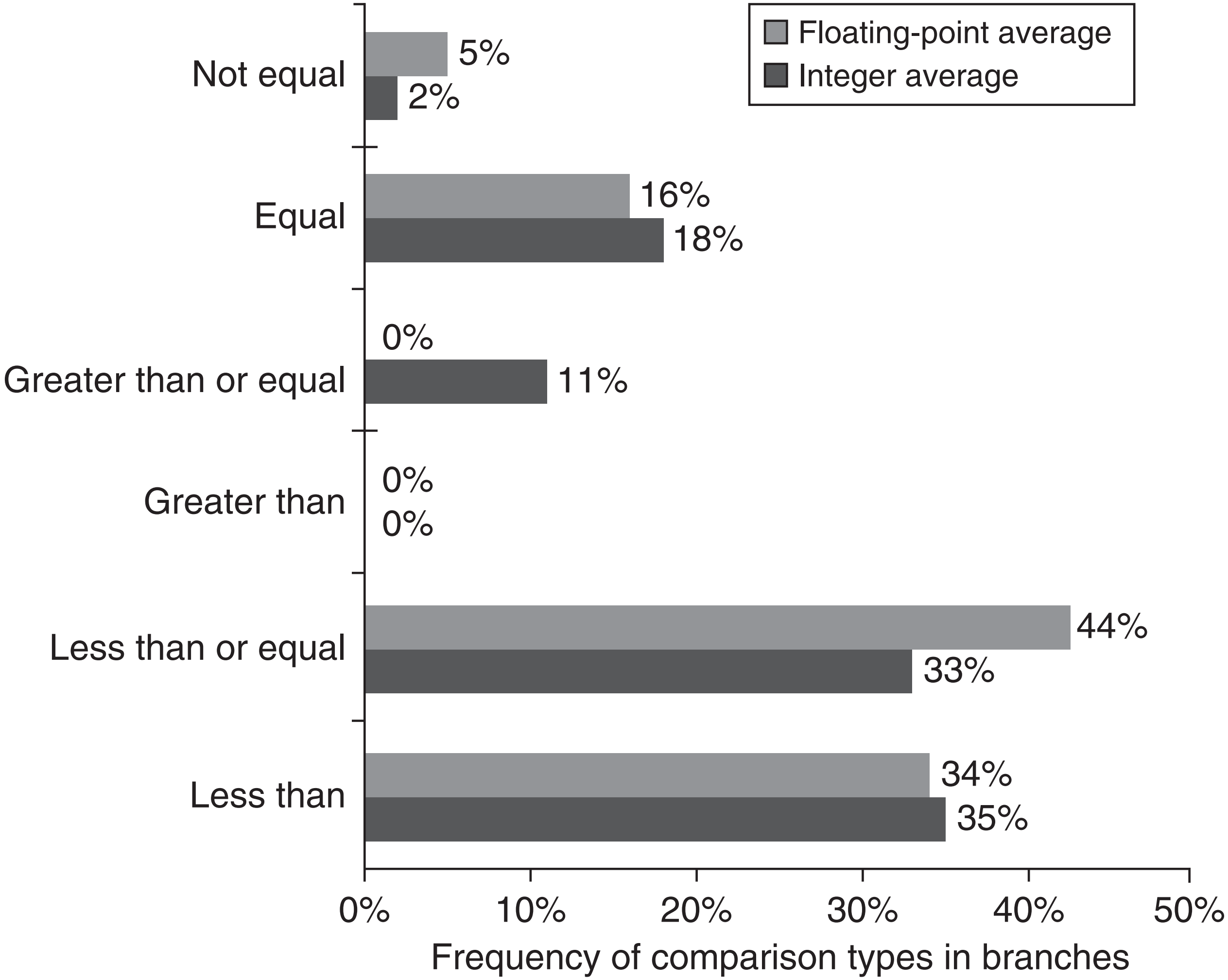
SPEC CPU 2000 on Alpha

Branch Condition Evaluation



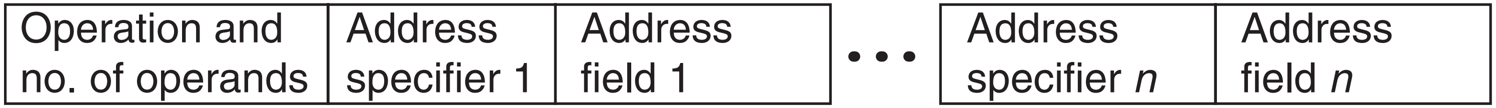
Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions 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



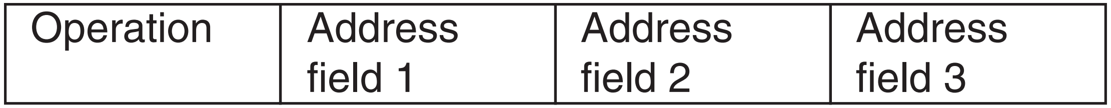
SPEC CPU 2000 on Alpha

Instruction Encoding



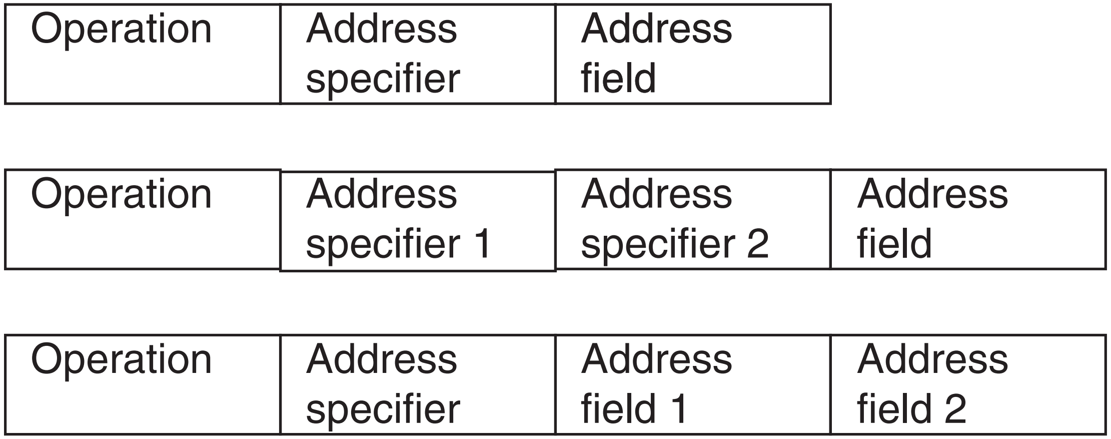
Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier *n*	Address field *n*

(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”…)
- Operands – 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 RISC 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

Introduction to Pipelining

Introduction

- Design Principle – exploit parallelism
- Pipelining become universal technique in 1985
  - → Overlaps execution of instructions
  - → Exploits "Instruction Level Parallelism"
- There are two main approaches:
  - → Hardware-based dynamic approaches
    - ➤ Used in server and desktop processors
    - ➤ Not used as extensively in PMD processors
  - → Compiler-based static approaches
    - ➤ Not as successful outside of scientific applications

Instruction Execution of RISC

- Initial State: PC is set to point to the first instruction

- For each instruction, perform the following 5 steps:

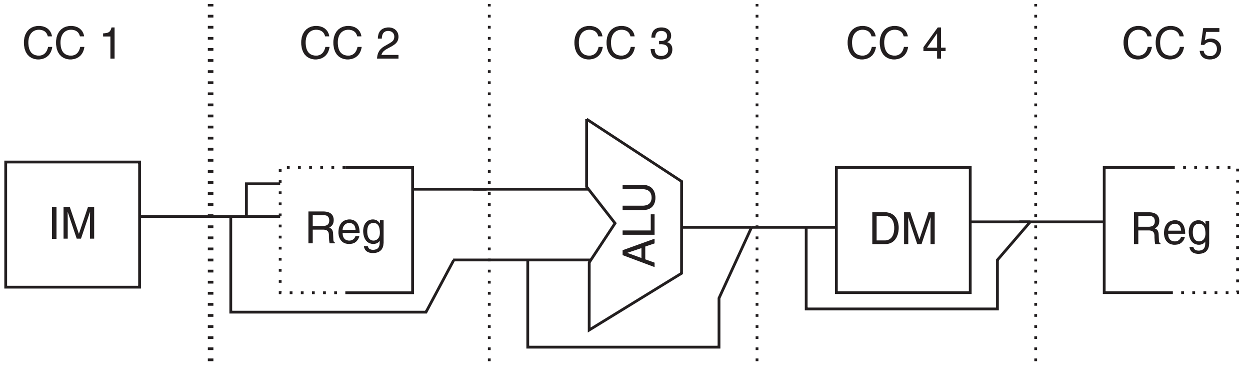
→ Instruction Fetch (**IF**)

→ Instruction Decode/Register Read (**ID**)

→ Execution/Effective Address Calculation (**EX**)

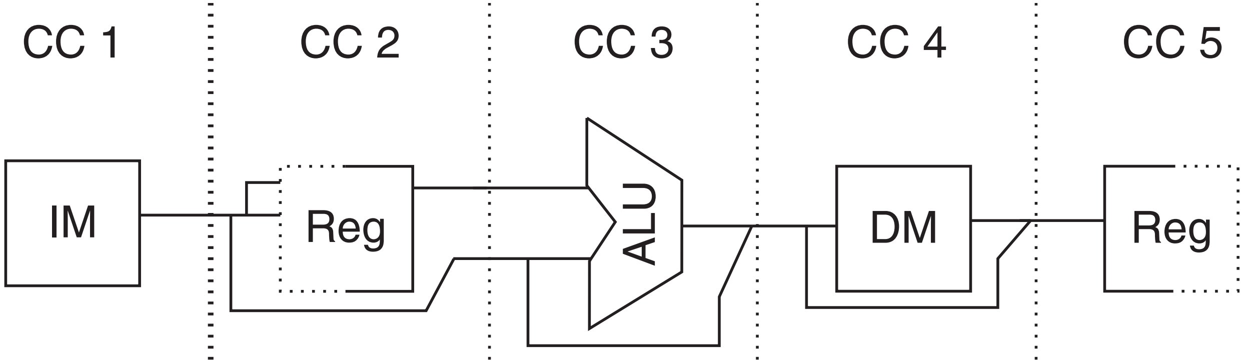
→ M

→ W



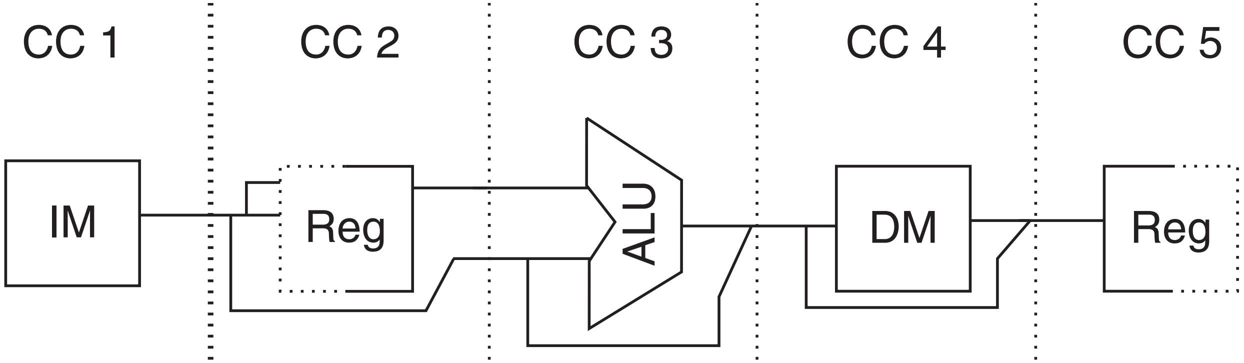
Instruction Execution

- **Instruction Fetch**:
  - → Send PC to memory, assert MemRead signal
  - → Instruction read out from memory
  - → Place instruction in IR: IR ← [PC]
  - → Update PC to next instruction: PC ← [PC] + 4



Instruction Execution

- **Instruction Decode**:
  - → Instruction in IR decoded by control logic, instruction type and operands determined
  - → Source operand registers read from general purpose register file

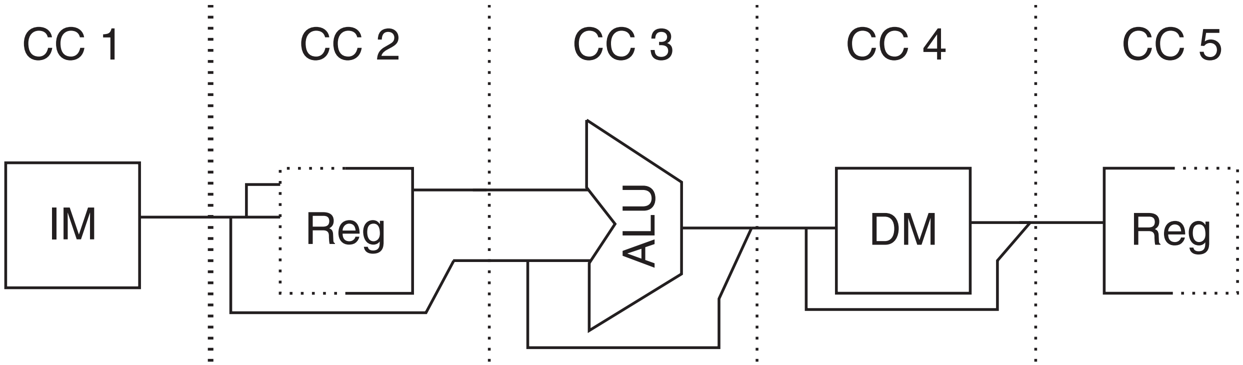


Instruction Execution

- **Execute**:
  - → ALU operates on operands prepared in previous cycle
  - → One of four functions depending upon opcode
  - → Memory Reference
    - ➤ Form effective address from base register and immediate offset
    - ➤ ALU Output ← [A] + Imm
  - → Register-Register ALU Instruction
    - ➤ ALU Output ← [A] func [B]
  - → Register-Immediate ALU Instruction
    - ➤ ALU Output ← [A] func Imm
  - → Branch
    - ➤ Compute branch target by adding Imm to PC
    - ➤ ALU Output ← [PC] + (Imm << 2)
    - ➤ Evaluate the branch condition

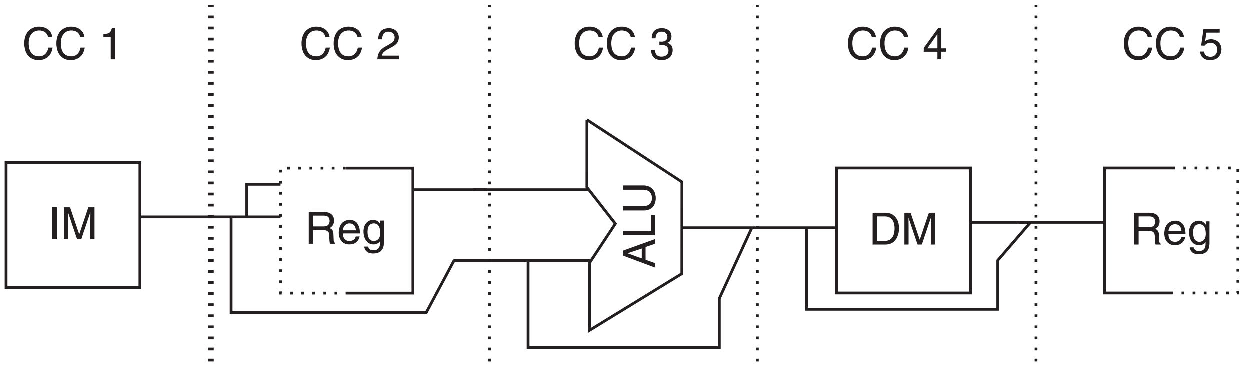
Instruction Execution

- **Memory Access**:
  - → For load instructions, read data from memory
  - → For store instructions, write data to memory



Instruction Execution

- **Writeback:**
  - →Results written to destination register



Instruction Execution – Example

- Add R3, R4, R5 ; R3←[R4]+[R5]
- Source registers: R4, R5 Destination register: R3
- Instruction steps:
  - → **Fetch**: Fetch the instruction and increment the program counter
  - → **Decode**: Decode the instruction in IR to determine the operation to be performed (add). Read the contents of registers R4 and R5
  - → **Execute**: Compute the sum [R4] + [R5]
  - → **Memory Access**: No action, since there are no memory operands
  - → **Writeback**: Write the result into register R3

Instruction Execution – Example

- Load R5, X(R7) ; R5←[[R7]+X]
- Source register: R7 Destination register: R5
- Immediate value X is given in the instruction word
- Instruction steps:

Fetch: Fetch the instruction and increment the program counter

Decode: Decode the instruction in IR to determine the operation to be performed (load). Read the contents of register R7

Execute: Add the immediate value X to the contents of R7

Memory Access: Use the sum X+[R7] as the effective address of the source operand, read the contents of that location from memory

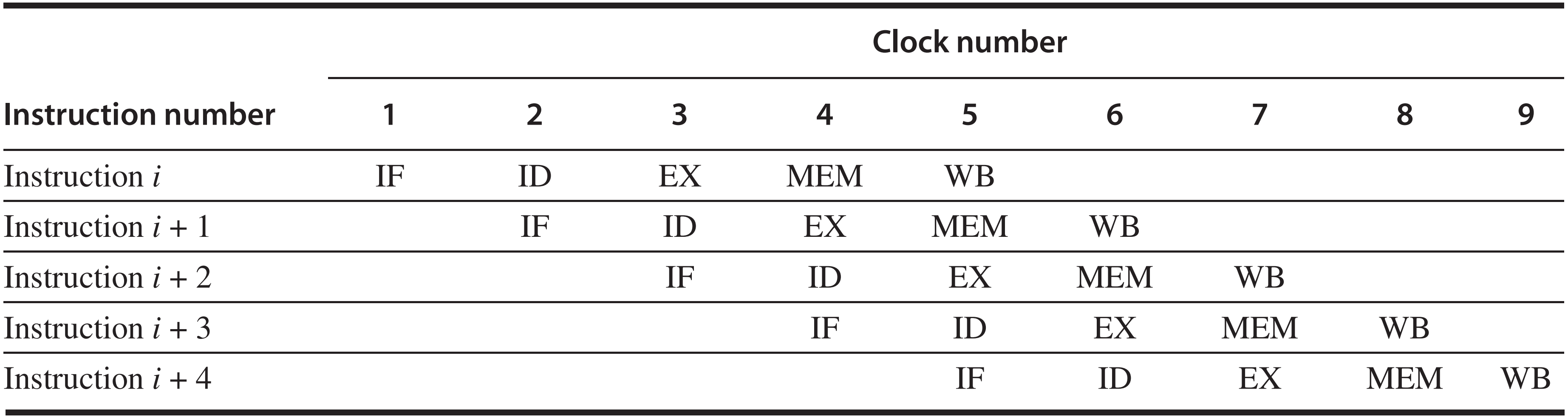
Writeback: Write the data received from memory into register R5

Instruction Execution – Example

- Store R6, X(R8)   ; Mem[X+[R8]]←[R6]
- Source registers: R6, R8 Destination register: None
- The immediate value X is given in the instruction word
- Instruction steps:
  - → **Fetch**: Fetch the instruction and increment the program counter
  - → **Decode**: Decode the instruction in IR to determine the operation to be performed (store). Read the contents of registers R6 and R8.
  - → **Execute**: Compute the effective address X + [R8]
  - → **Memory Access**: Store the contents of register R6 into memory location X + [R8]
  - → **Writeback**: No action

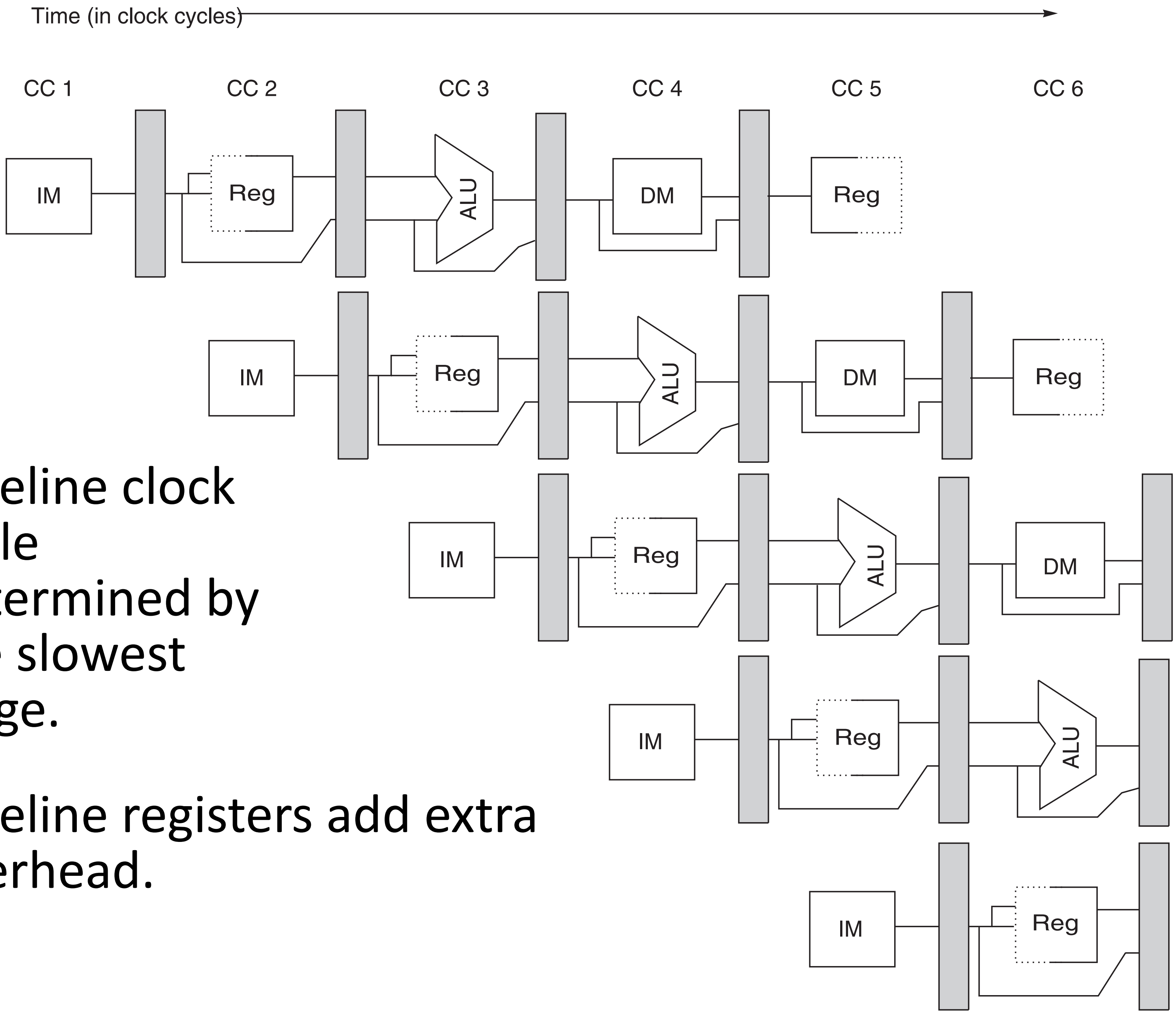
Basic Pipeline

To improve performance, we can make circuit faster, or use …



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

$$\text{Ideal time/instruction} = \frac{\text{time/instruction unpipelined}}{\text{\# pipeline stage}}$$



- Pipeline clock cycle determined by the slowest stage.

- Pipeline registers add extra overhead.

Ideal Pipeline and Performance

- Balanced pipeline (each stage has the same delay)
- Zero overhead due to clock skew and pipeline registers
- Ignore pipeline fill and drain overheads

$$Average\ time/instruction = \frac{Average\ time/instruction_{non-pipelined}}{Number\ of\ pipeline\ stages}$$

$$Speedup = \frac{Average\ time/instruction_{non-pipeline}}{Average\ time/instruction_{pipeline}}$$

$$= Number\ of\ pipelinestates$$

Pipeline Performance

- Example: A program consisting of 500 instructions is executed on a 5-stage processor. How many cycles would be required to complete the program. Assume ideal overlap in case of pipelining.
- Without pipelining:
  - → Each instruction will require 5 cycles. There will be no overlap amongst successive instructions.
  - → Number of cycles = 500 * 5 = 2500
- With pipelining:
  - → Each pipeline stage will process a different instruction every cycle. First instruction will complete in 5 cycles, then one instruction will complete in every cycle, due to ideal overlap.
  - → Number of cycles = 5 + ((500-1)*1) = 504
- Speedup for ideal pipelining = 2500/504 = 4.96