EEL 4768
Computer Architecture

Instruction Set Architecture (1)

Outline

- Instruction Set Architectures
- Memory Alignment
- Addressing Modes

Instruction Set Architecture (ISA)

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

What is an ISA?

- The ISA consists of:
  - List of all instructions
  - The format of the instructions
  - The addressing mode(s)

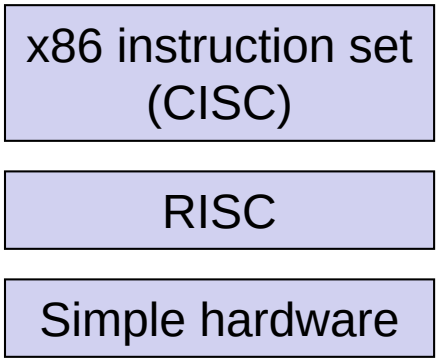
- Who has to deal with the ISA?
  - Assembly language programmer: to write an assembly program
  - The CPU hardware engineer: to implement the assembly instructions
  - A compiler writer: to translate a high-level code (e.g., C or C++) into assembly code

Types of ISA

- A common taxonomy: based on the type of internal storage:
  - **Accumulator architecture**
  - **Stack architecture**
  - **Memory-memory architecture**
  - **Register-memory architecture**
  - **Load-store architecture**

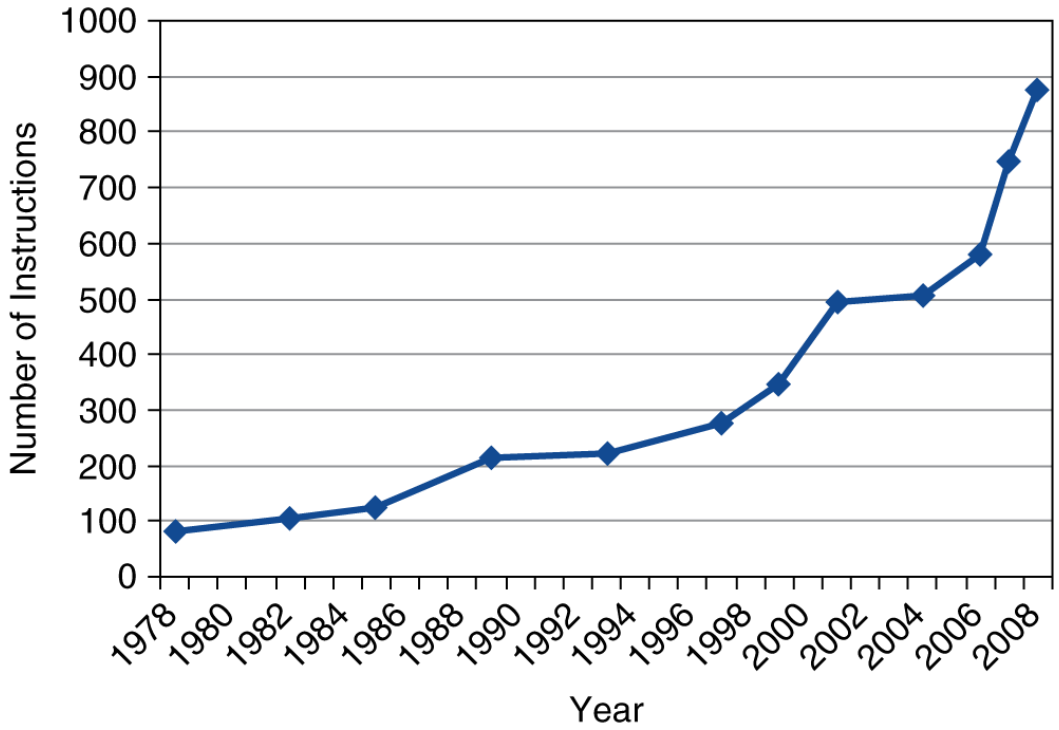
Layers of ISA

- Desire for simplicity in hardware implementation
  - RISC
- Intel's x86 instruction set, a CISC, provides backwards compatibility to earlier architectures
- Via layering: CISC to RISC conversion



Layers of ISA

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions

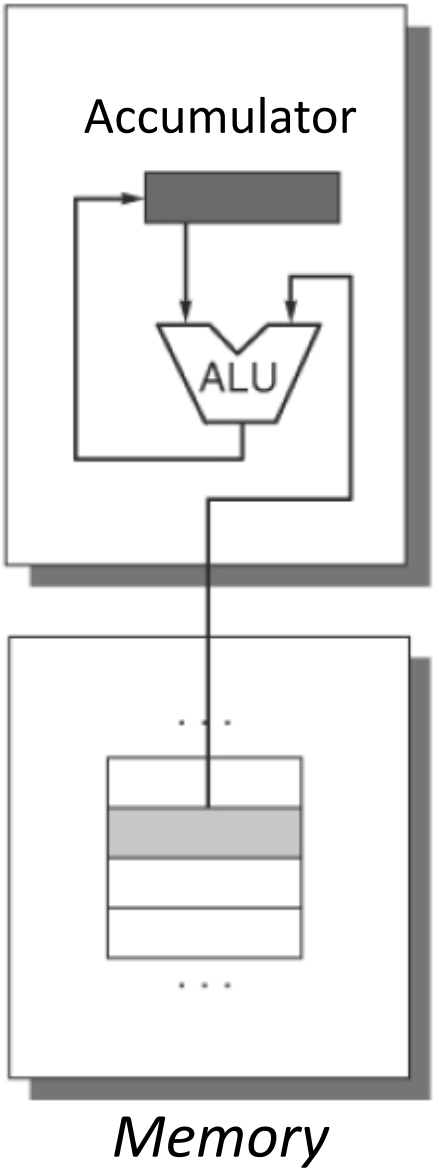


x86 instruction set

Accumulator Architecture

- The main approach in the earliest CPUs
- These CPUs didn't have a lot of storage space (not possible to put multiple registers)

- The accumulator is always an (implicit) operand of ALU
- The other (explicit) operand is in the memory



Accumulator Architecture

- The code below evaluates the expression: **C = A+B**
- The variables A, B and C are in the memory

```
Load   A      // load A in accumulator
Add    B      // add B to the accumulator
Store  C      // store the accumulator in C
```

Accumulator Architecture

- The accumulator code below evaluates this expression
- The variables A, B, C and D are in the memory

(A+B) * C / D

```
Load        A       // load A in the accumulator
Add         B       // add B to accumulator
Multiply    C       // multiply C to the accumulator
Div         D       // divide the accumulator by D
```

- *What is the accumulator code for this expression?*

(A+B) / (C+D)

Accumulator Architecture

- Some accumulator architectures have more than one register
- But the accumulator is used in all the operations
- The other operand can be another register or a memory address

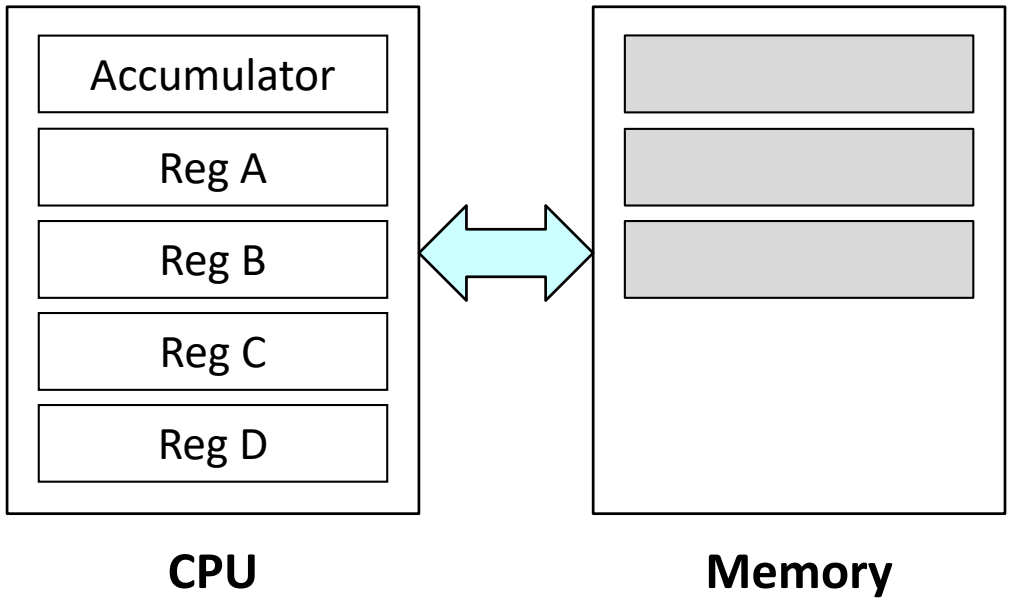
Instruction: **Add A**

- Accumulator = Accumulator + Register A

Instruction: **Add [200]**

- Accumulator = Accumulator +

Data at memory address 200

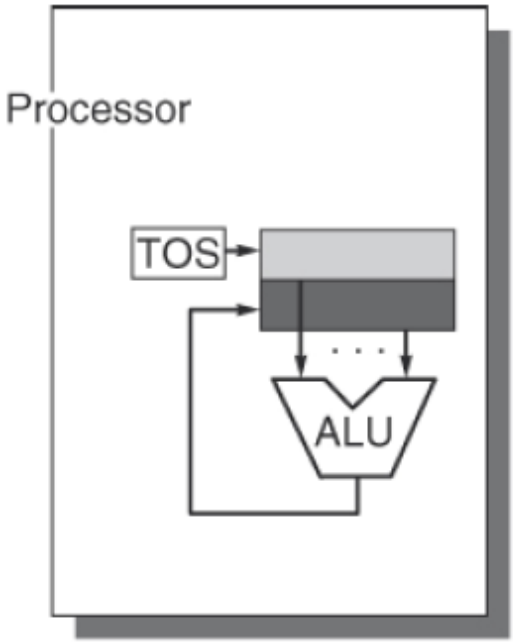


Accumulator Architecture

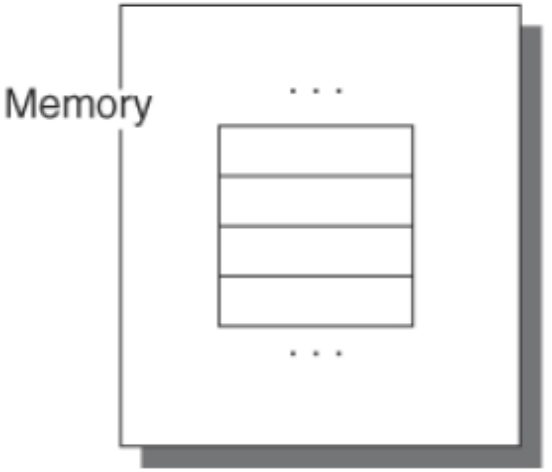
- Not used anymore – too rigid, yields too long assembly codes
- Modern architectures use general purpose registers which can store any value
- Pro:
  - The compiler is simple
  - Simple hardware
- Con:
  - Too rigid
  - Too restrictive in parallelism
  - Results in too long assembly codes

Stack Architecture

- Up to 80s
- ALU operands are the two top locations of the stack
  - TOS: Top of Stack
  - The two operands are popped from the stack, and the result is pushed on the stack



- Data from memory can be loaded to TOS
- TOS can be stored in the memory



Stack Architecture

- One benefit: The compiler doesn't have to do variable-to-register allocation
  - This used to be a difficult problem and the stack architecture tries to circumvent it

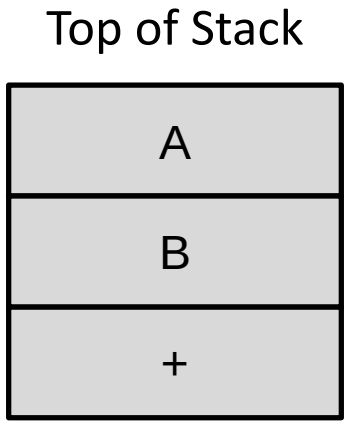
- There are two variants of the stack architecture:
- 1) The data and the operations are pushed on the stack
- 2) The data only is pushed on the stack; the operations come from the code

Stack Architecture

(Case 1) The data and the operations are pushed on the stack

- We want to evaluate this expression: **C = A + B**

- The stack is initialized as below

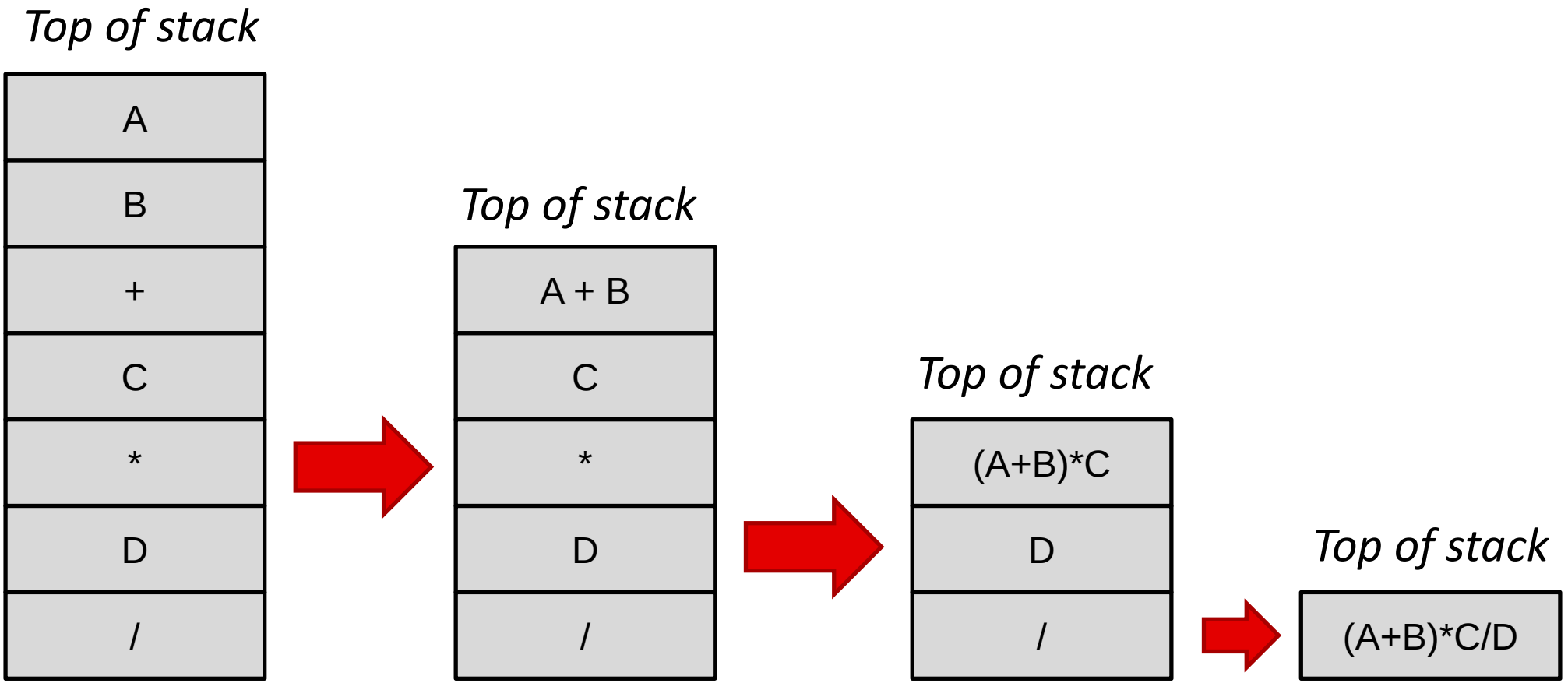


- Then, the two operands are popped; the operation is popped; they're added and the results is stored on the stack

Stack Architecture

- Initialization of the stack to compute the expression:

(A+B) * C/D



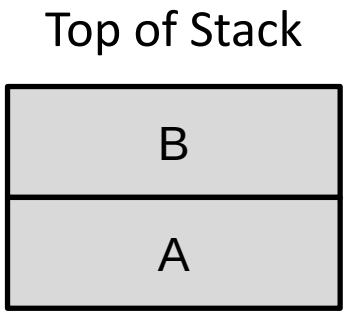
Stack Architecture

(Case 2) Only the data goes on the stack

- This is the expression we're evaluating: **C = A + B**

- The code pushes A and B from the memory onto the stack
- The 'Add' operation adds them
- Finally, the 'Pop' operation grabs the top of stack (the result of the addition) and stores it a the variable C in the memory

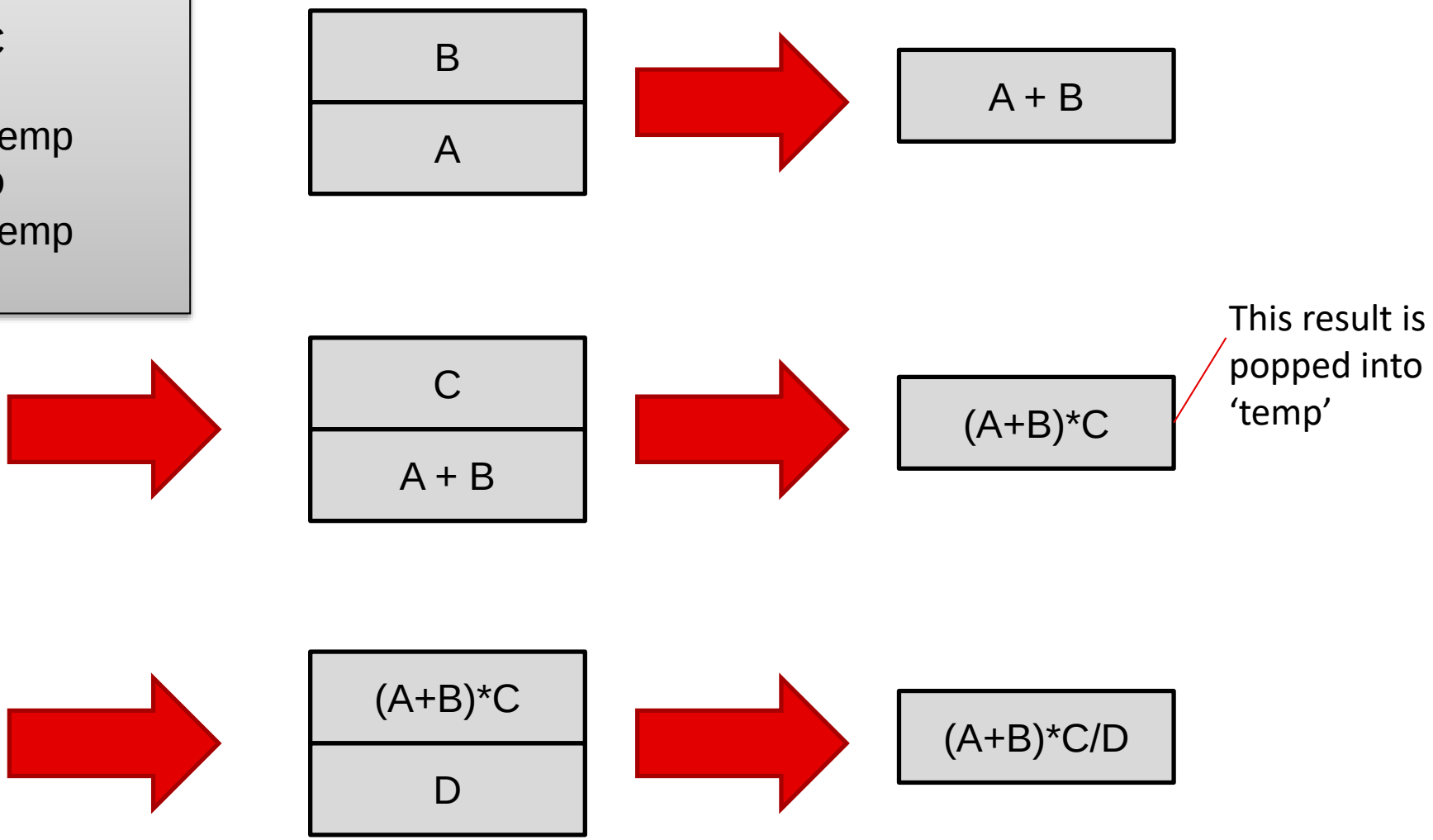
```
Push    A
Push    B
Add
Pop     C
```



Stack Architecture

```
Push     A
Push     B
Add
Push     C
Multiply
Pop      Temp
Push     D
Push     Temp
Divide
```

(A+B) * C/D



- *How can we rewrite this code without using the 'temp' variable?*

Stack Architecture

- Where is the stack located?
  - Usually located in the memory
  - Due to extensive usage, the top few words of the stack can be saved in registers for fast operation
- Not used in the modern CPUs
- Pro:
  - The compiler is simple
  - Simple hardware
- Con:
  - Too restrictive in parallelism
  - Results in too long assembly codes

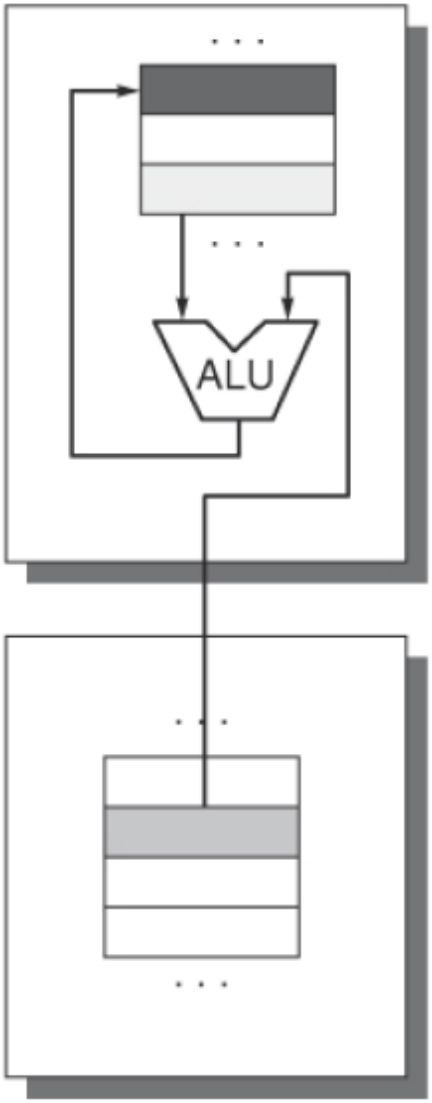
Memory-Memory Architecture

- Keeps all the data in the memory, no data is stored in registers
- Not used anymore in today's CPUs

- Pro: The compiler is simple
  - The variables don't have to be allocated to registers since they always reside in the memory
- Con: Too slow, every operation requires multiple memory accesses

Register-Memory Architecture

- Operates on data that's in the memory directly
  - ALU can have one operand as a register (top) and the other operand from the memory (bottom)
  - Some instructions may use two registers but not two memory locations
- There's no need to load the data from memory into a register beforehand



Register-Memory Architecture

- Two operands in the instruction (e.g., add eax, ebx) as opposed to MIPS which uses three operands (e.g., add t0, t1, t2)

- Possible instruction in from Intel x86

```
ADD   EAX, EBX     # add two registers; leftmost one takes
                   # the result → EAX = EAX + EBX

ADD   EAX, [400]   # add register EAX and data @ address 400
                   # result goes in EAX
```

- However, it's not possible to add two memory locations in one instruction (there's at most one memory address)

Register-Memory Architecture

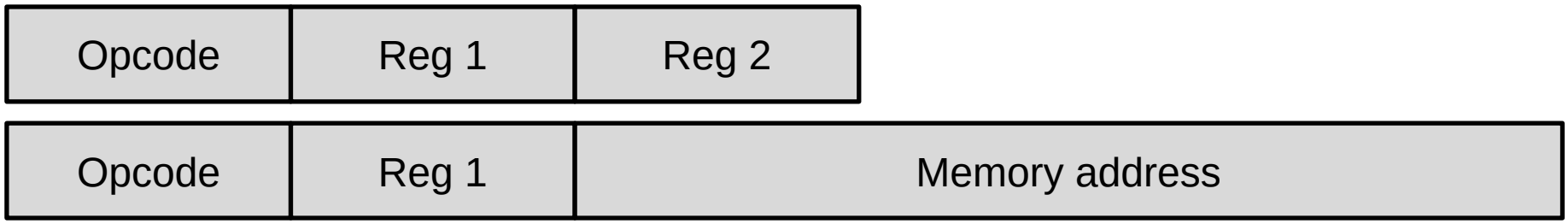
- A, B, C and D are located in the memory initially:

(A + B) * C/D

```
Load   R1, A        // copy A from memory into R1
Add    R1, B        // add R1 to B from the memory
Mul    R1, C        // multiply R1 by C from the memory
Div    R1, D        // divide R1 by D from the memory
```

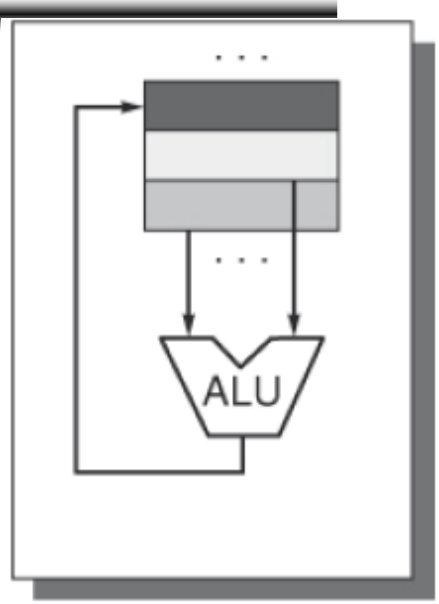
Register-Memory Architecture

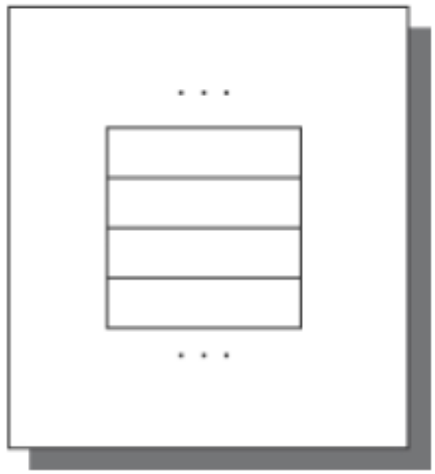
- Pro:
  - Data in the memory can be accessed directly, convenient for compiler, results in short code
- Con:
  - Not uniform instructions: Some instructions many clock cycles due to the memory access for fetching the operand in the memory
  - Instruction encoding is complex:



Load-Store Architecture

- Also called '**register-register architecture**'
  - Both ALU operands of the ALU are registers
  - ALU can't access a memory location directly
- To operate on a memory location
  - **Load**: Fetch the data into a register
  - **Calculate/operate** using ALU
  - **Store**: Transfer the result back to the memory location





Memory

Load-Store Architecture

- Usually three operands per instruction:
  - destination, source 1, source 2

- In MIPS architecture:

  add $t0, $t1, $t2

Load-Store Architecture

- A, B, C and D are initially in the memory:

(A + B) * C/D

```
Load   R1, A        // copy A from memory into R1
Load   R2, B        // copy B from memory into R2
Add    R3, R1, R2   // R3 contains A+B
Load   R1, C        // copy C from memory into R1
Mul    R3, R3, R1   // R3 now contains (A+B)*C
Load   R1, D        // copy D from memory into R1
Div    R3, R3, R1   // R3 contains the final result
```

Load-Store Architecture

- Pro:
  - Short and simple instruction encoding, the count of registers is small and require fewer bits to express them in the instructions:

  - Uniform (usually fixed-length) processing. Either a memory access (load or store) or register-only instruction

- Con:
  - Long code, every memory variable requires load and store instructions

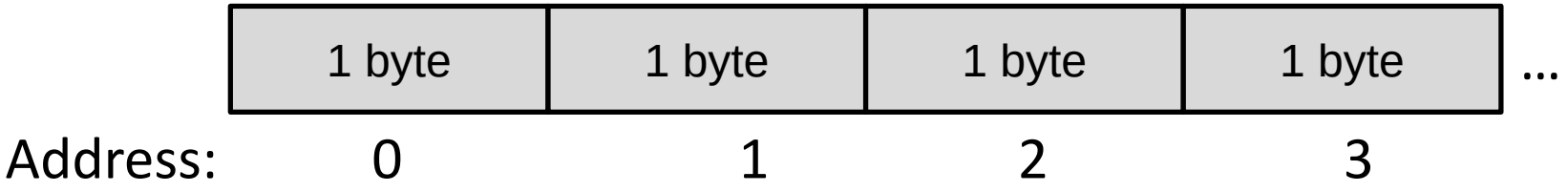
Types of ISA

Stack	Not used anymore
Accumulator	Not used anymore
Register-memory	
Load-store	
Memory-memory	Not used anymore

Memory Addressing

Memory Addressing

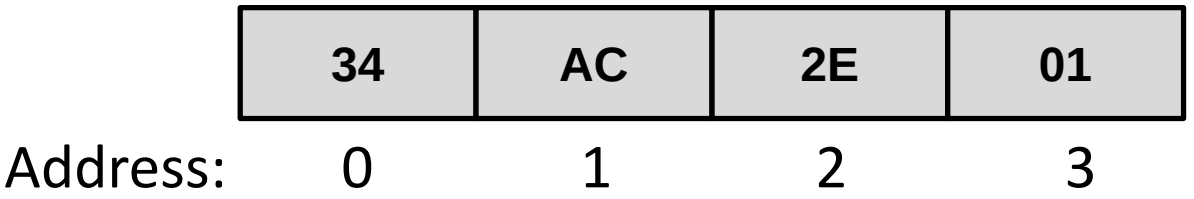
- Typical setup: ‘byte addressable’
  - Every byte has an address
- Increment by 1 refers to the next byte



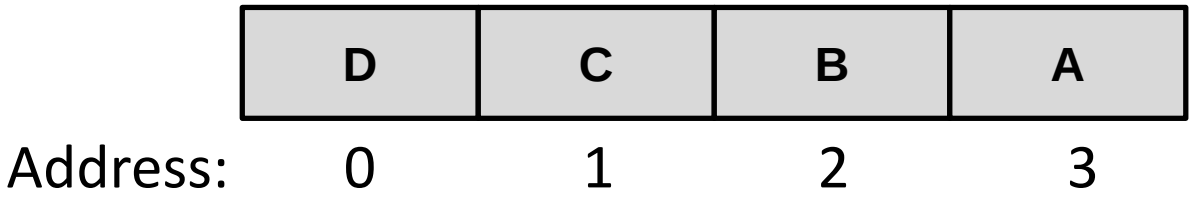
- Byte is too small to leverage spatial locality.
- Unit of operation: ‘word’
  - 16-bit computer: word is 2 bytes
  - 32-bit computer: word is 4 bytes
  - 64-bit computer: word is 8 bytes

Little Endian vs. Big Endian

- How to store a word in the memory?
- Little Endian: the data type ends at the 'little address'
- Example 1: The 32-bit hex number **01 2E AC 34** is represented in the memory as shown below

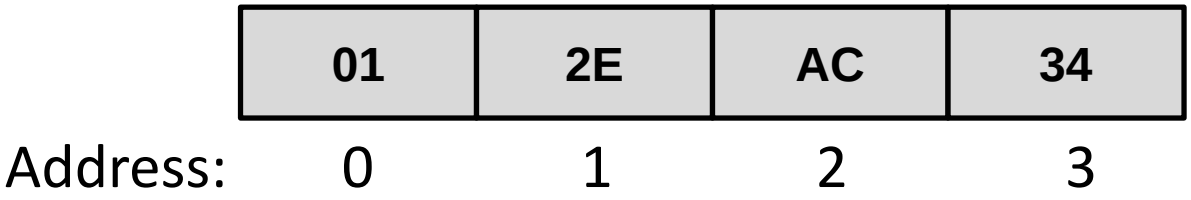


- Example 2: The string "ABCD" is represented as shown below; this is a bit of a disadvantage since the string is spelled in reverse order in the memory

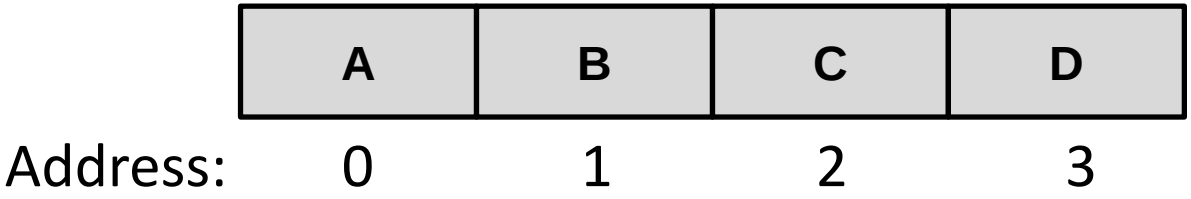


Little Endian vs. Big Endian

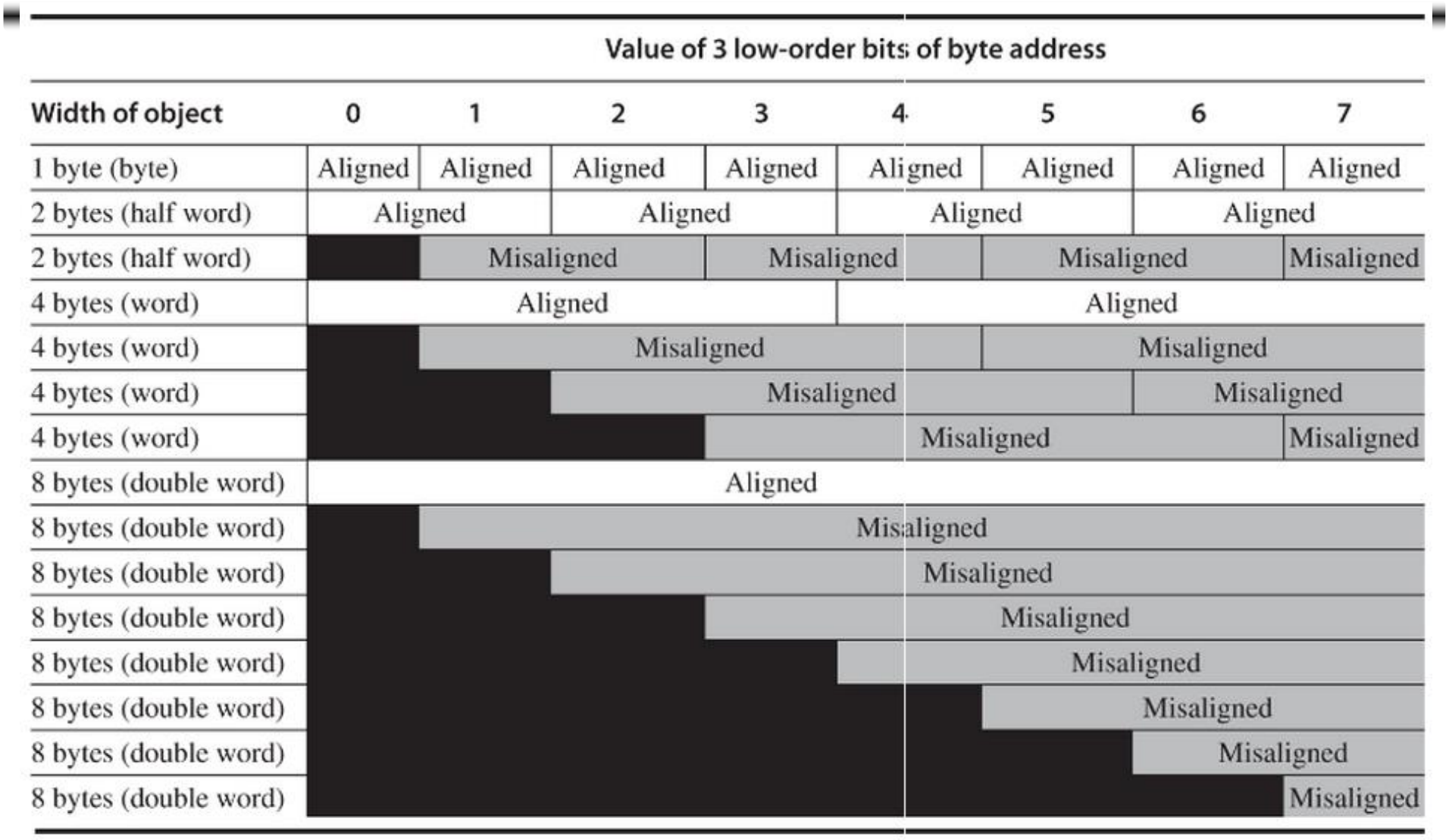
- Big Endian: the data type ends at the ‘big address’
- Example 1: The 32-bit hex number **01 2E AC 34** is represented in the memory as shown below



- Example 2: The string “ABCD” is represented as shown below
  - Big Endian is preferred here since the string is stored in the same order it’s read



Aligned and Misaligned Addresses



Width of object	Value of 3 low-order bits of byte address: 0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

A data type of size ‘n’ bytes stored at address ‘A’ is aligned if:

A modulo n = 0

Memory Alignment

- Misalignment causes multiple memory accesses to fetch a word!

- MIPS: memory is always aligned
  - Word = 4 bytes
  - Valid word addresses are: **0, 4, 8, 12…**
  - These addresses are multiples of 4 end in ‘00’ when they’re written in binary
  - Encoding and decoding of memory addresses are done with this knowledge

Addressing Mode

- Defines how an instruction specifies the address of its operands
- **Register** address
  - Usually consists of a few bits
  - With 8 registers on the CPU, 3 bits are sufficient
- **Memory** address
  - Usually many more bits than a register address
- **Immediate** number
  - There's no real address here, the constant value is encoded in the instruction

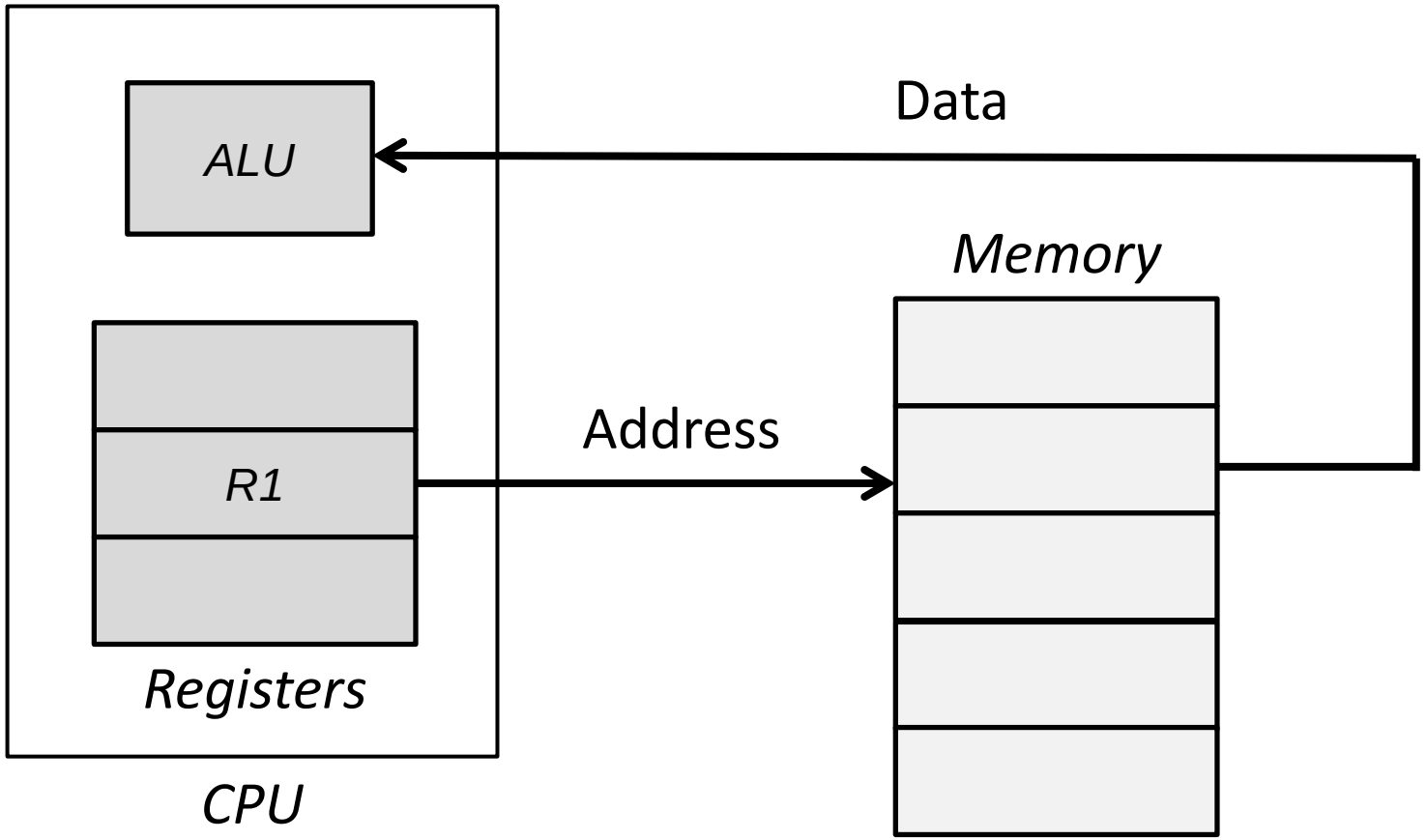
Popular Addressing Modes

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.
Register deferred or 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 or memory deferred	`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 machines.

Addressing Modes

Register indirect Example: Add R4, (R1)

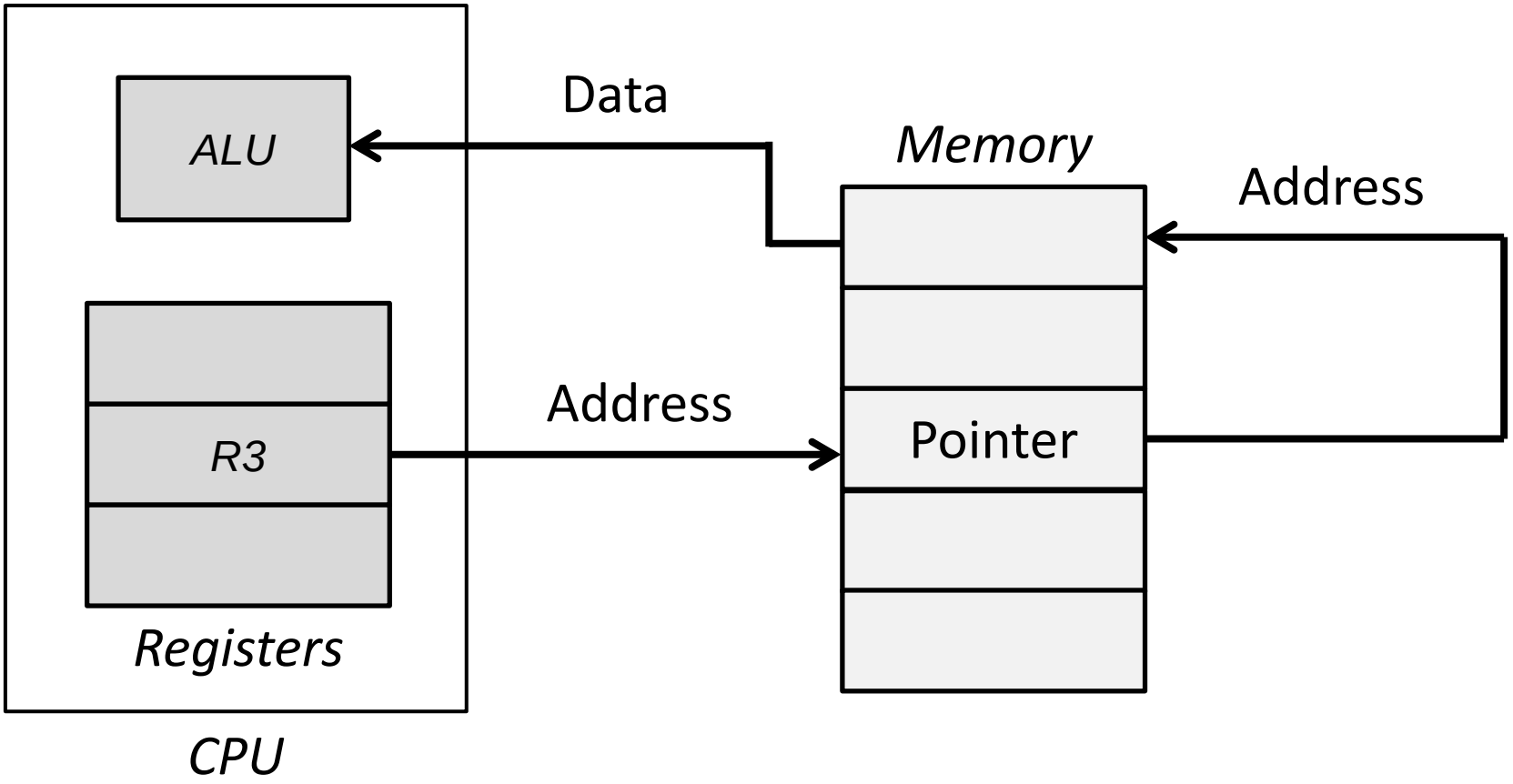
- R1 is the address in the memory
- 1 memory access



Addressing Modes

Memory indirect mode Example: Add R1, @(R3)

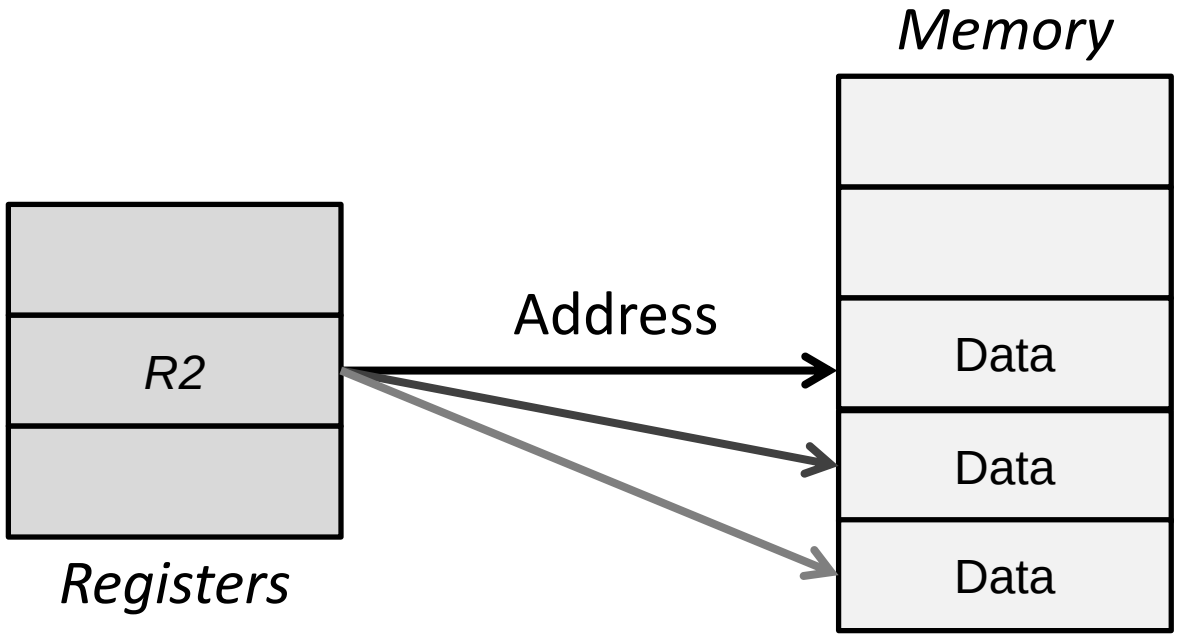
- R3 is the address of the pointer; once the pointer is read, another memory access fetches the data
- 2 memory accesses



Addressing Modes

Autoincrement Example: Add R1, (R2)+

- Used to access array elements in the memory
  - R2 is the address of the data in memory
  - When the data is fetched, R2 is incremented automatically so it's the address of the next array element



Addressing Modes

Scaled Example: Add R1, 100 (R2) [R3]

- This addressing mode is used to access data (array elements or data structures) from the memory
- First, let's look at the address of an array element in the memory

- The address of element A[y] is:

Start Address + (Element Size in bytes * y)

- For example, the address of A[3] = 200 + (4*3) = 212

Addressing Modes

Scaled Example: Add R1, 100 (R2) [R3]

- The address of the data in memory is:

100 + R2 + (R3 * scale)

- The scale is the size of the array element
  - If the array element is 4 bytes, then scale=4

- Let's consider this instruction: Add R1, 0 (R2) [R3]
  - Accesses Array[3]
  - We initialize: R2=400 (the start address of the array) and R3=3 (since we want to Array[3])
  - The address is: 0 + R2 + (4*R3) = 400 + 4*3 = 412

Addressing Modes

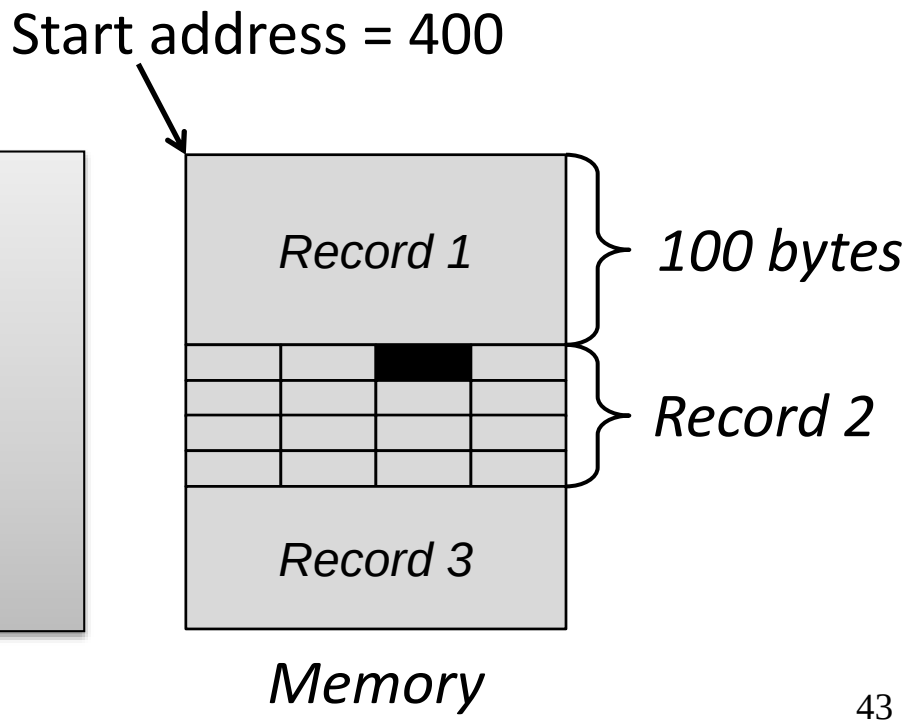
Scaled Example: Add R1, 100 (R2) [R3]

- Why does the scaled addressing mode contain a constant number?
- The constant number is used to skip a ‘record’ in a data structure

What’s the address of the ‘black box’ element?

It’s the element A[2] of Record 2

400 + 100 + 2*4 = 508



Addressing Modes

PC-Relative Addressing

- Used in ‘branch’ instructions, e.g.:

  branch R1, R2, Label

- A possible encoding:

- The ‘offset’ is added to the PC (Program Counter)
- Branch address is: **PC + Offset**

Addressing Modes: Simple vs. Complex

Simple Addressing Modes	Somewhere in the middle	Complex Addressing Modes
• Register `Add R4, R3`	• Displacement `Add R4, 100(R1)`	• Memory indirect `Add R1,@(R3)`
• Immediate `Add R4, #3`	• Register indirect `Add R4,(R1)`	• Autoincrement `Add R1,(R2)+`
• Direct `Add R1, (1001)`	• Indexed `Add R3,(R1+R2)`	• Autodecrement `Add R1,-(R2)`
	• PC-Relative `Address = PC+Offset`	• Scaled `Add R1,100(R2)[R3]`

Addressing Modes: Simple vs. Complex

Simple Addressing Modes	
Advantage	• Keep the hardware simple because the hardware implements the instructions • Keep the CPI (Clocks-per-instruction) small since the instruction does a small task
Disadvantage	• More instructions will be used because there's less flexibility in accessing data from the memory (additional instructions are used to compute memory addresses)

Addressing Modes: Simple vs. Complex

Complex Addressing Modes	
Advantage	• Reduce the instruction count since the instruction is 'powerful' in its ability to access data from the memory; this reduces the memory use
Disadvantage	• The hardware is complex since it implements the instructions' complex ways of access the memory • There will be a great variations between the number of clock cycles used by the instructions (instructions that use simple modes need few clock cycles; others that use the complex modes take more clock cycles); this variation makes it difficult to apply pipelining

Popular Addressing Modes

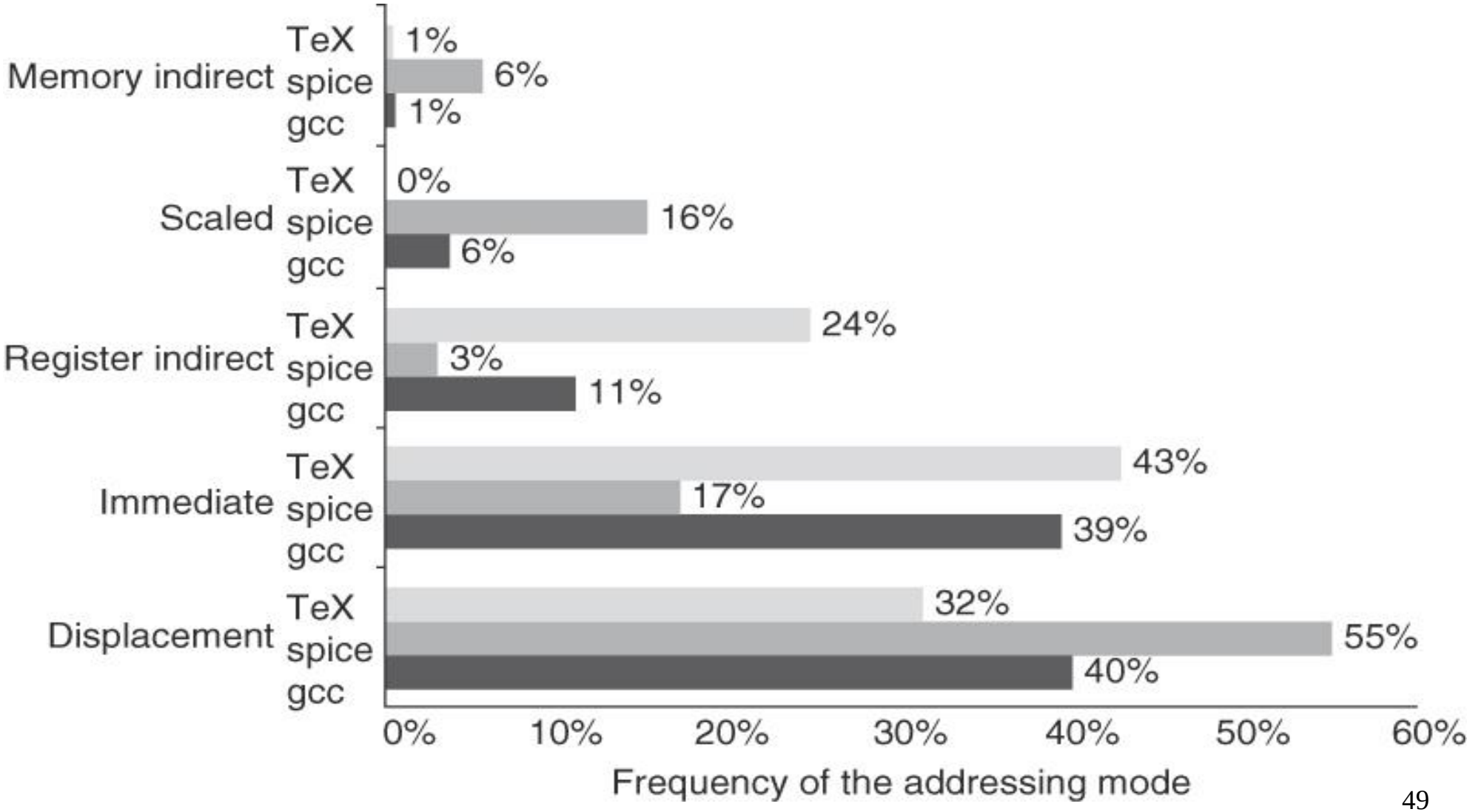
- Which modes are the most popular?
- One way: Measure the frequency of addressing modes in a typical program
  - What is a typical program?: Benchmarks, e.g., TeX, spice, gcc
  - Who picks the addressing mode or instructions?: The compiler!
  - Which computer architecture to use? One with a lot of addressing modes, e.g., VAX architecture.
  - The computer

The experiment setup

- The program is a fair program (benchmark)
- The computer supports a lot of addressing mode
- The compiler is unbiased; only aiming at high performance
- Due to these constraints, this study is considered a best attempt to get unbiased measurements as to which addressing mode is the most popular

Popular Addressing Modes

- It turns out that the **'immediate'** and **'displacement'** addressing modes are the most used ones



Popular Addressing Modes

- **Immediate** mode contains a constant: useful for simple arithmetic

  addi t0, zero, 4

- It's useful since it's used to load constants in a register

- **Displacement** mode has a base and offset: useful for array access

  lw t0, 12(s0)

  Used to access array elements

  A[4]=0; → sw zero, 12(s0)

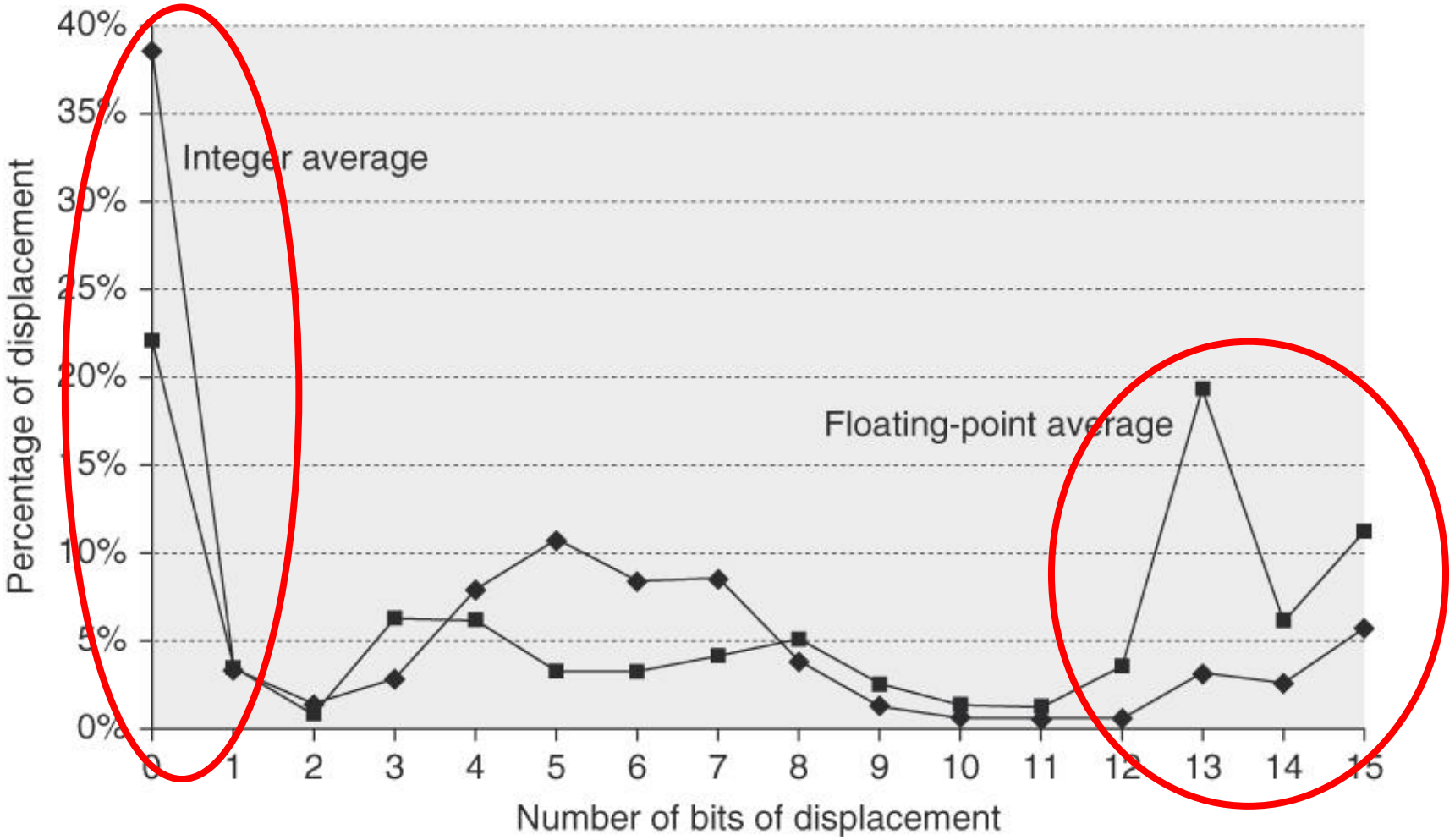
Displacement Field Size

- This instruction uses displacement mode:

Add R4, 100 (R1)

- A possible encoding:

Opcode	*R1*	*R4*	Displacement



Immediate Field

- **Terminology: Displacement vs. Immediate**
  - If the constant number is part of a memory address, it's called 'displacement'
  - If the constant number is loaded into a register or used in an arithmetic or logic operation, it's called 'immediate' field

- The constant number is a displacement:

Add R4, 100 (R1) // 100 is part of a memory address

- The constant number is an immediate:

Load R1, 200 // 200 is loaded in register R1 (no memory address)

Add R1, 300 // 300 is added to register R1 (no memory address)

Immediate Field

- How often are instructions with 'immediate' field used?
  - Immediates are used in 21% of integer instructions and 16% of floating-point instructions; they're quite useful

- 22% of the 'loads' load an immediate into a register (e.g., Load R1, 200)
- The remaining load from the memory

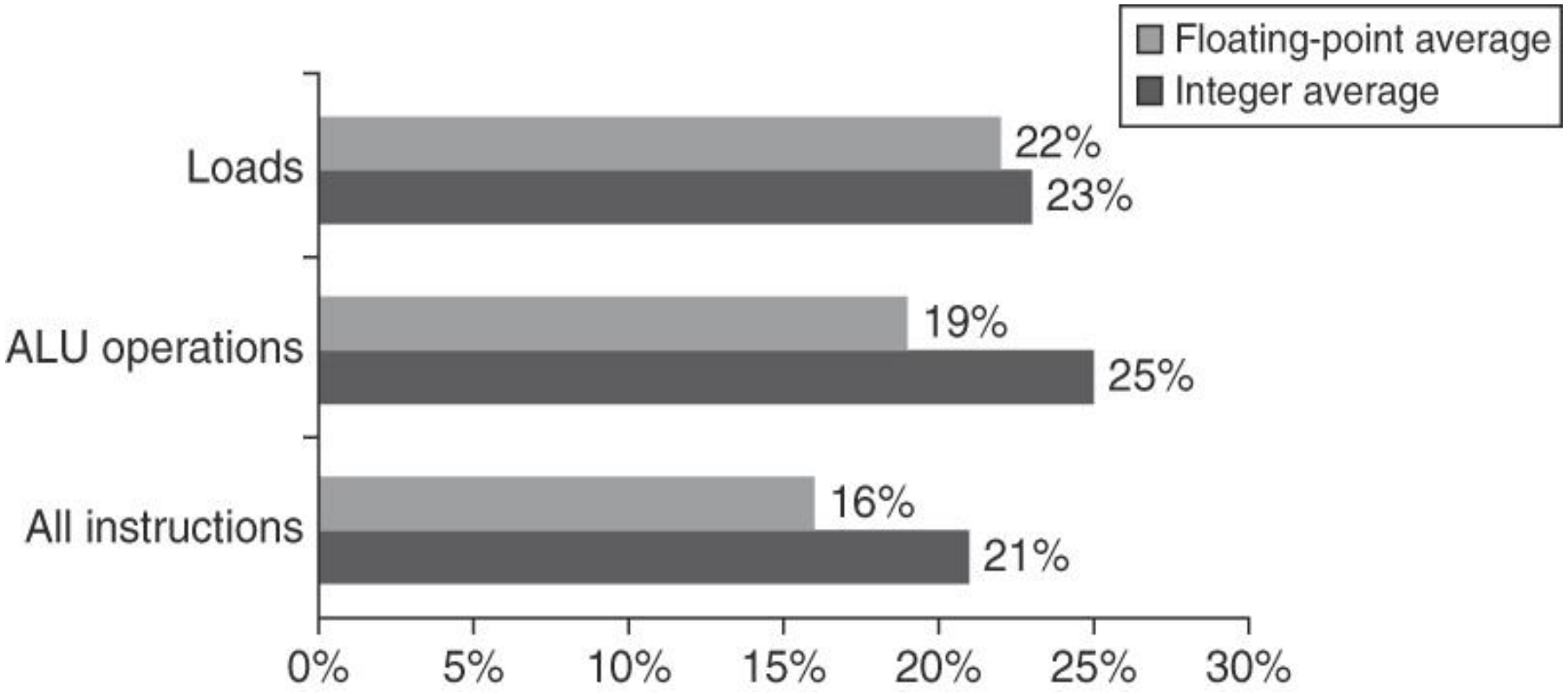


Figure A.9

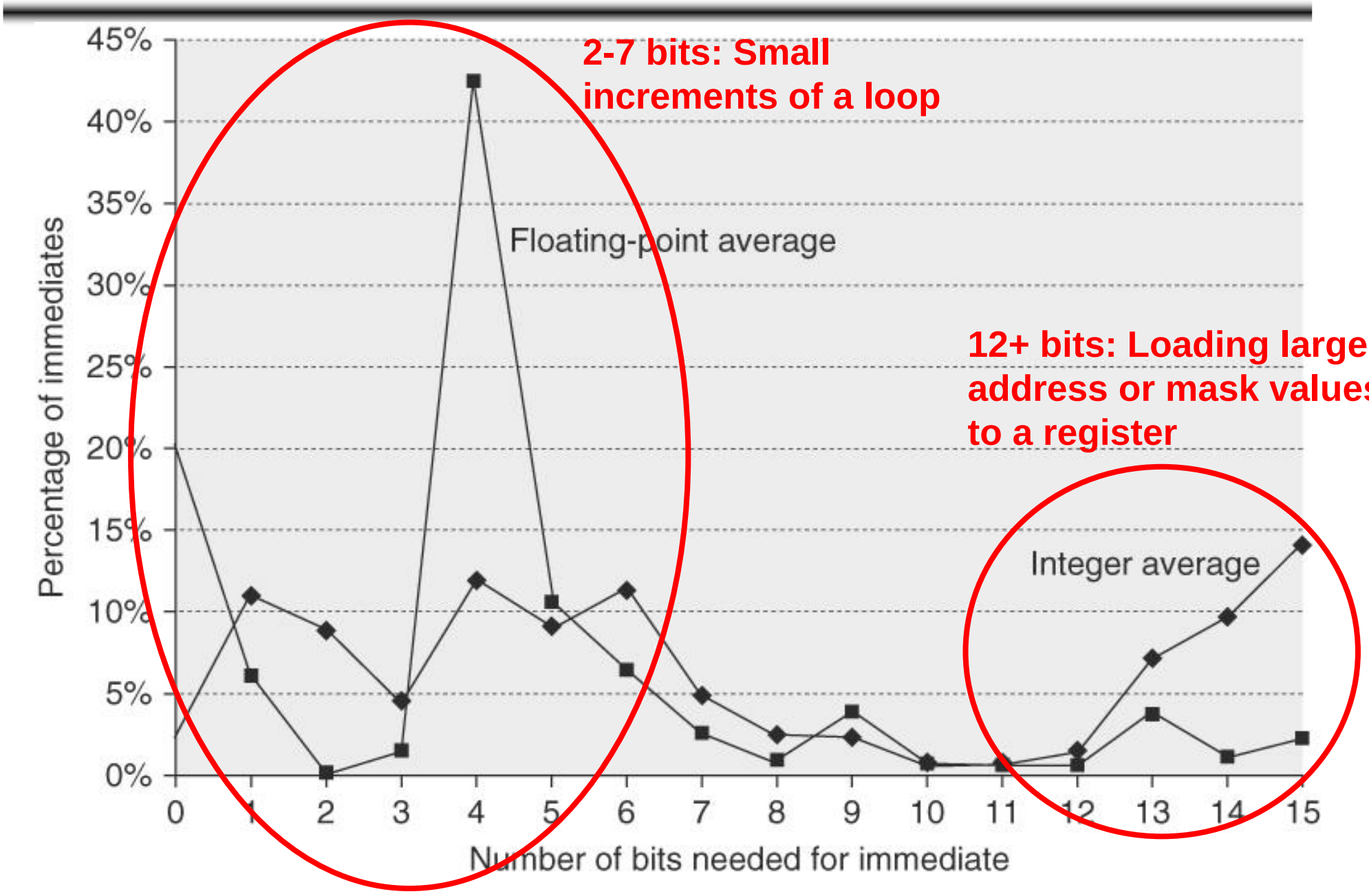
Immediate Field Size

- How many bits should the ‘immediate’ field be?
- Figure A.10 shows measurements taken on the Alpha computer; the immediate field supported is 16-bit

- Figure A.10 shows that, for integer instructions, small immediate values are quite useful (that use 6 bits or less) and large immediate values are useful (that use 13 bits or more)
  - The values in the middle, that use between 7 and 12 bits are not as frequent

- Why are the small and large values more useful than the intermediate ones?

Immediate Field Size



Immediate Field Size

- Other measurements were done on the VAX computer with immediate field size of 32 bits
  - 16-bit immediate: captures 75-80%
  - 8-bit immediate: captures 50%
- So, 16-bit immediate field size is good choice in a 32-bit architecture
- Do recent studies for 64-bit architectures show similar patterns but with larger number of bits?

Summary

- ISA types
- Addressing modes
  - Most popular: **‘immediate’, ‘displacement’** and **‘register indirect’**
  - Simple vs. complex modes
- Displacement: should be at least 12 to 16 bits
- Immediate: should be at least 8 to 16 bits

Readings

- H&P CA
  - App A
  - App K