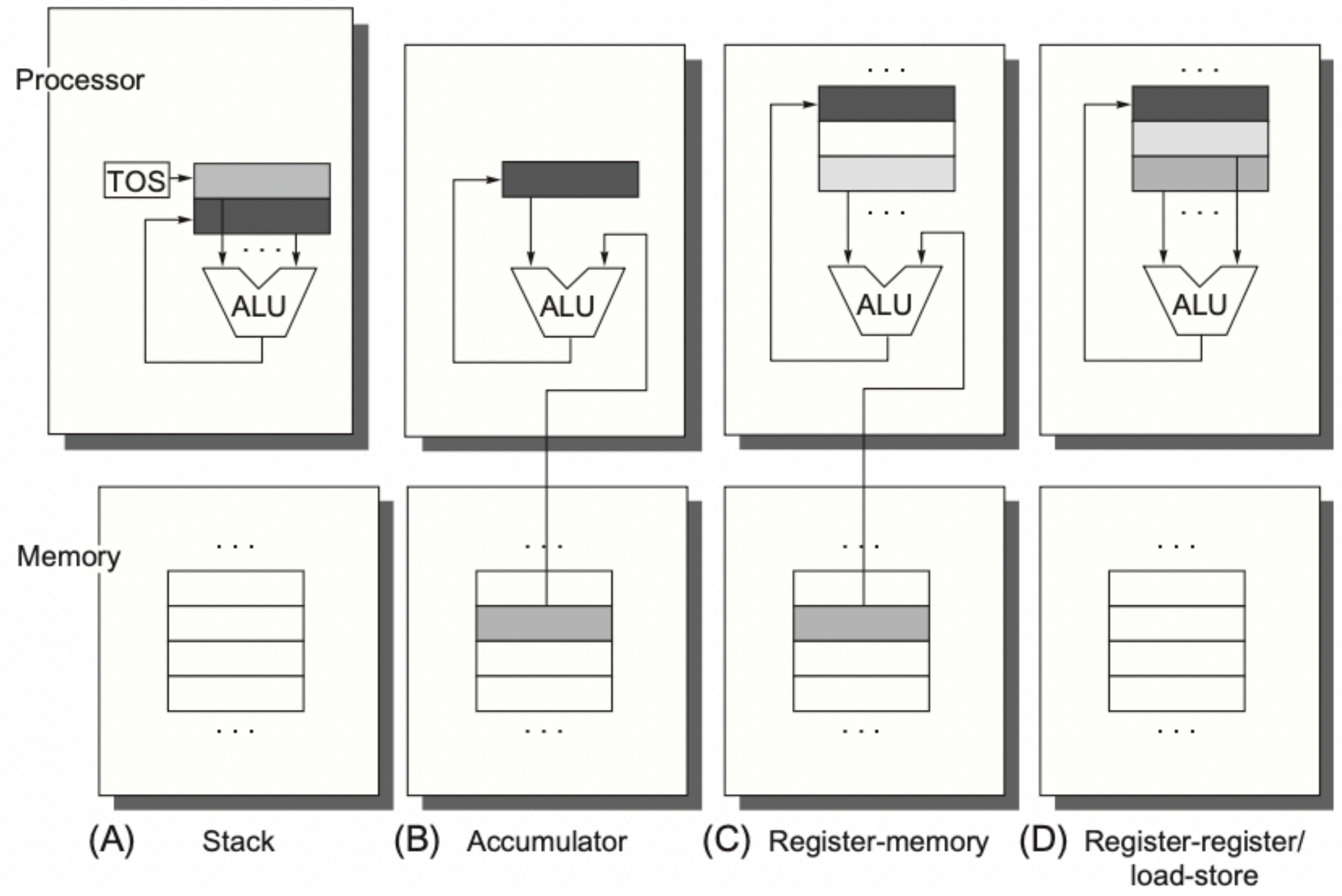
Computer Architecture

Appendix A



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

Figure A.2 The code sequence for C = A + B for four classes of instruction sets. Note

Architecture Characteristics

- Number of Operands
- Types of Operands
- Memory Addressing
- Addressing Modes
- Operations
- Data Types
- Branch specifications
- Instruction Encoding

Number and Type of Operands

- 0 (stack)
- 1 (stack, accumulator)
- 2 (register-register, register-memory)
- 3 (load-store)
- Register reference
- Memory reference

Memory addressing

- Little endian

7	6	5	4	3	2	1	0

- Big endian

0	1	2	3	4	5	6	7

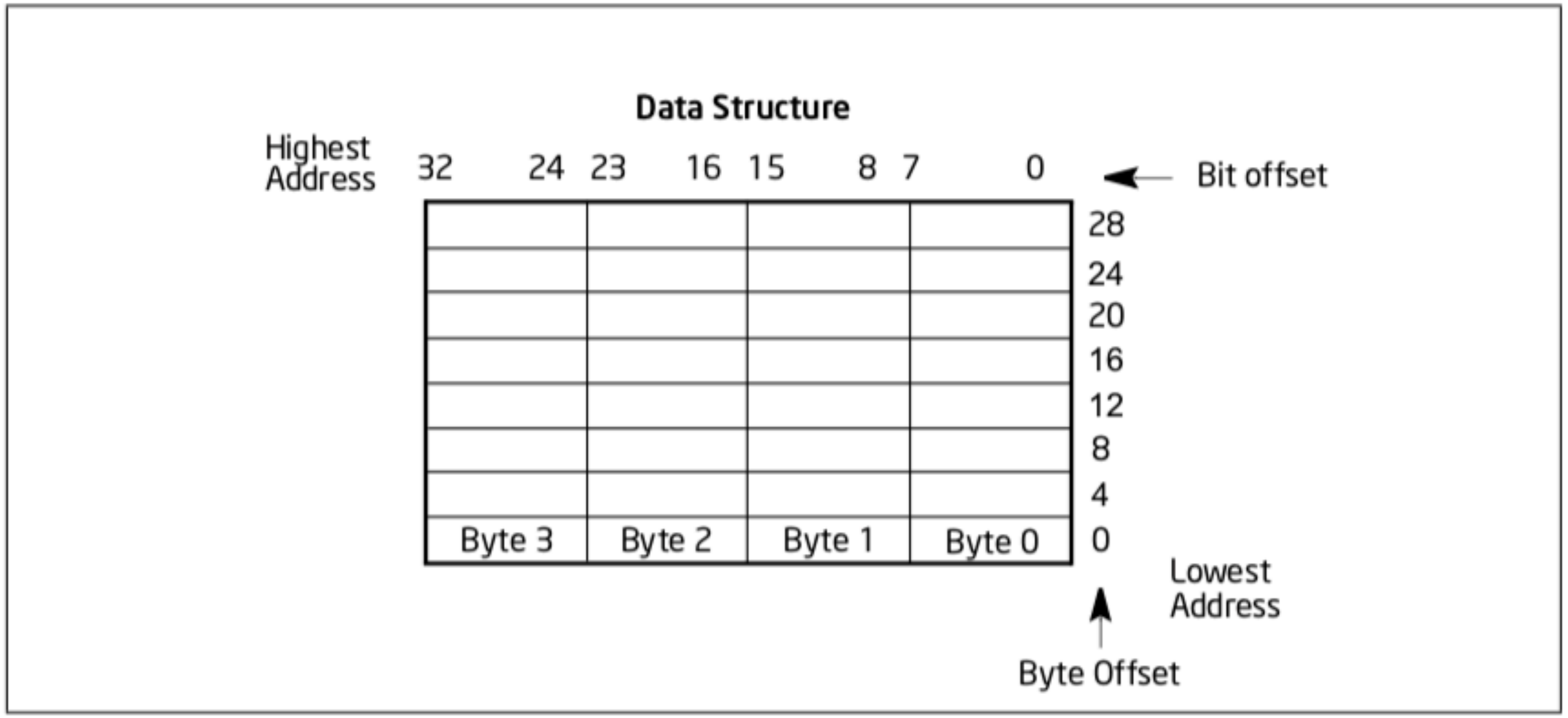
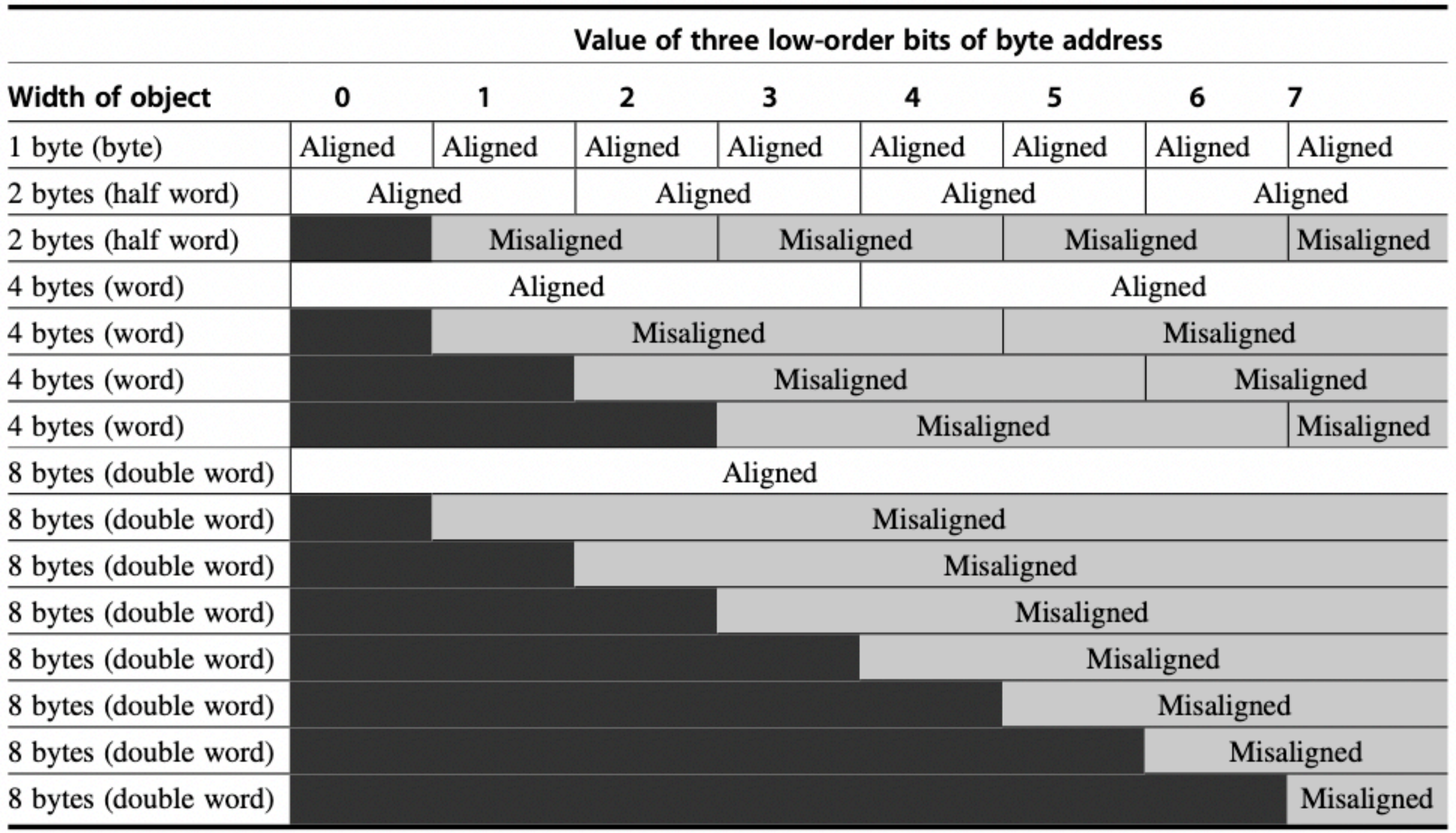


Figure 1-1. Bit and Byte Order

From Intel IA-64 Manual

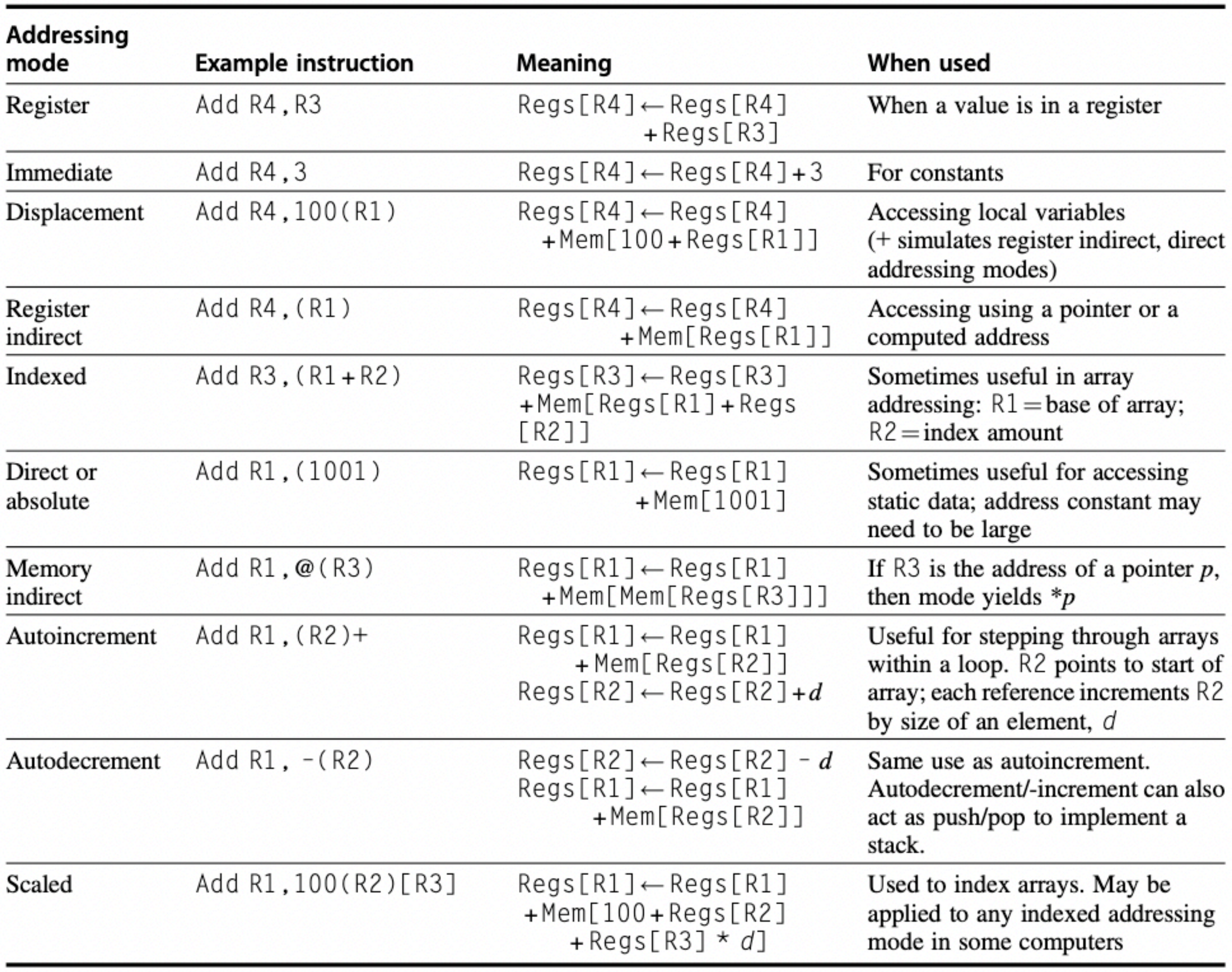
Memory Alignment



	Value of three low-order bits of byte address							
Width of object	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

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order three bits of the address.

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 (+ 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 mode usage

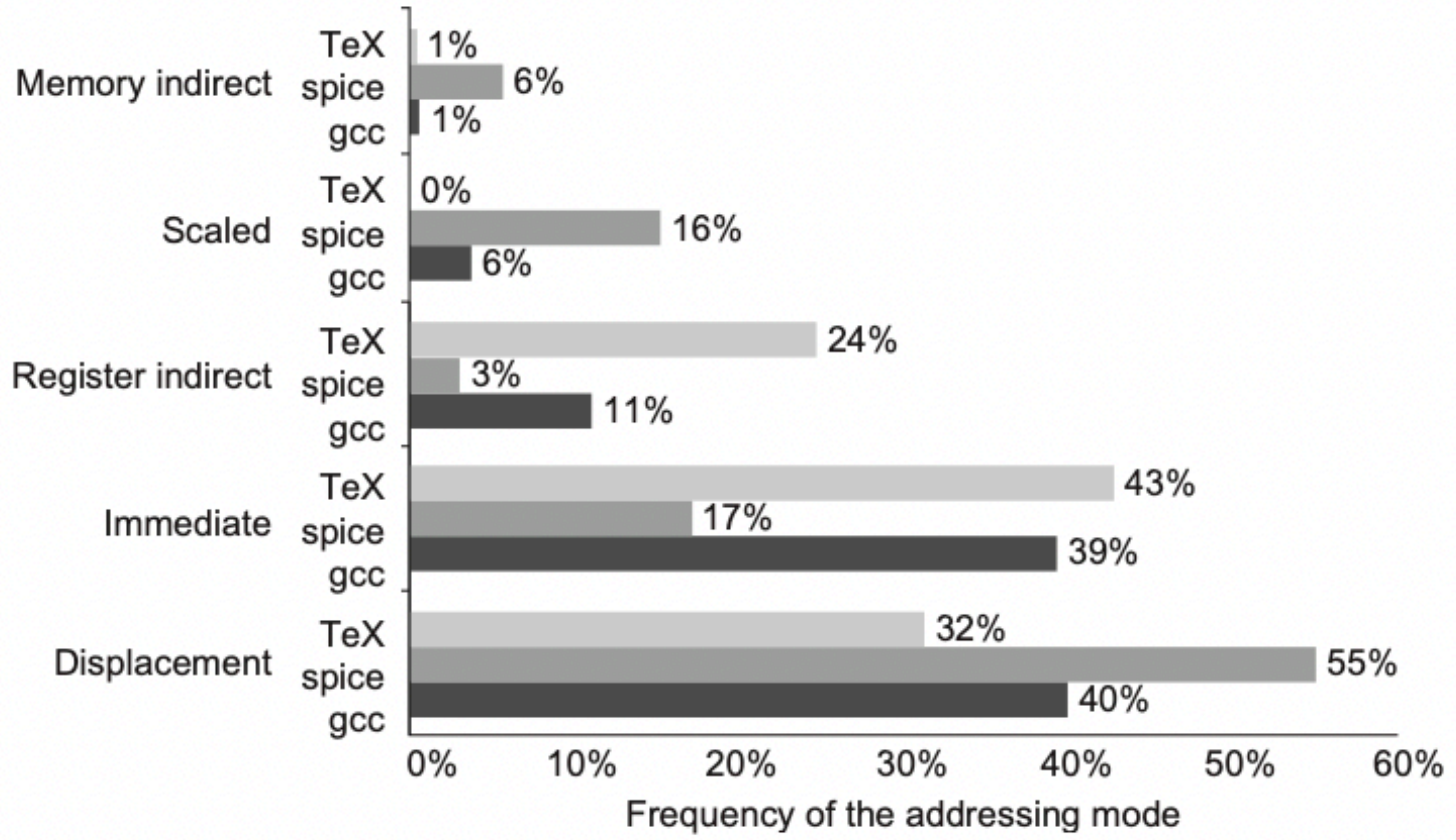
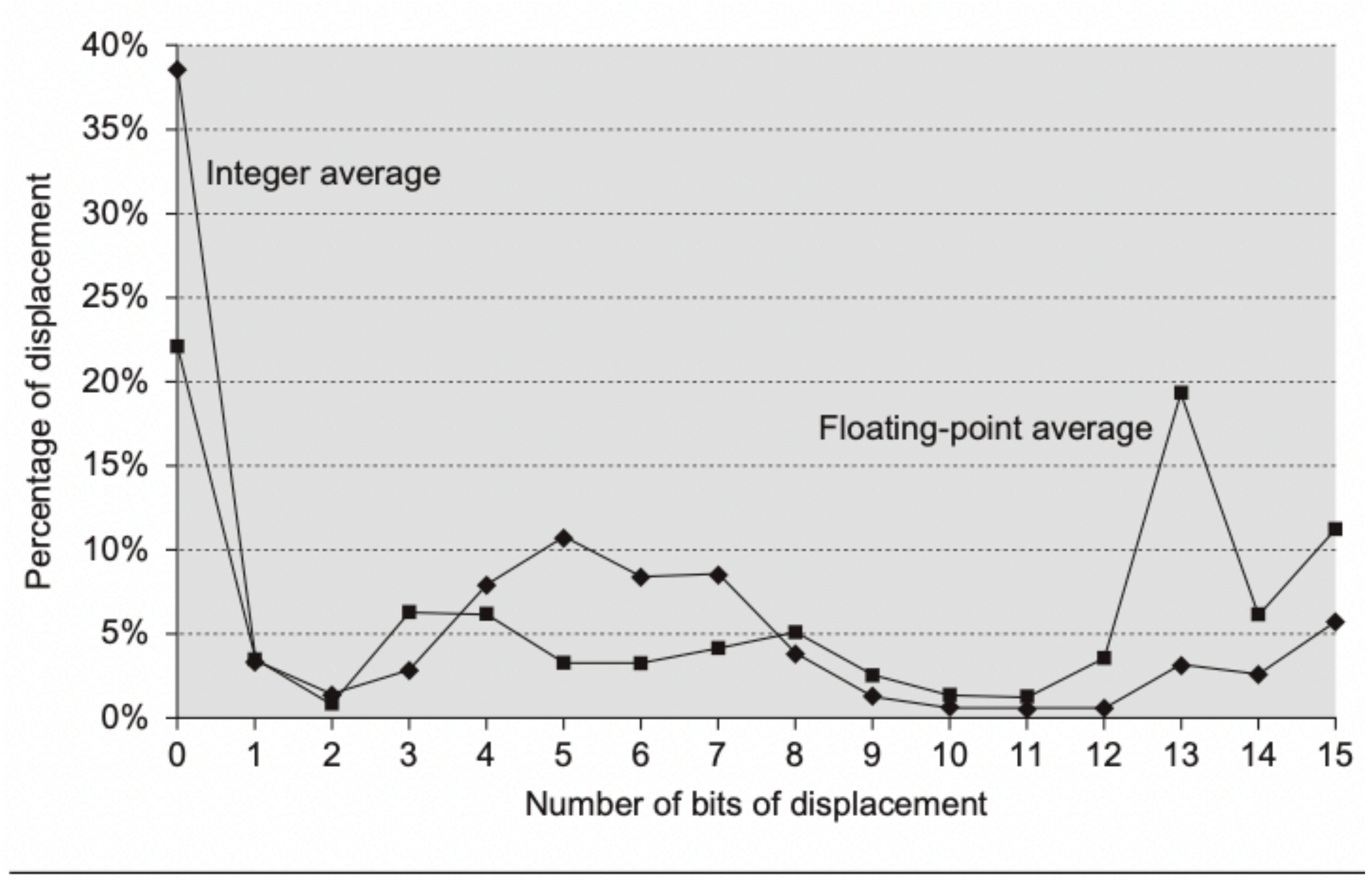
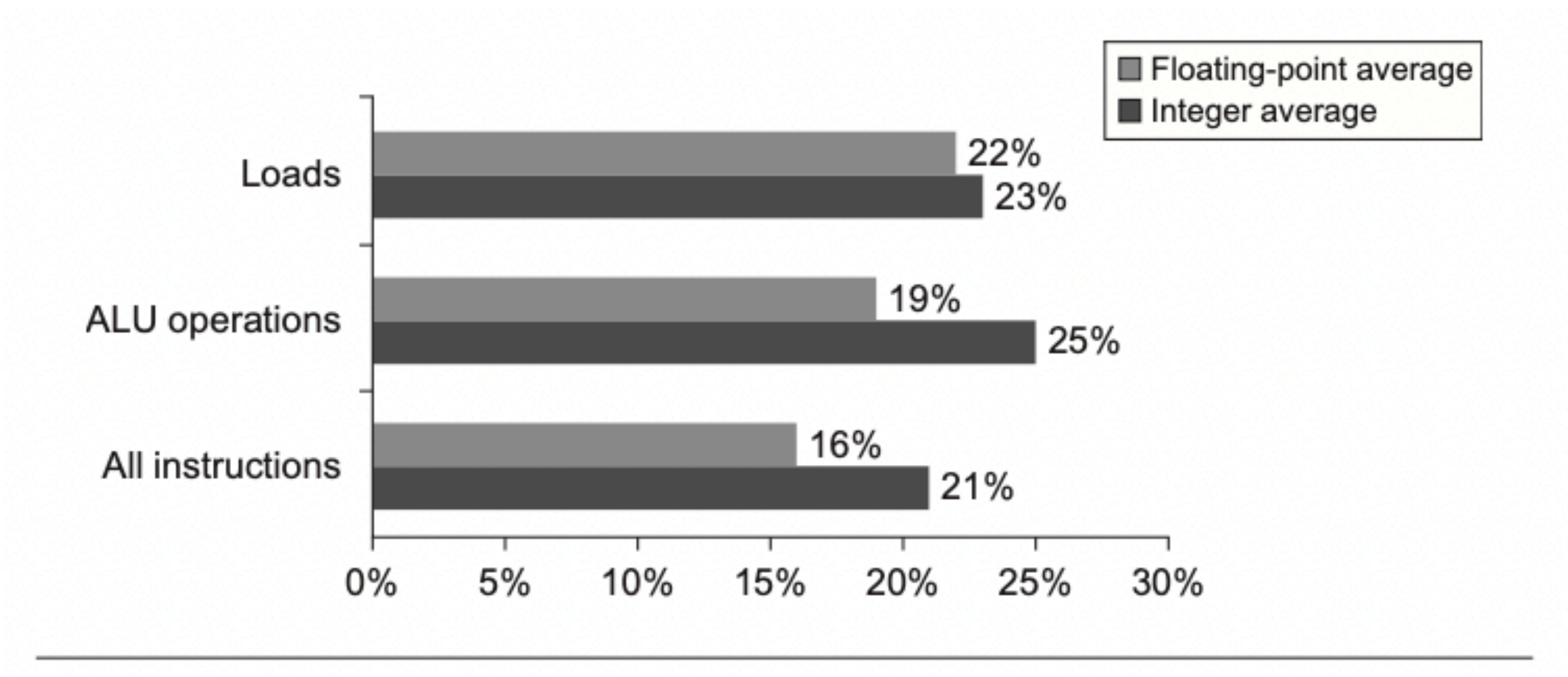


Figure A.7 Summary of use of memory addressing modes (including immediates).

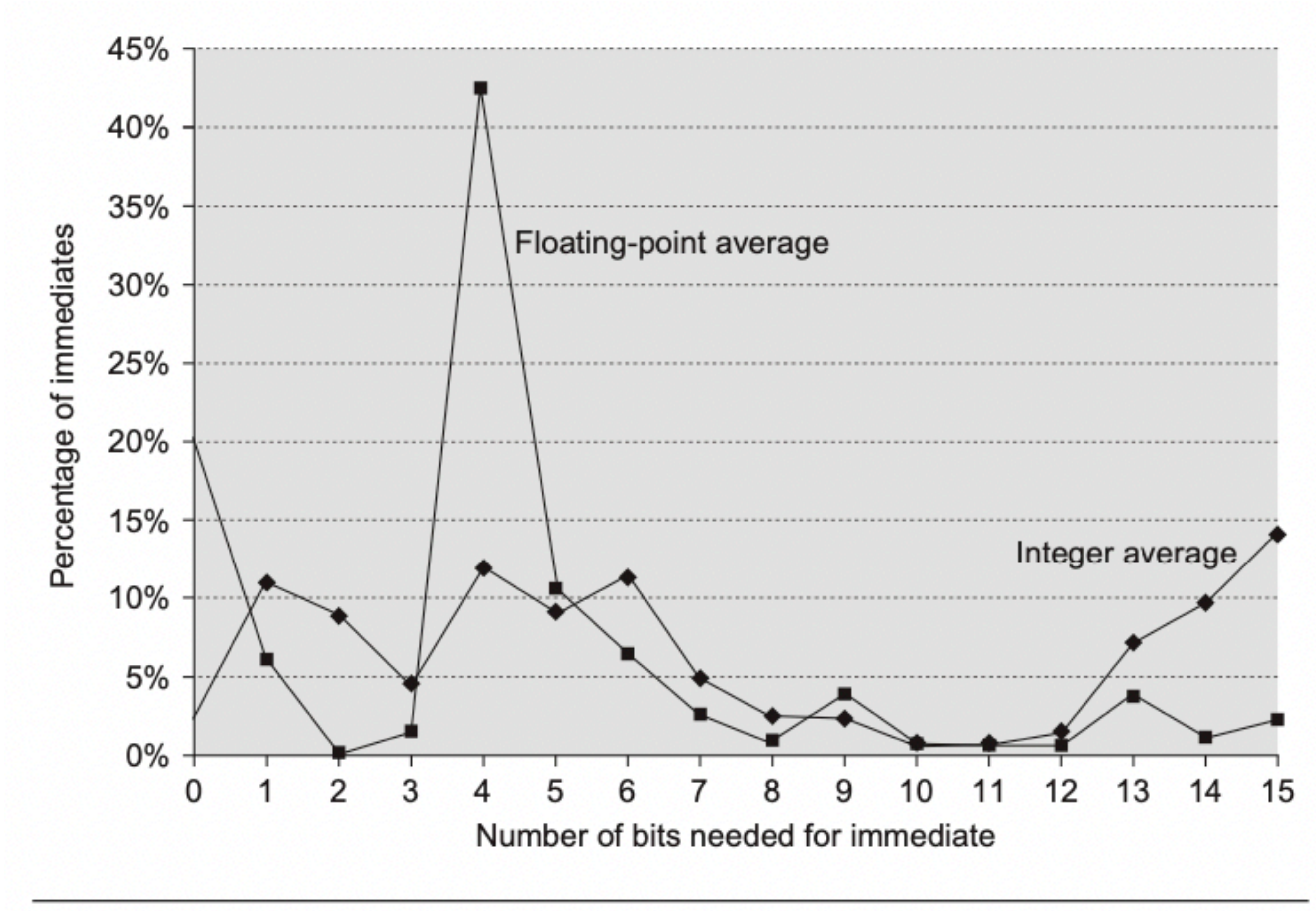
Displacement Addressing Mode



Immediate Addressing Mode Frequency



Immediate Mode Bits



Data Access Sizes

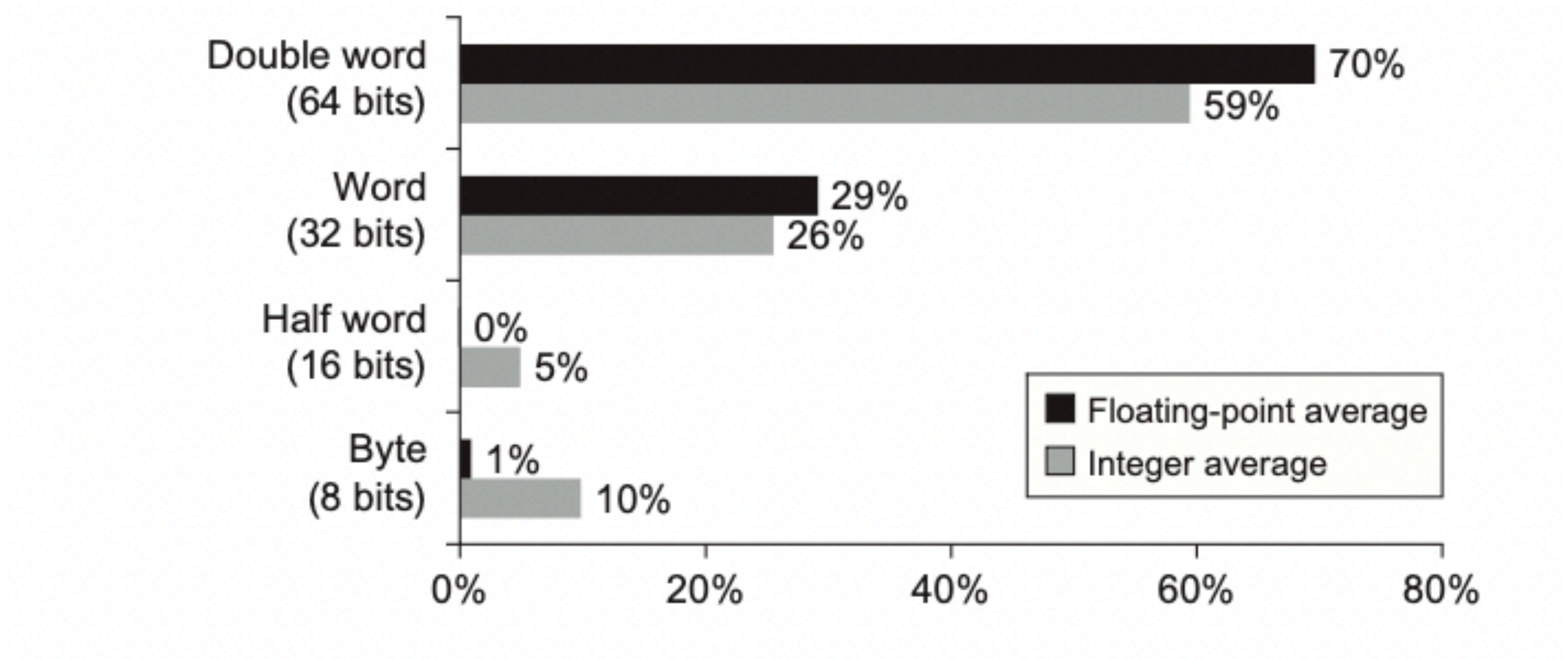
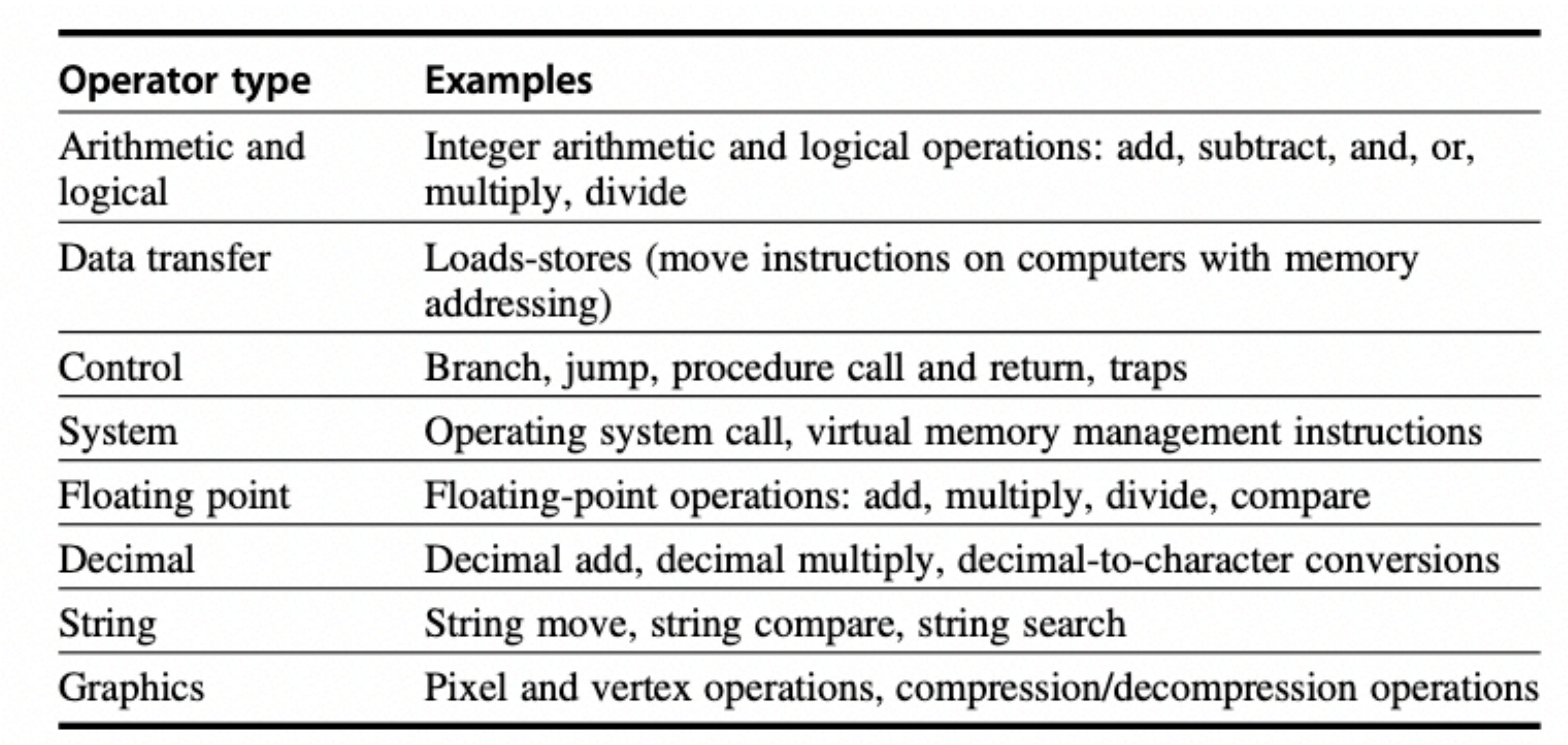


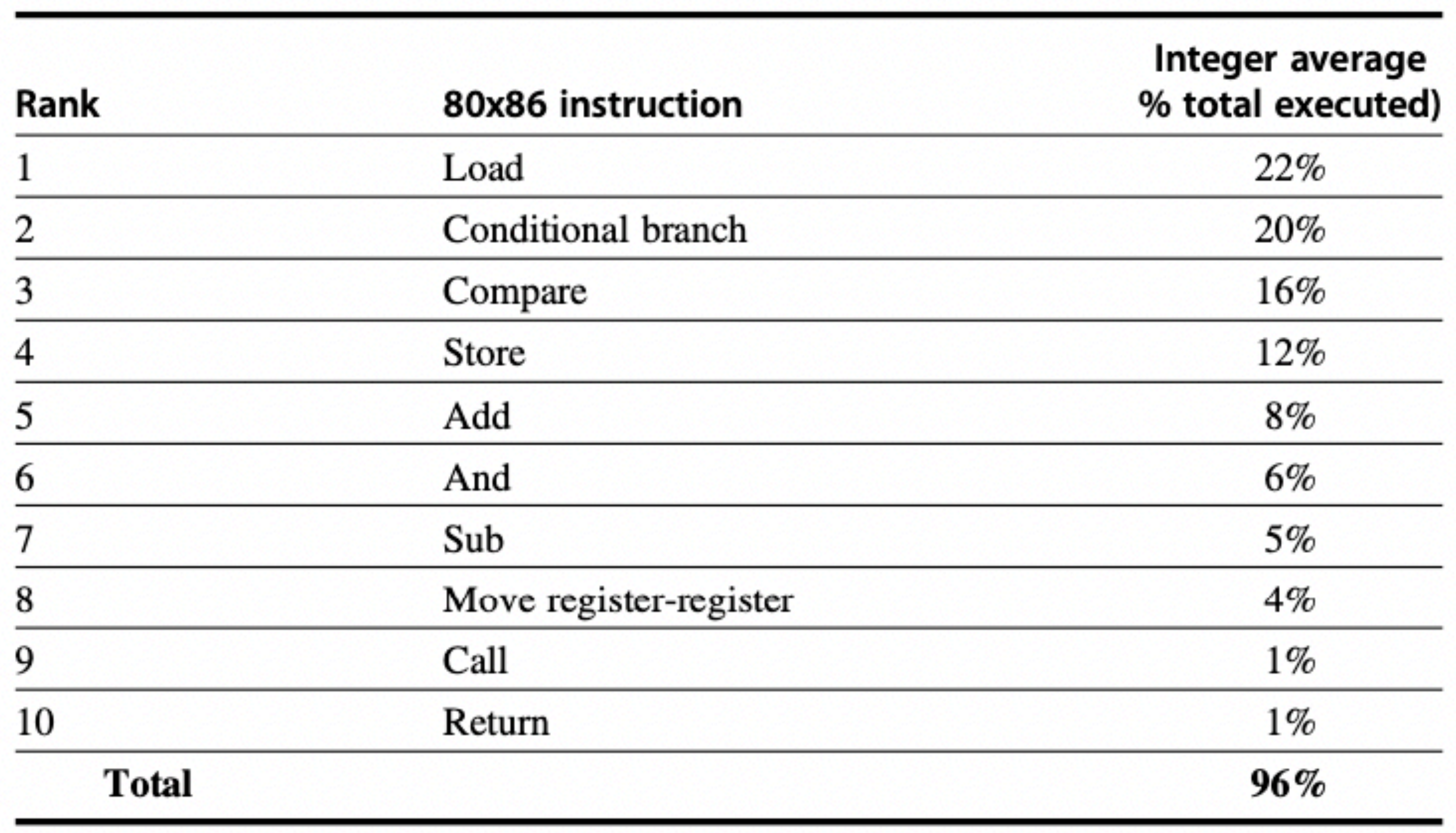
Figure A.11 Distribution of data accesses by size for the benchmark programs.

Operation Categories



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

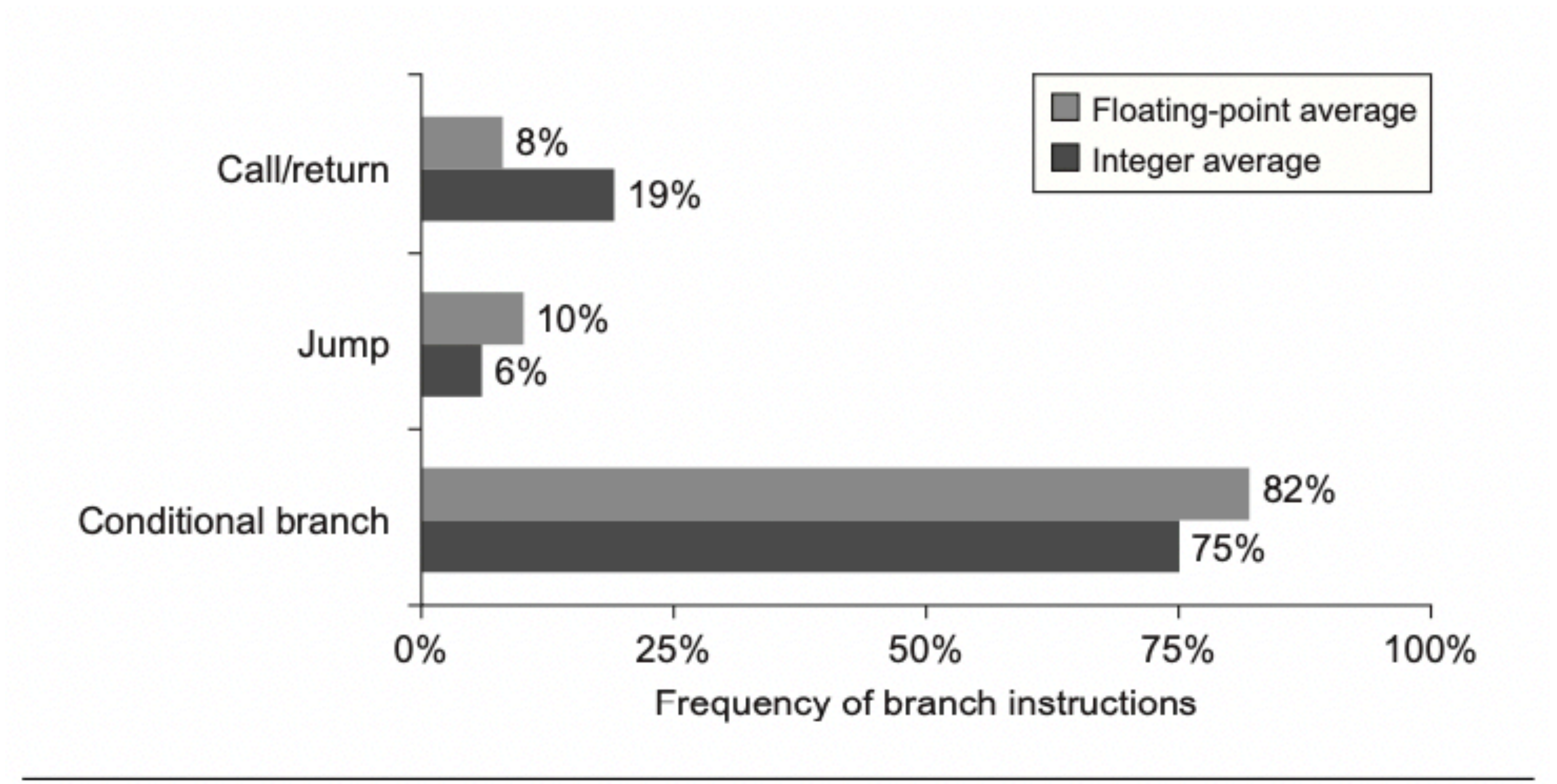
Instruction Frequency



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%**

Figure A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

Control Flow Instructions



Branch Displacement (Distance)

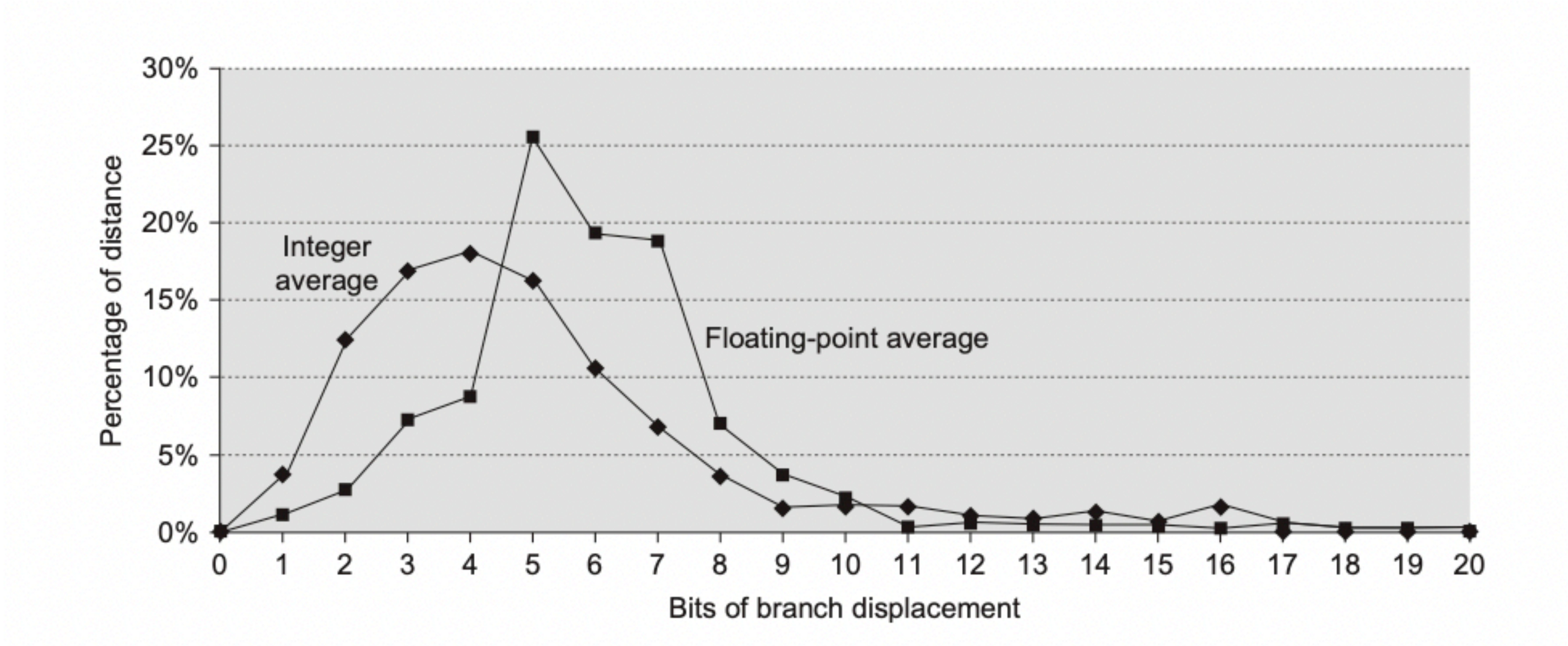
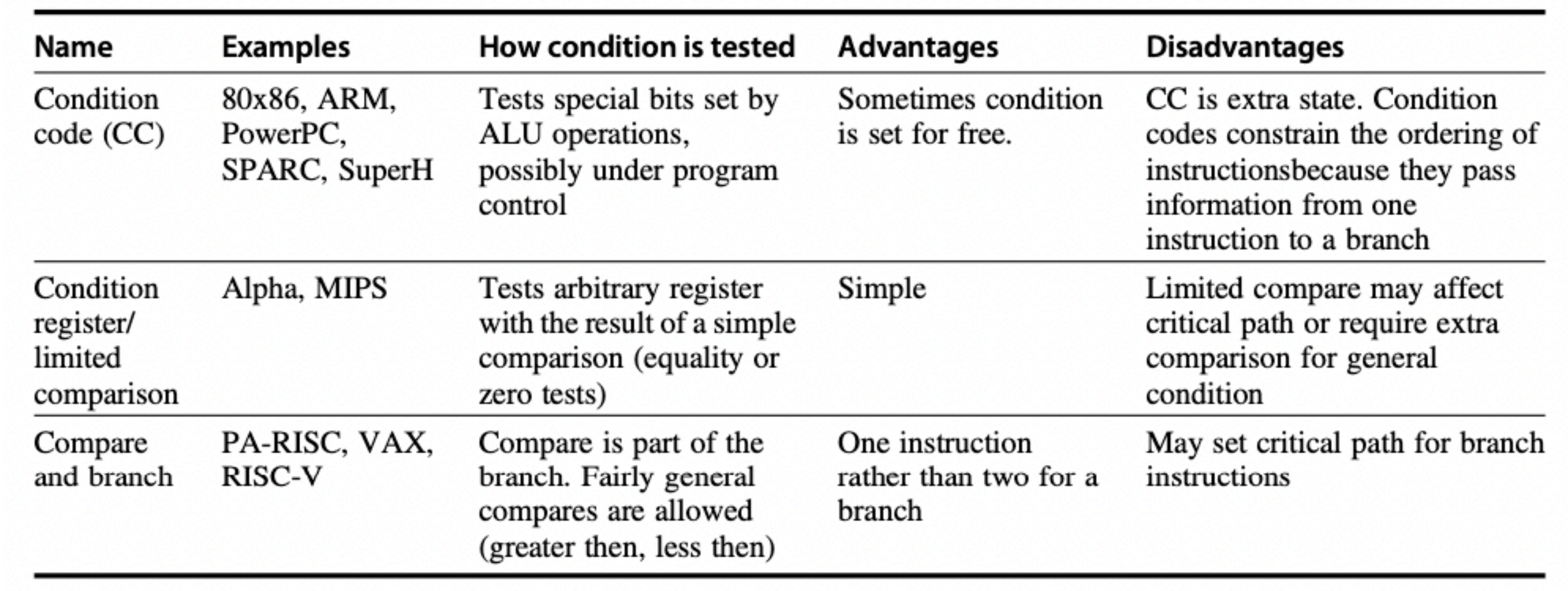


Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction.

Branch Evaluation Methods



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

Figure A.16 The major methods for evaluating branch conditions, their advantages, and their disadvantages.

Compare Frequencies

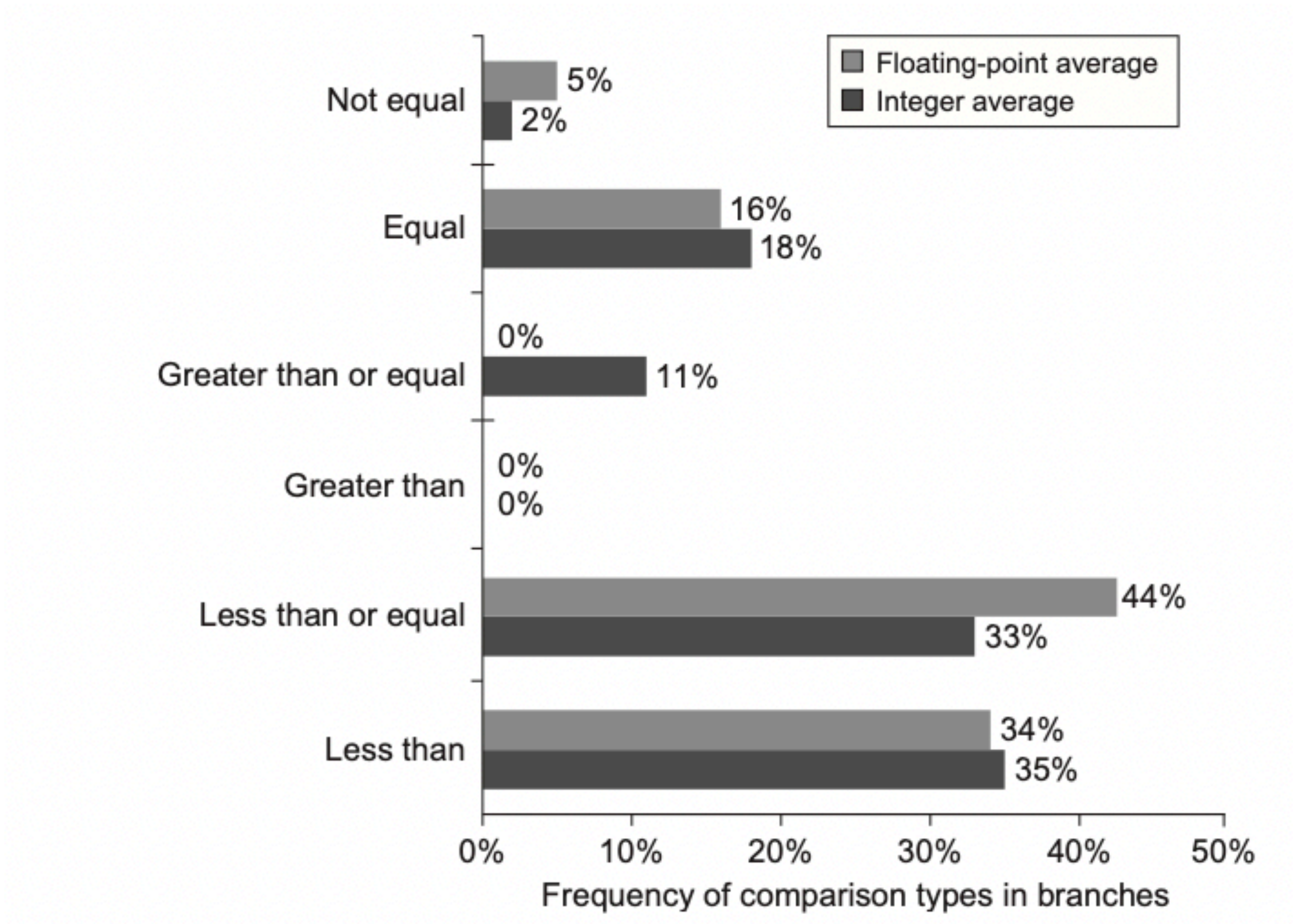
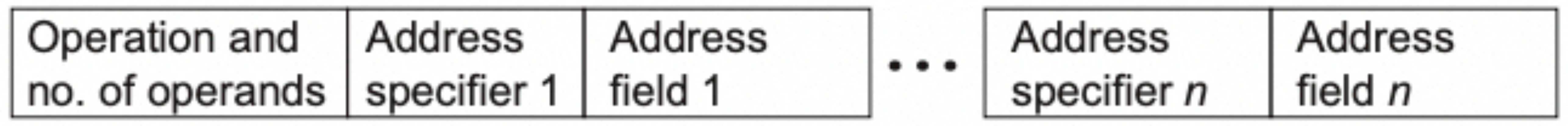
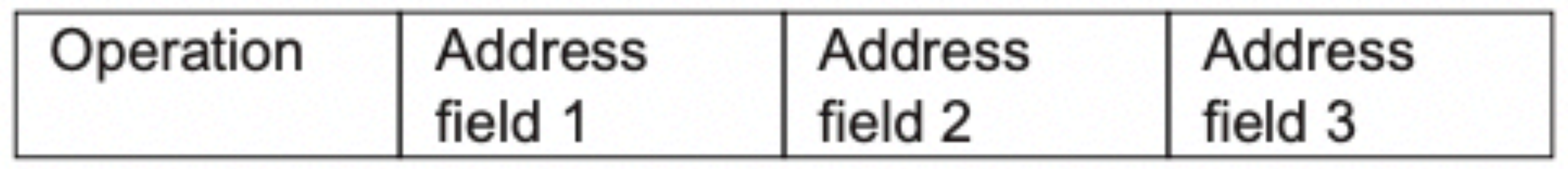


Figure A.17 Frequency of different types of compares in conditional branches.

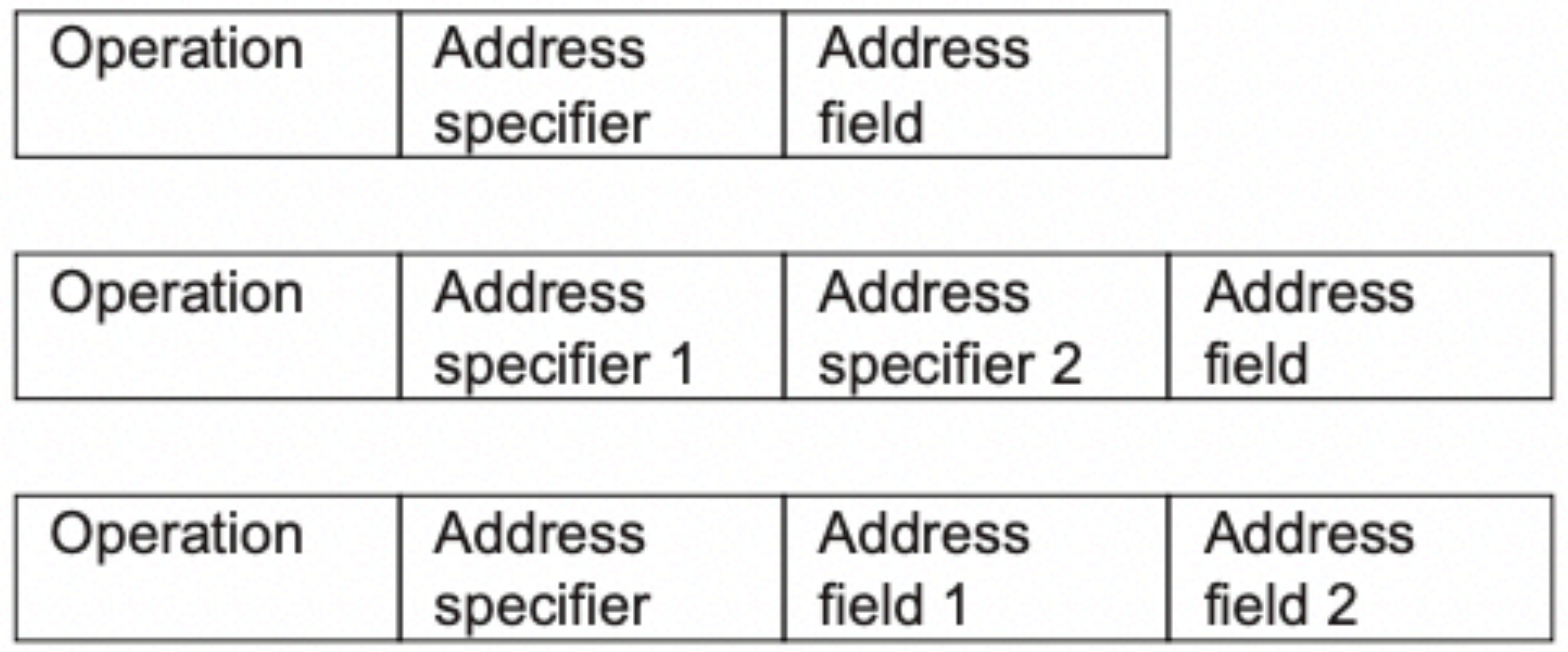
Instruction Encoding Types



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

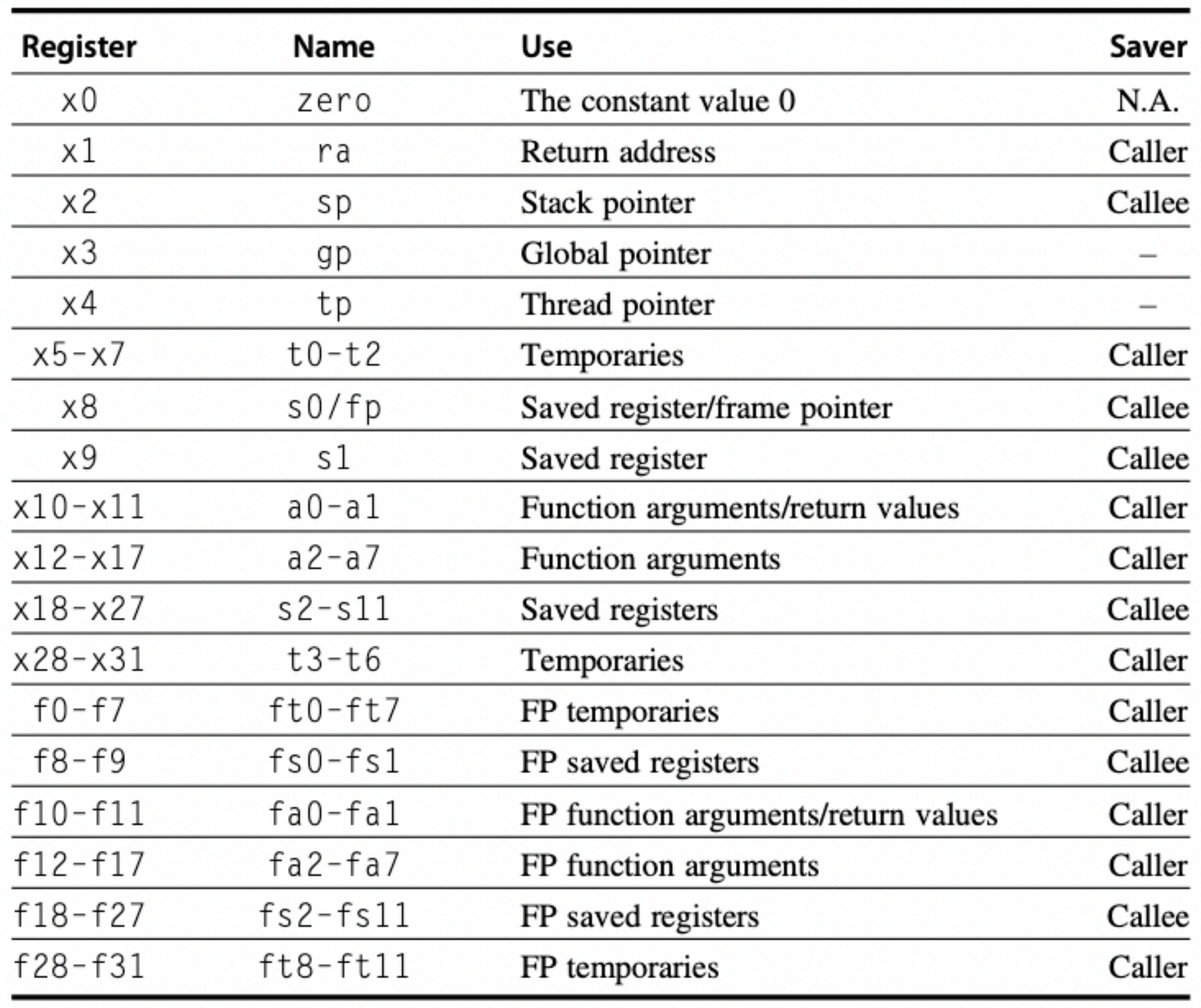


(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)



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

RISC-V Registers



Register	Name	Use	Saver
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP temporaries	Caller
f8-f9	fs0-fs1	FP saved registers	Callee
f10-f11	fa0-fa1	FP function arguments/return values	Caller
f12-f17	fa2-fa7	FP function arguments	Caller
f18-f27	fs2-fs11	FP saved registers	Callee
f28-f31	ft8-ft11	FP temporaries	Caller

Figure 1.4 RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0–x31), RISC-V has 32 floating-point registers (f0–f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled "Callee" saved.

RISC-V Data Types

- Byte - 8 bits
- Half Word - 16 bits
- Word - 32 bits
- Double Word - 64 bits
- Single Precision FP - 32 bits
- Double Precision FP - 64 bits

RISC-V Addressing Modes

- Register
- Displacement
- Immediate

RISC-V Instruction Formats

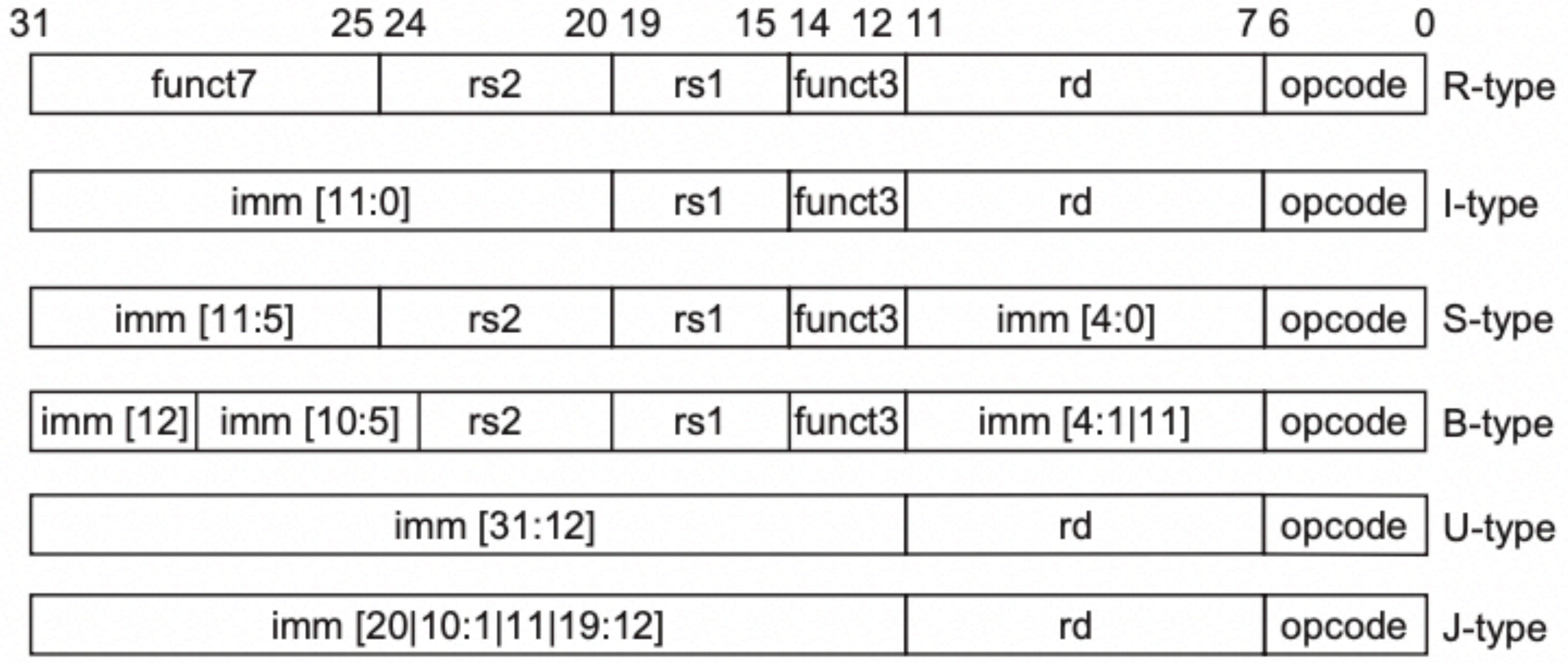
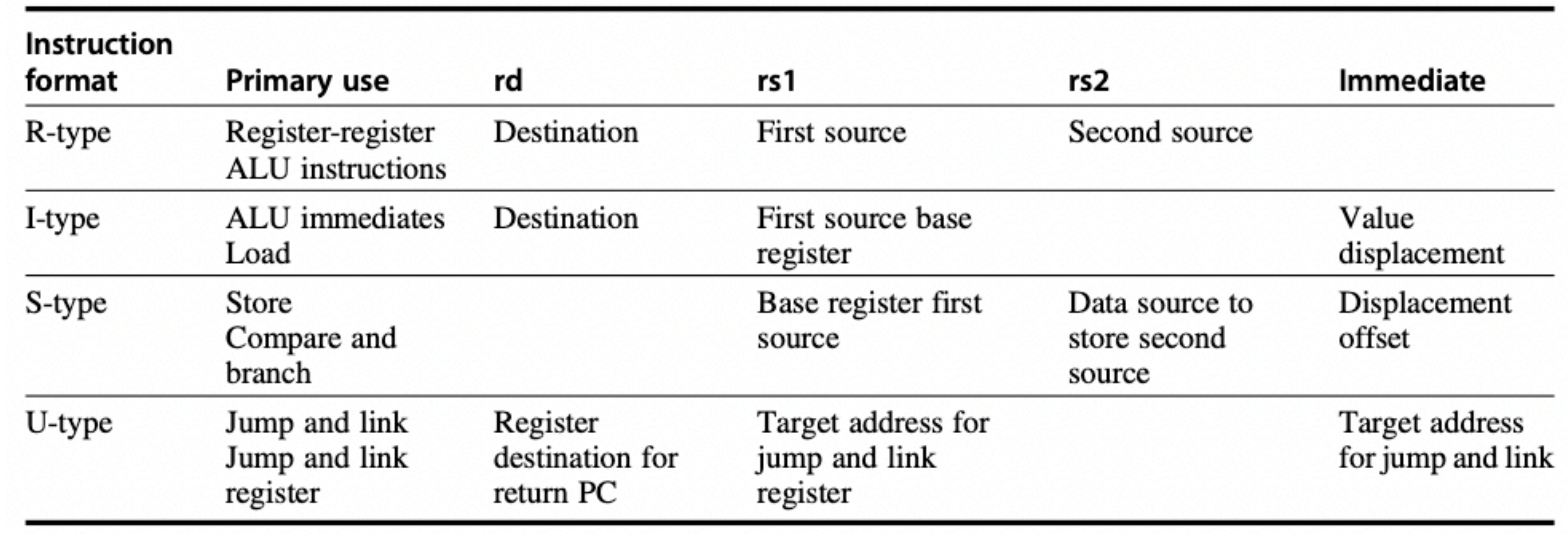


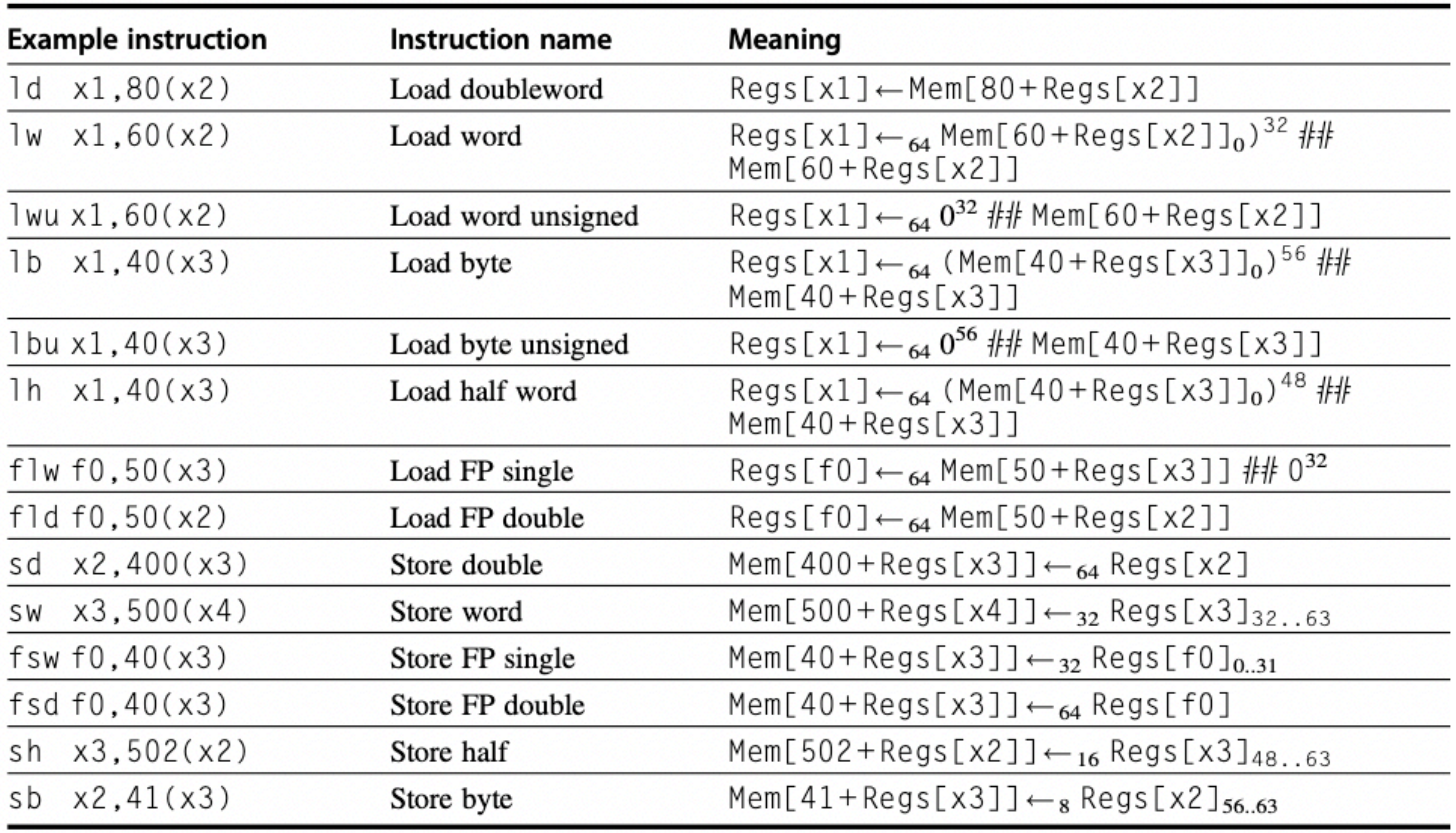
Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

RISC-V Instruction Formats - Usage



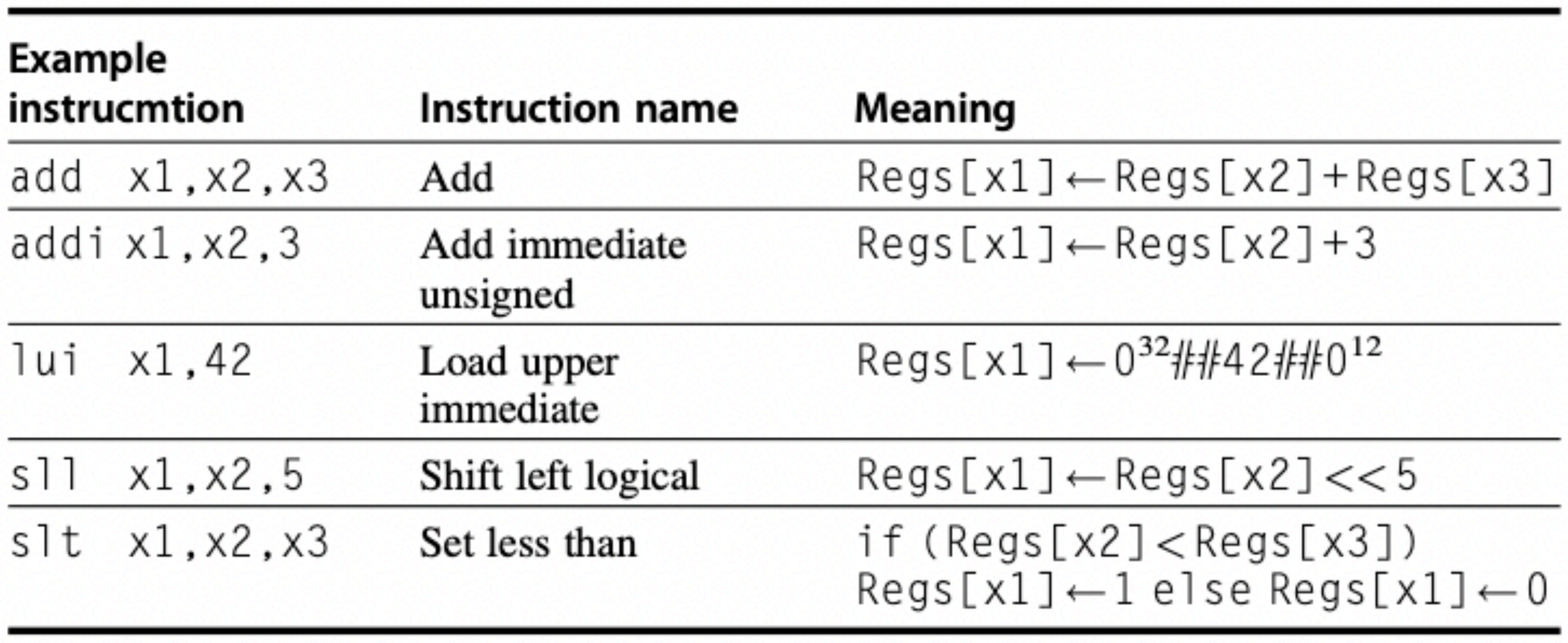
Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

RISC-V Load-Store Instruction Examples



Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1]$\leftarrow$Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1]$\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1]$\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]]$\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]]$\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]]$\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]]$\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]]$\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]]$\leftarrow_{8}$ Regs[x2]$_{56..63}$

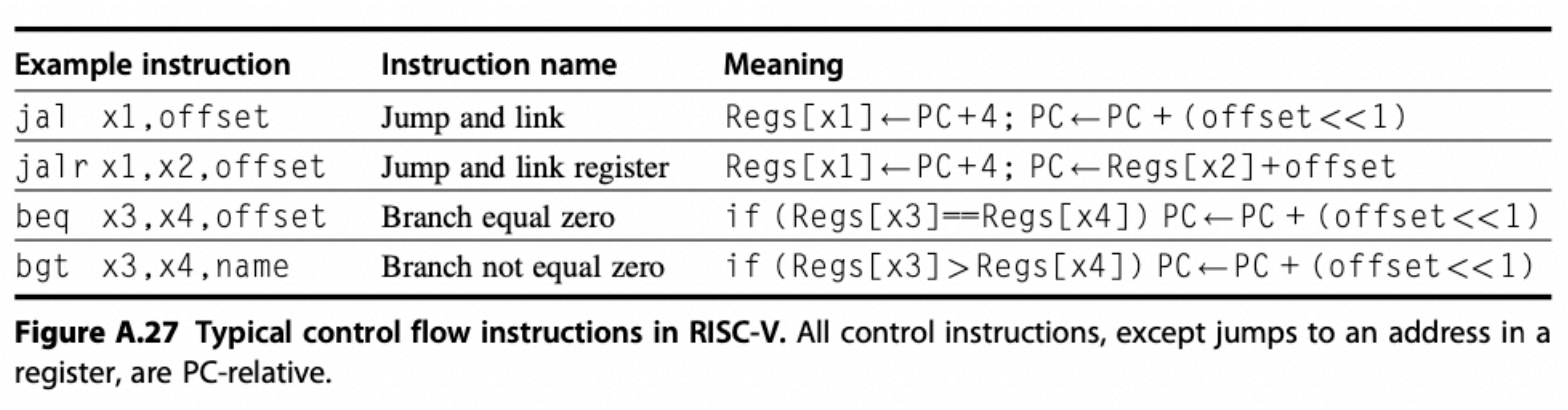
RISC-V ALU Examples



Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	`Regs[x1]`$\leftarrow 0^{32}\#\#42\#\#0^{12}$
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`
`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0`

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

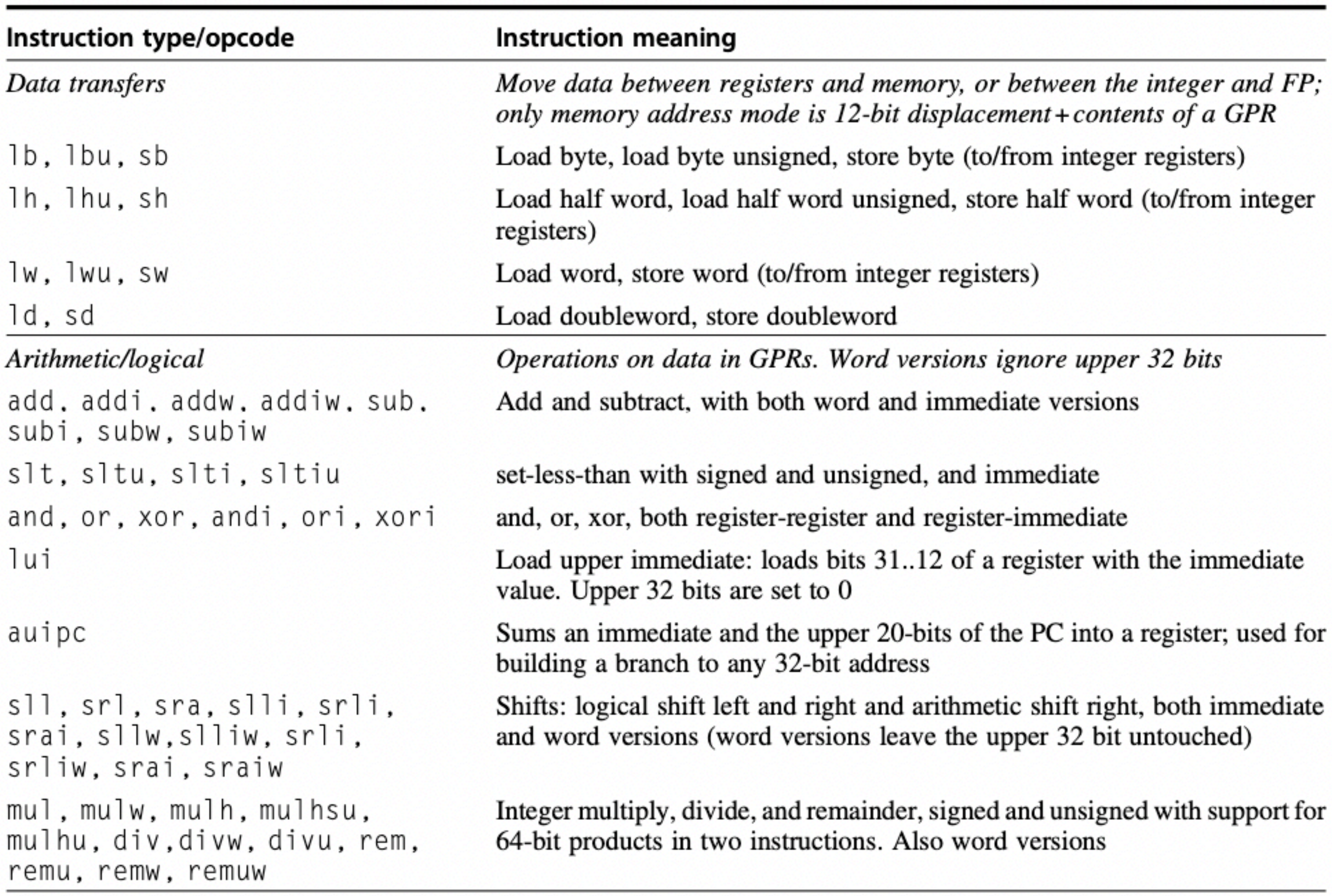
RISC-V Control Flow Examples



Example instruction	Instruction name	Meaning
`jal  x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq  x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt  x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

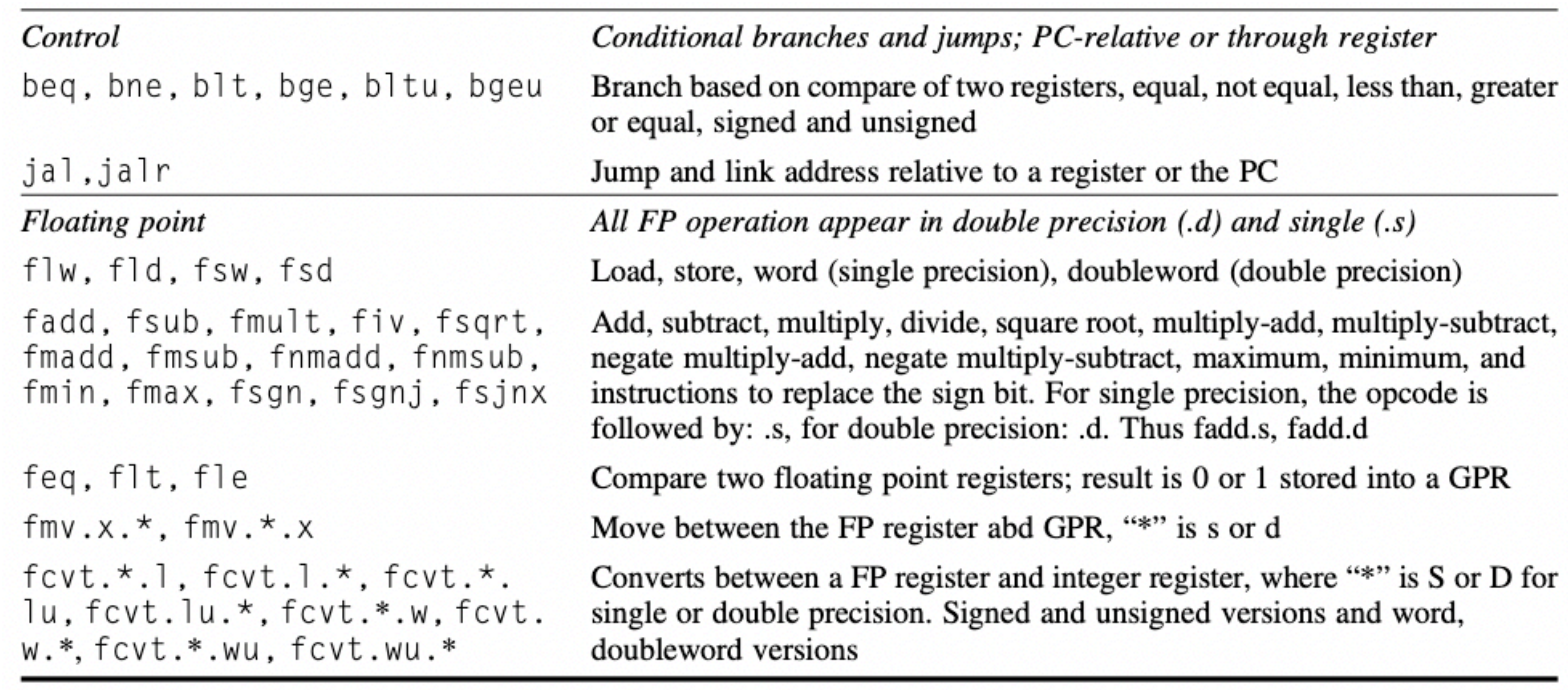
Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V Instruction Subset (1 of 2)



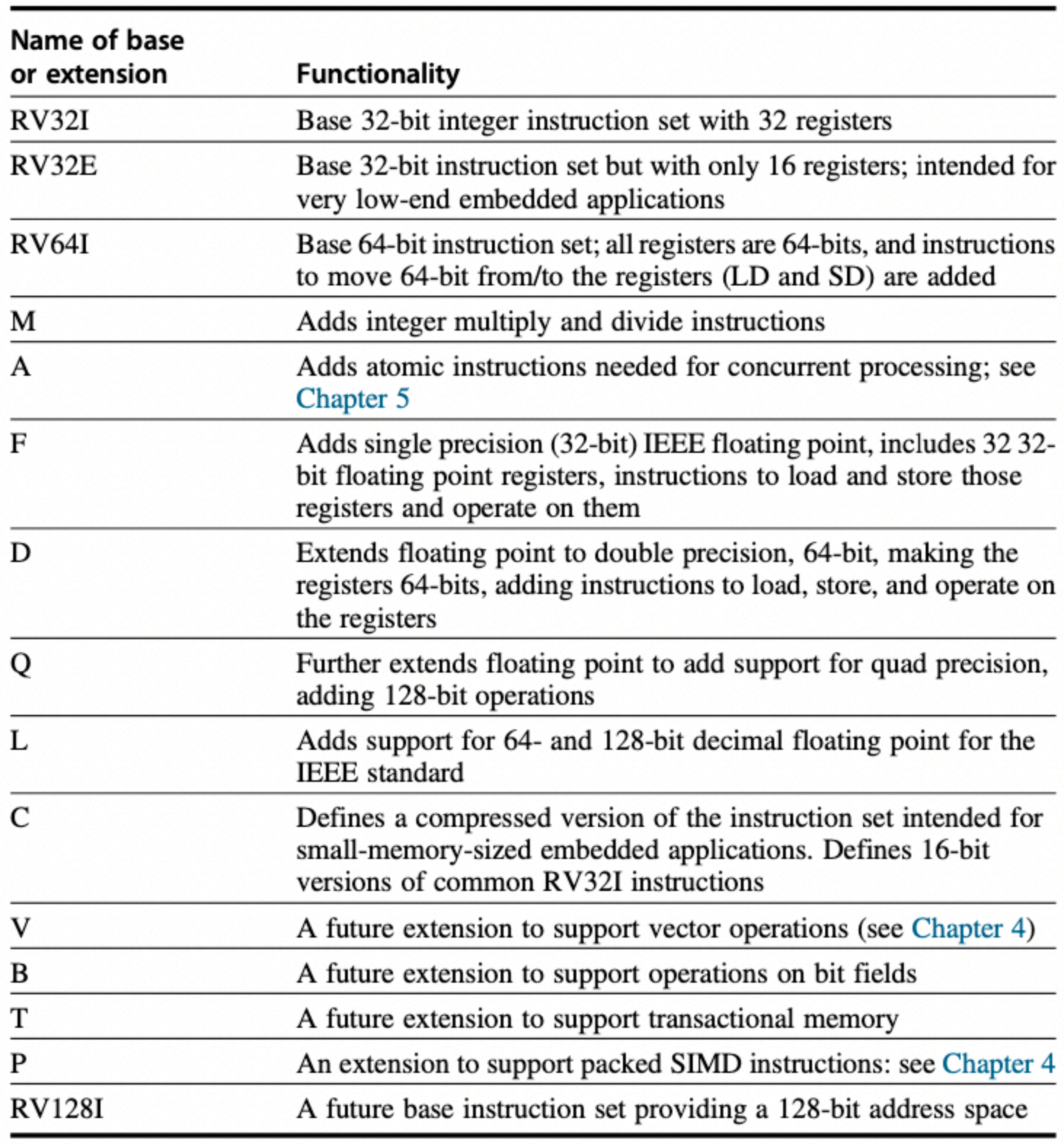
Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions

RISC-V Instruction Subset (2 of 2)



Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

RISC-V Variants and Extensions



Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space