Instruction Set Principles

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

Instruction Set Architecture

The *Instruction Set Architecture* (ISA) is how the computer is seen by the programmer or the compiler.

There are many alternatives possible for the ISA designer, any choice modifies the way to encode the instructions.

The different alternatives may be evaluated in terms of

- Processor performance
- Processor complexity
- Compiler complexity
- Code size
- Power consumption, …

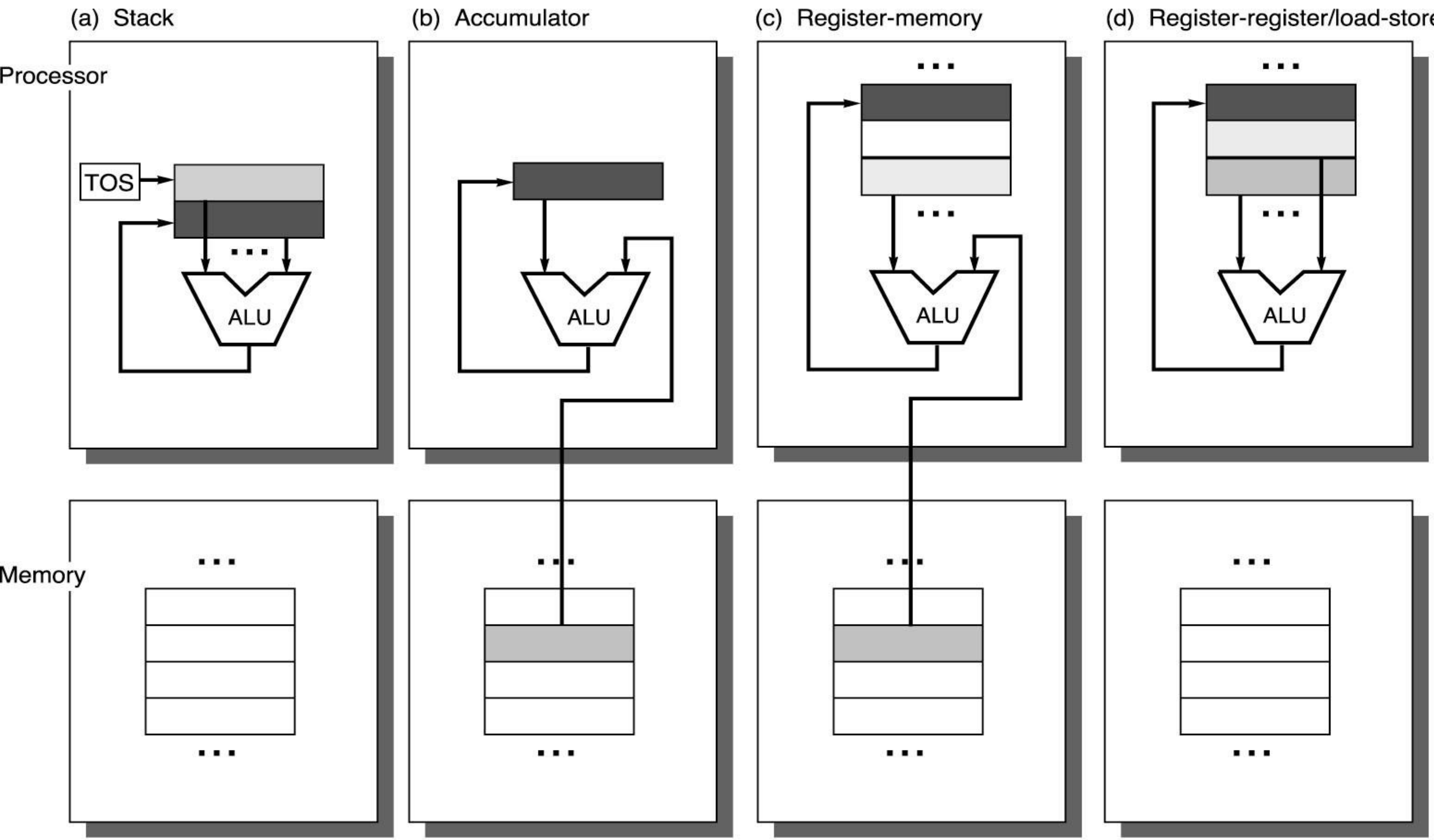
Different product areas may assign different weights to the above parameters.

Taxonomy

CPUs are often classified according to the type of their internal storage:

- **stack**
- **accumulator**
- **registers:**
  - **register-memory**
  - **register-register (load-store)**
  - **memory-memory (no real cases).**

Architectures



Code example

The code sequence 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`

GPR machines

Currently, all processors are *General-Purpose Register* (GPR) machines.

The reason for this is:

- registers are faster than memory
- registers are easier for a compiler to use.

Operands per ALU instruction

CPUs can be classified according to

- typical number of operands per ALU instruction (2 or 3)
- typical number of memory operands per ALU instruction (from 0 to 3).

Examples

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Instructions Set Characteristics

- **Memory addressing**
- Operations in the Instruction Set
- Type and Size of Operands
- Instruction Encoding.

Memory Addresses

Alternatives:

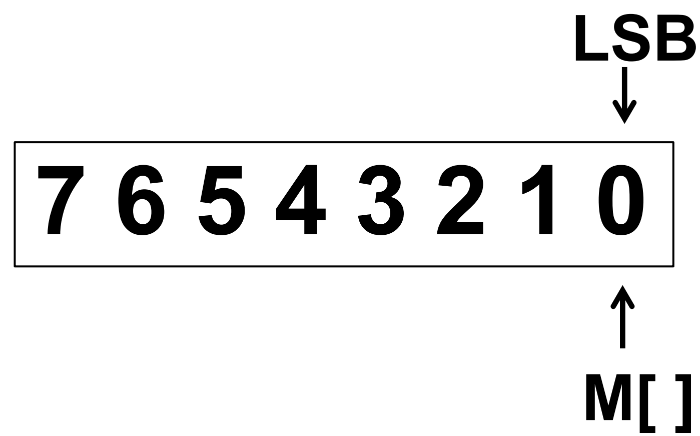
- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**

Memory Addresses

Alternatives:

- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**

It puts the byte with the lower address (X...X000) at the least significant position.
The address of the data is that of the least significant byte.



Memory Addresses

Alternatives:

- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**

It puts the byte with the lower address (X...X000) at the most significant position.
The address of the data is that of the most significant byte.

LSB

0	1	2	3	4	5	6	7

M[ ]

Memory Addresses

Alternatives:

- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**

Allowing only aligned accesses to memory is a limitation.

Memory Addresses

Alternatives:

- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**

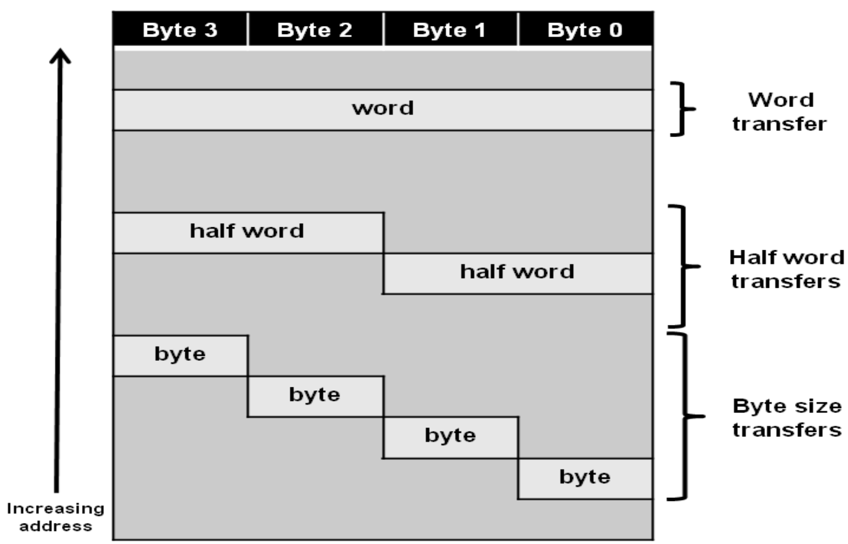
Allowing misaligned accesses to memory requires:

- **hardware overhead**
- **performance overhead.**

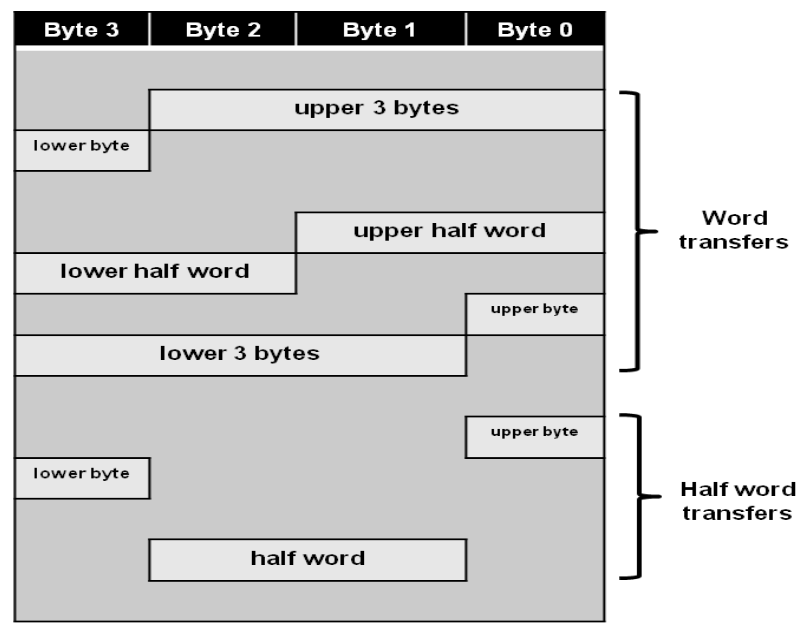
Memory Addresses

Alternatives:

- **Little Endian vs. Big Endian**
- **Aligned vs. misaligned accesses.**



Aligned access



Misaligned access

https://microchip.wdfiles.com/

Addressing modes

In GPR machines an addressing mode specifies a constant, a register, or a memory location (through its *effective address*).

Register Mode

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 Mode

Immediate	`Add R4,#3`	`Regs[R4]←Regs[R4]+3`	For constants.

Displacement Mode

Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4]+ Mem[100+Regs[R1]]	Accessing local variables.

Register Deferred (or Indirect) Mode

Register deferred or indirect	Add R4,(R1)	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	Accessing using a pointer or a computed address.

Indexed Mode

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) Mode

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 Mode

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*.

(post) autoincrement Mode

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.

(pre) autodecrement Mode

Auto-decrement	`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 Mode

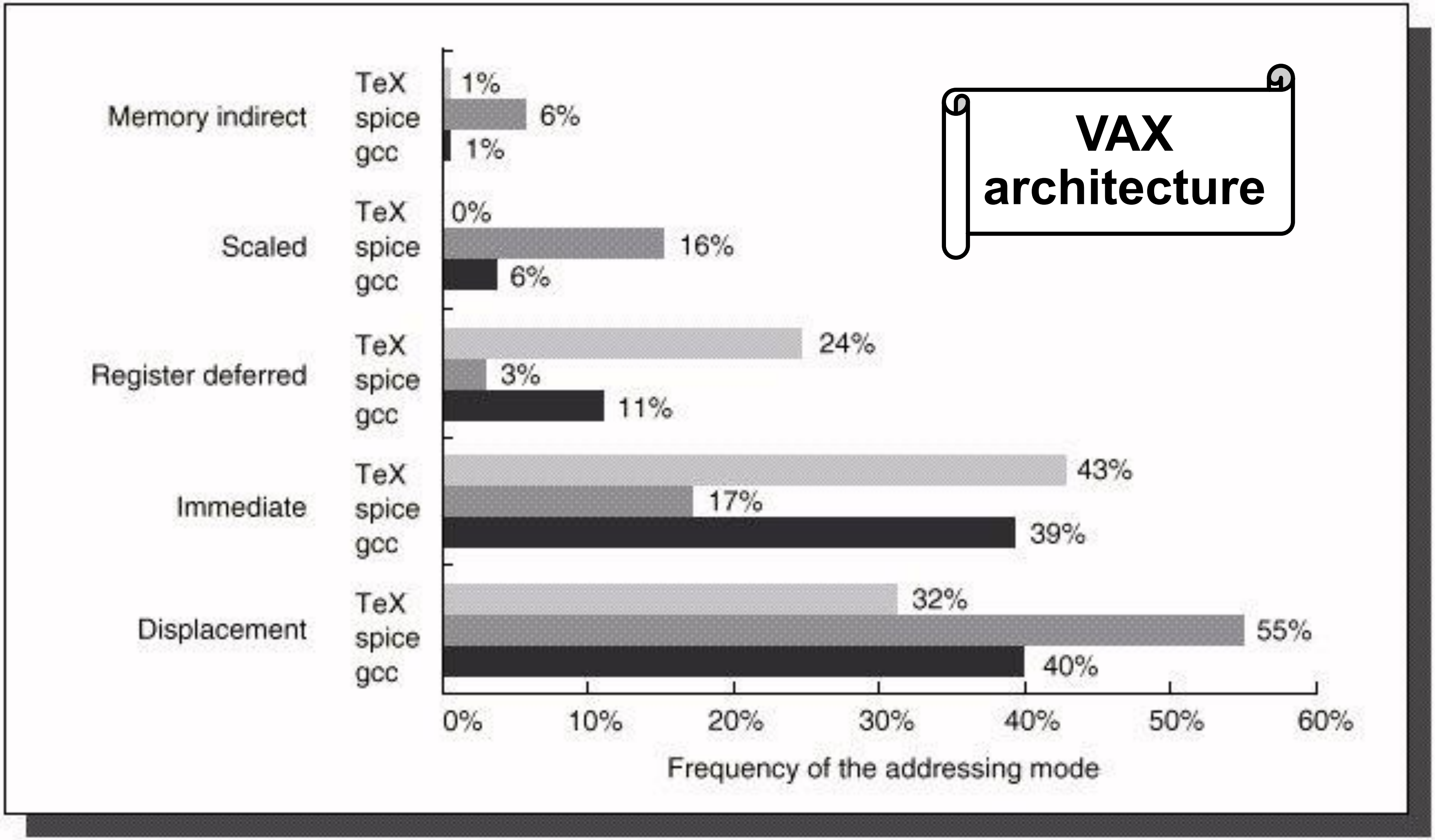
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1]+Mem[100+Regs[R2]+Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some machines.

Choosing the addressing modes

By carefully choosing the addressing modes, one can obtain some important consequences:

- Reducing the number of instructions
- Increasing the CPU architecture complexity
- Increasing the average Cycles Per Instruction (CPI).

Usage of addressing modes



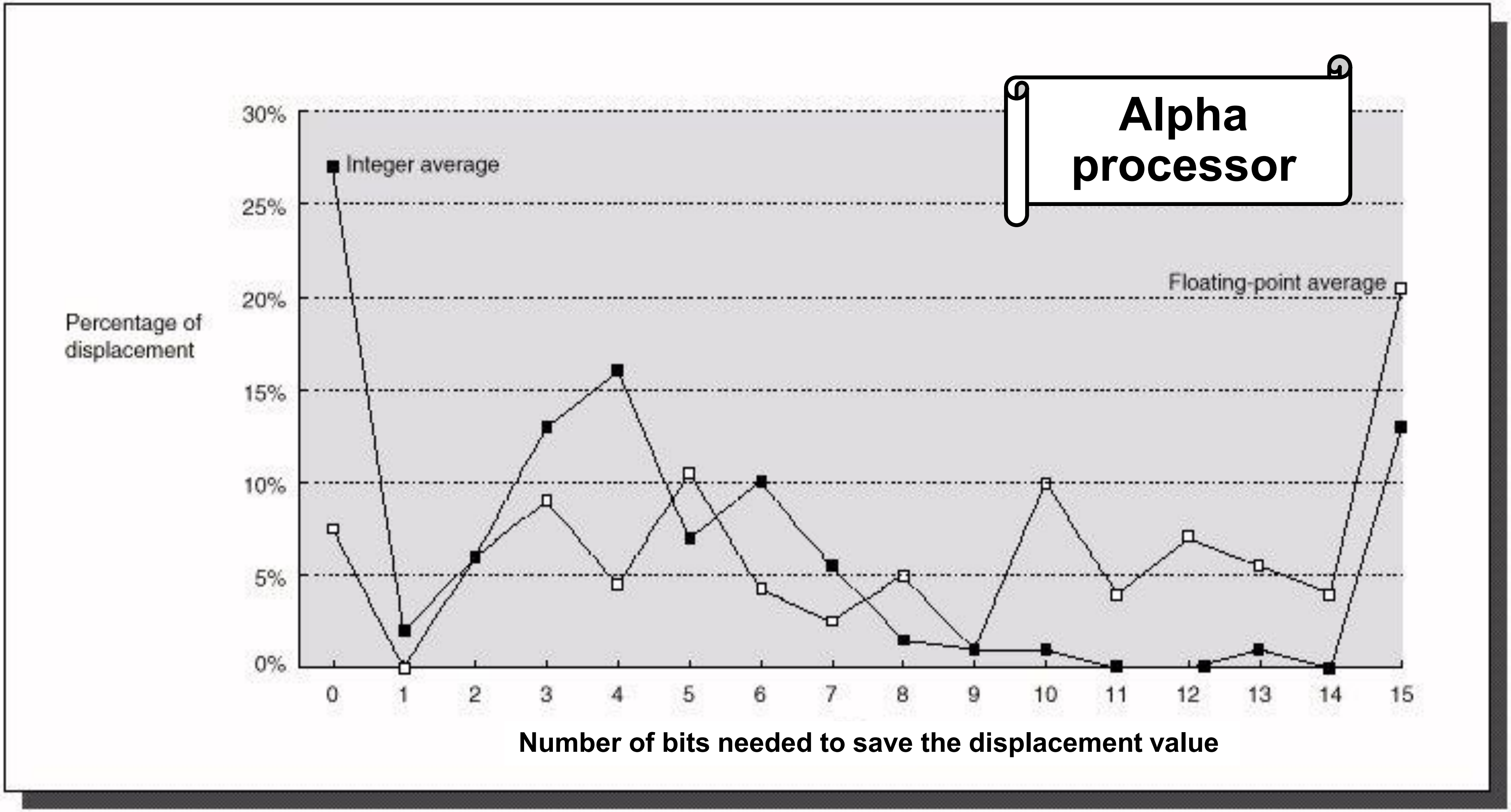
Open issue

When the selected addressing requires a displacement, how many bits should be devoted to it in the instruction code?

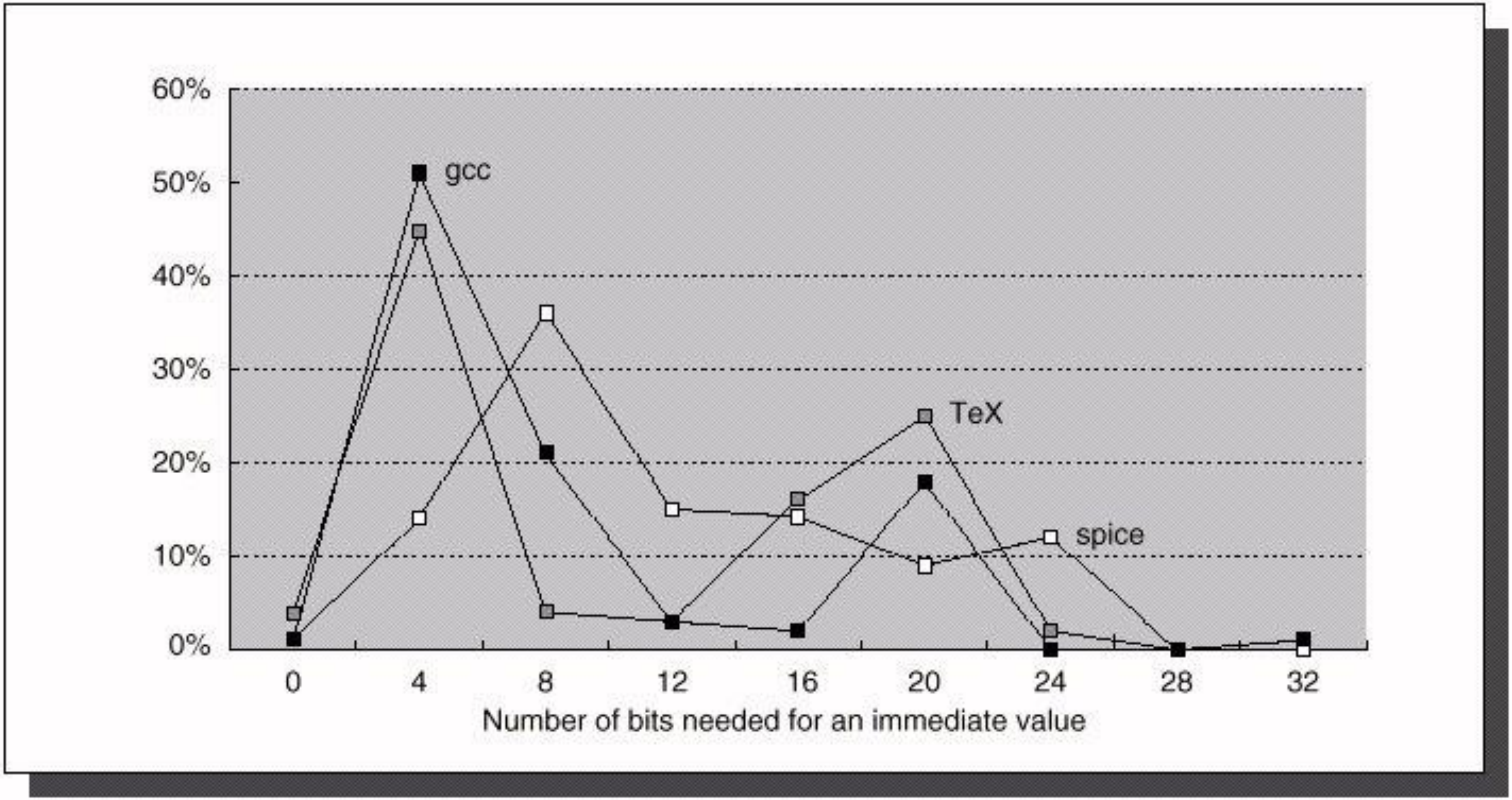
How large can be the immediate value, which is embedded in the addressing part of the instruction?

An experimental evaluation can be performed.

Displacement values



Immediate values



Summary

- Displacement, immediate and register indirect modes represent from 75% to 99% of the addressing modes
- The address size for displacement mode should be from 12 to 16 bits (75% to 99% of the displacements)
- The size of the immediate field should be at least 8 or 16 bits (50% and 80% of the cases, respectively).

Instructions Set Characteristics

- Memory addressing
- **Operations in the Instruction Set**
- Type and Size of Operands
- Instruction Encoding.

Operations in the Instruction Set

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads-stores (move instructions on machines 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
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel operations, compression/decompression operations

Making the Common Case Fast

Not all the instructions are executed with the same frequency!

When designing and implementing an instruction set, the most commonly executed instructions should be made faster.

80x86 instruction frequency

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%

RISC-V instruction frequency

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%
AVG≈	24,58%	11,00%	17,67%	2,67%	42,58%

SPECint2006

Control Flow Instructions

They can be distinguished in four categories:

- conditional branches
- jumps
- procedure calls
- procedure returns.

Conditional branches are by far the most frequently executed control flow instructions.

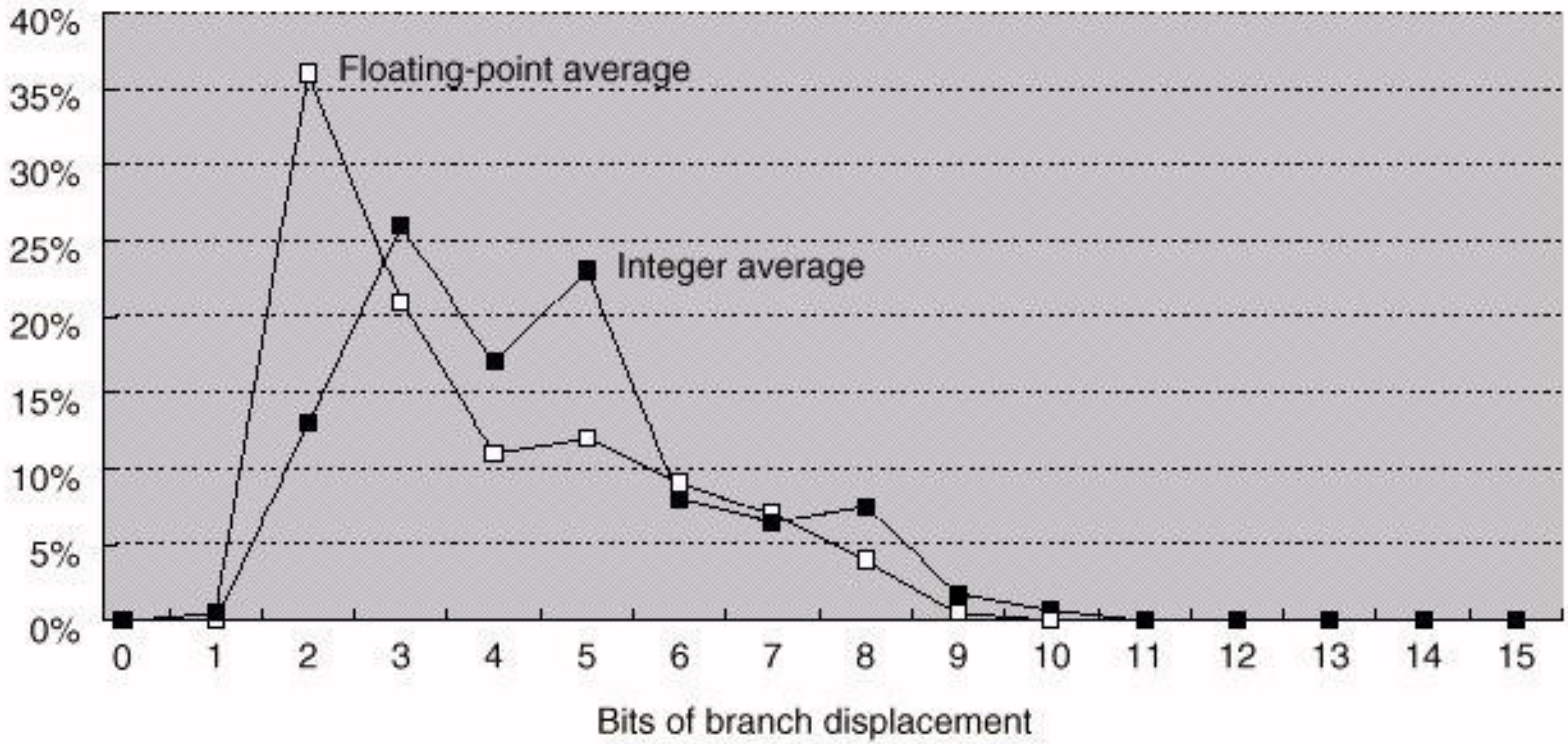
The destination address

It is normally specified as a displacement with respect to the current value of the Program Counter.

In this way:

- we save bits, since the target instruction is often close to the source one
- the code is position-independent.

Branch distances



Register Indirect Jumps and Procedure Calls

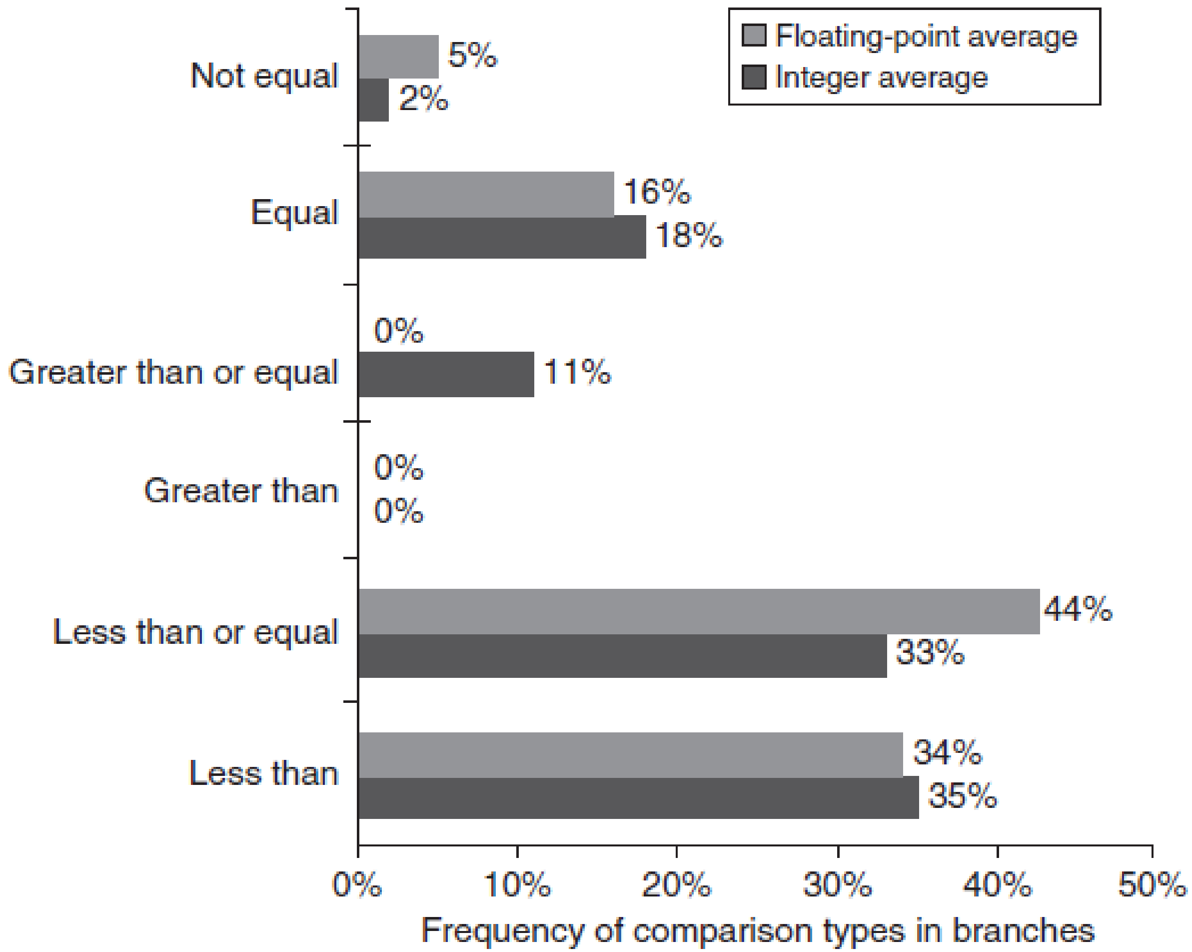
They allow:

- **to write code including jumps whose target is not known at compile time**
- **to implement *case* or *switch* statements**
- **to support *dynamically shared libraries* (i.e., libraries which are loaded only when called)**
- **to support *virtual functions* (i.e., calling different functions depending on the data type).**

Evaluating Branch Conditions

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 instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Conditional branches



Procedures

Some information need to be saved:

- the return address
- the accessed registers:
  - caller saving
  - callee saving.

Summary

- Few instructions are responsible for most of the execution time: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch, jump, call, and return.
- PC-relative branch displacements of at least 8 bits are the best choice.
- Register-indirect and PC-relative addressing can also be used in procedure call and return.

Instructions Set Characteristics

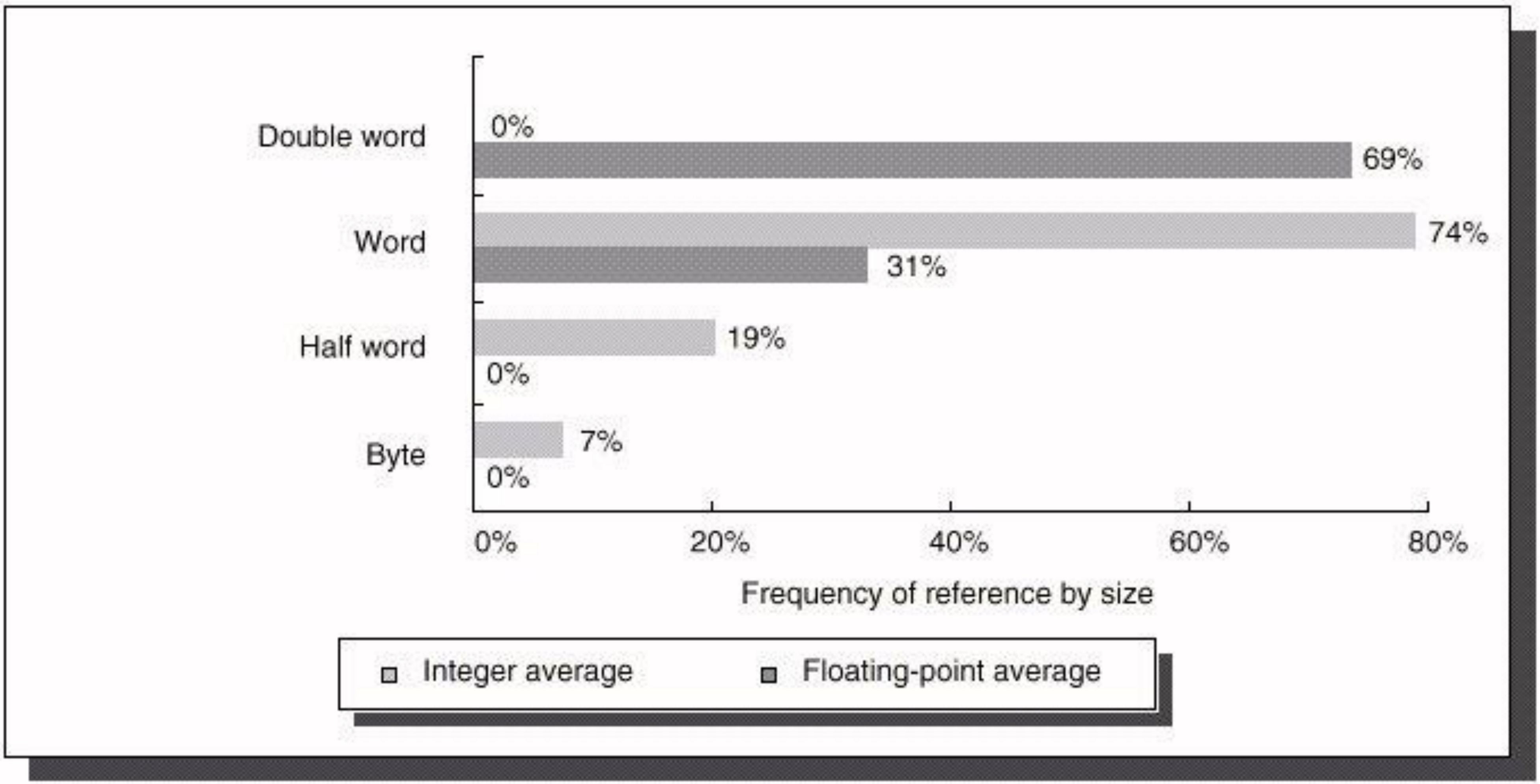
- Memory addressing
- Operations in the Instruction Set
- **Type and Size of Operands**
- Instruction Encoding.

Type and Size of Operands

Most frequently supported data types:

- **char (1 byte)**
- **half word (2 bytes)**
- **word (4 bytes)**
- **double word (8 bytes)**
- **single-precision floating-point (4 bytes)**
- **double-precision floating-point (8 bytes).**

Distribution of Data Accesses by Size



Instructions Set Characteristics

- Memory addressing
- Operations in the Instruction Set
- Type and Size of Operands
- **Instruction Encoding.**

Instruction Set Encoding

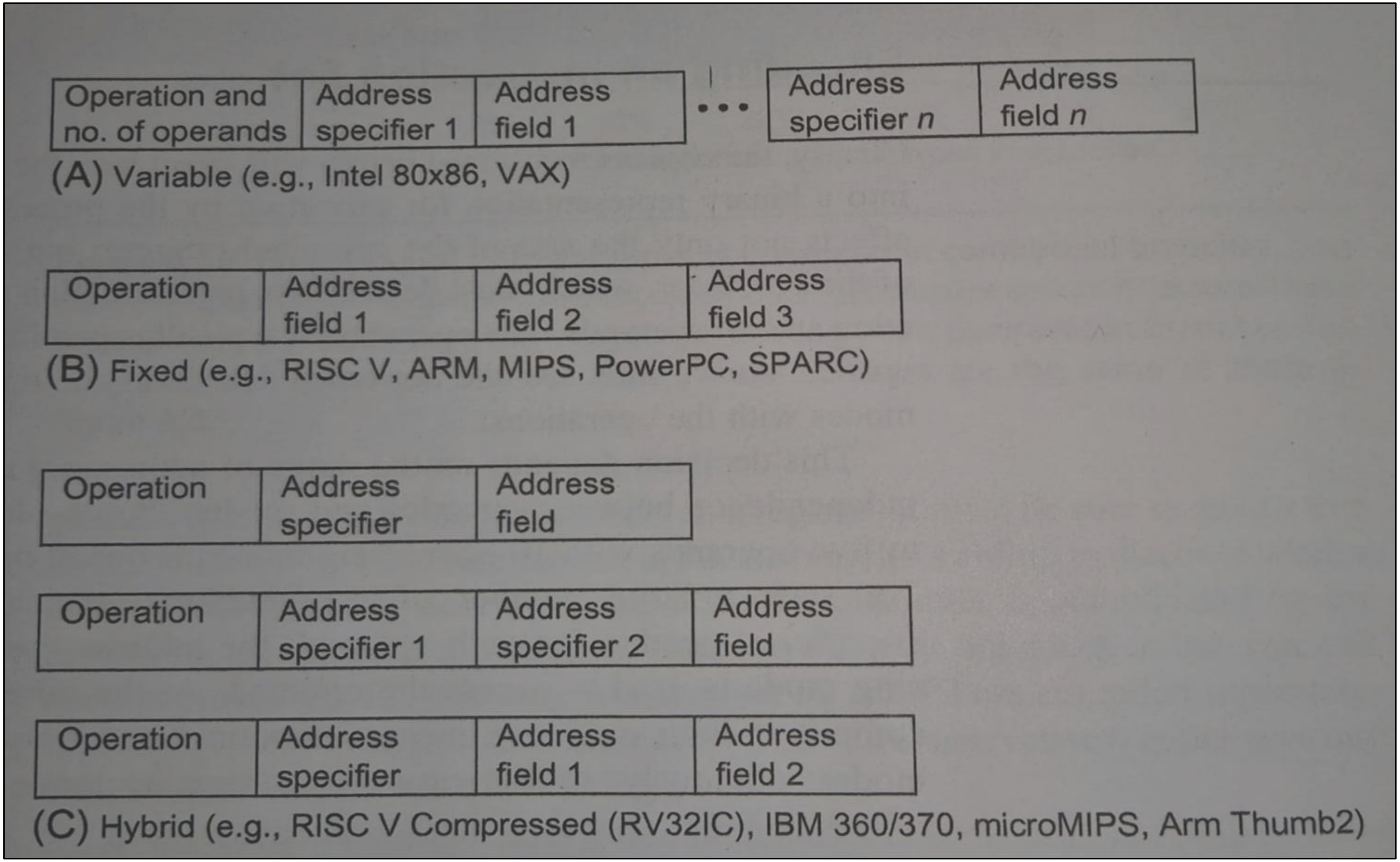
Instruction Set Encoding depends on

- which instructions compose the Instruction Set
- which addressing modes are supported.

When a high number of addressing modes is supported, an *address specifier* field is used to specify the addressing mode and the registers which are possibly involved.

When the number of addressing modes is low, they can be encoded together with the opcode.

Instruction Set Encoding

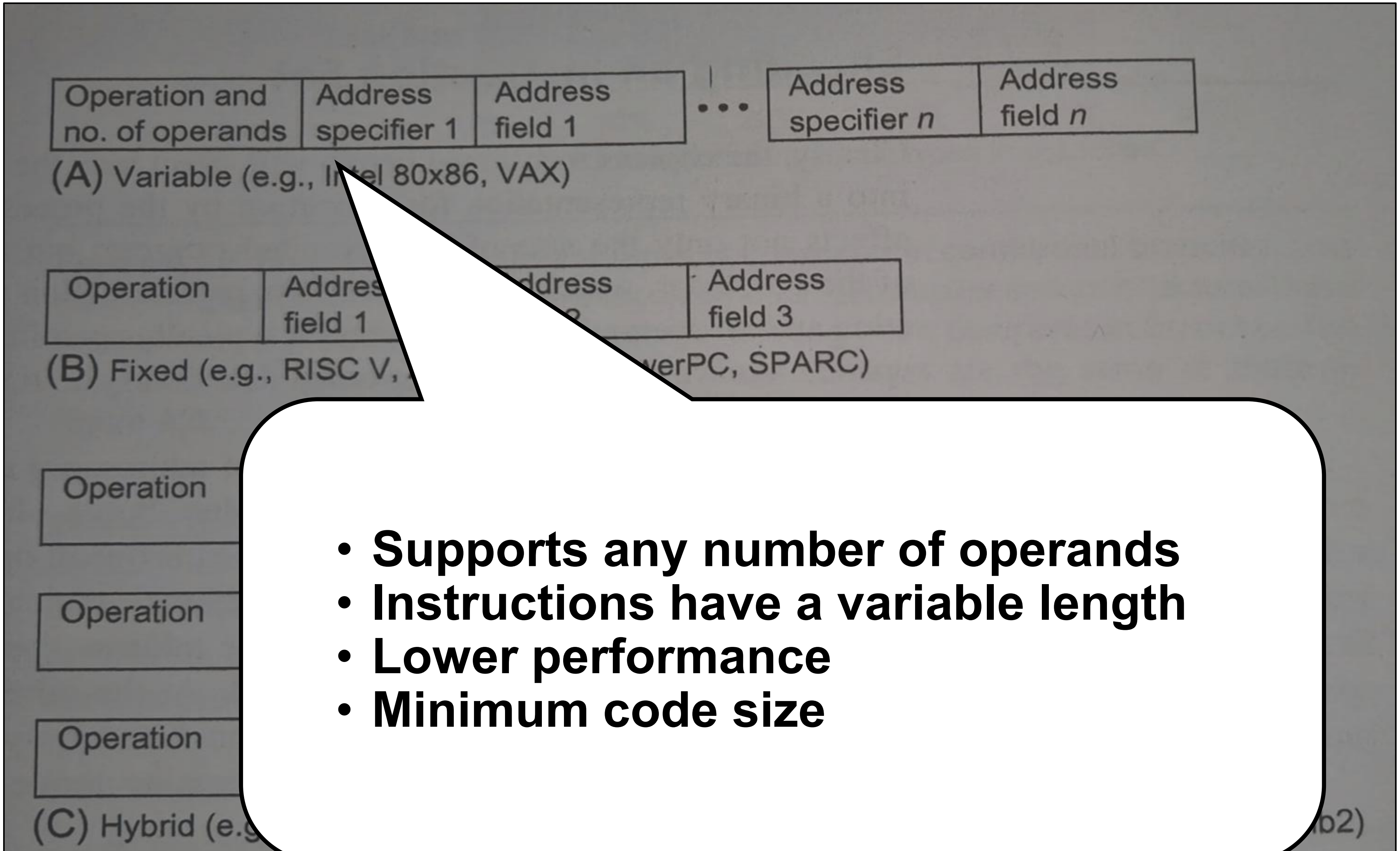


(A) Variable (e.g., Intel 80x86, VAX)

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)

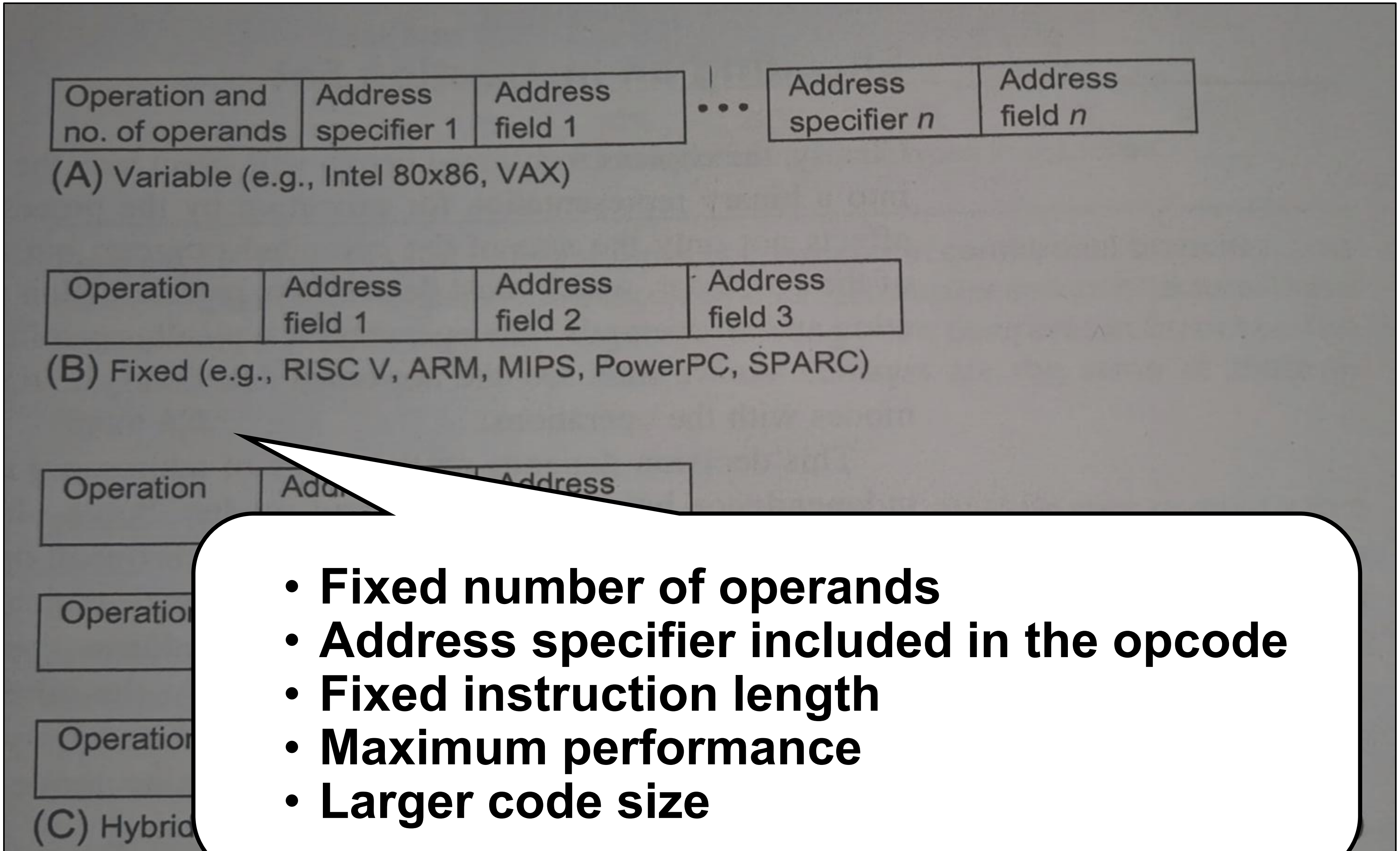
(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Instruction Set Encoding



- **Supports any number of operands**
- **Instructions have a variable length**
- **Lower performance**
- **Minimum code size**

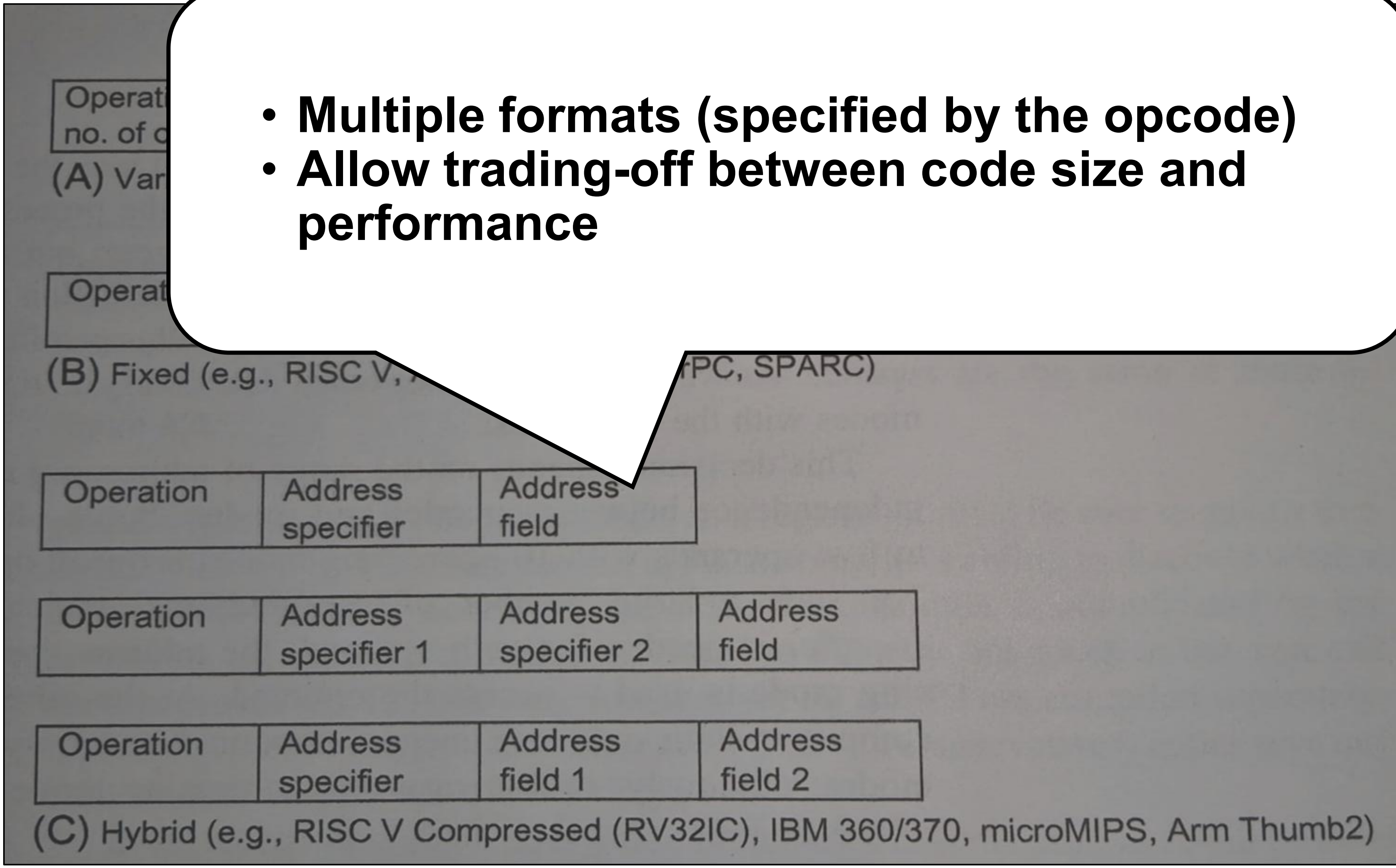
Instruction Set Encoding



- **Fixed number of operands**
- **Address specifier included in the opcode**
- **Fixed instruction length**
- **Maximum performance**
- **Larger code size**

Instruction Set Encoding

- **Multiple formats (specified by the opcode)**
- **Allow trading-off between code size and performance**



Conflicting Issues

The designer should address several conflicting issues:

- the code size
- the size of the Instruction Set, the number of addressing modes, and the number of registers
- the complexity of the fetch and decoding hardware.

Hardware-Compiler interaction

- **Assembly-level programs are now produced by compilers, only**
- **The CPU designer and the compiler writer must interact and cooperate.**

Register allocation

Choosing which variables have to be put in which registers and when, is one of the crucial optimization phases in a compiler.

This problem is based on *graph* coloring and can be better solved if the number of registers is high (>16).

Variables access

Optimizing variable access time by allocating variables to registers is only possible for those stored in the stack or for global variables in memory.

It is impossible for variables belonging to the heap, due to the *aliasing problem* (i.e., the access to a variable is done through pointers).

How the Architect can Help the Compiler Writer

Make the frequent case fast and the rare case correct

- **Regularity**
- **Provide primitives, not solutions**
- **Simplify trade-offs among alternatives**
- **Provide instructions that bind the quantities known at compile time as constants.**

Recommendations

- At least 16 registers
- Orthogonality
- Simplicity.