# MO401

IC/Unicamp

Prof Mario Côrtes

# Appendix A: ISA Principles

# Tópicos

- Tipos de ISA (Instruction Set Architectures)
- Endereçamento de memória
- Tipos de operandos
- Operações no ISA
- Instruções de controle de fluxo de execução
- Codificação
- O ISA do MIPS 64
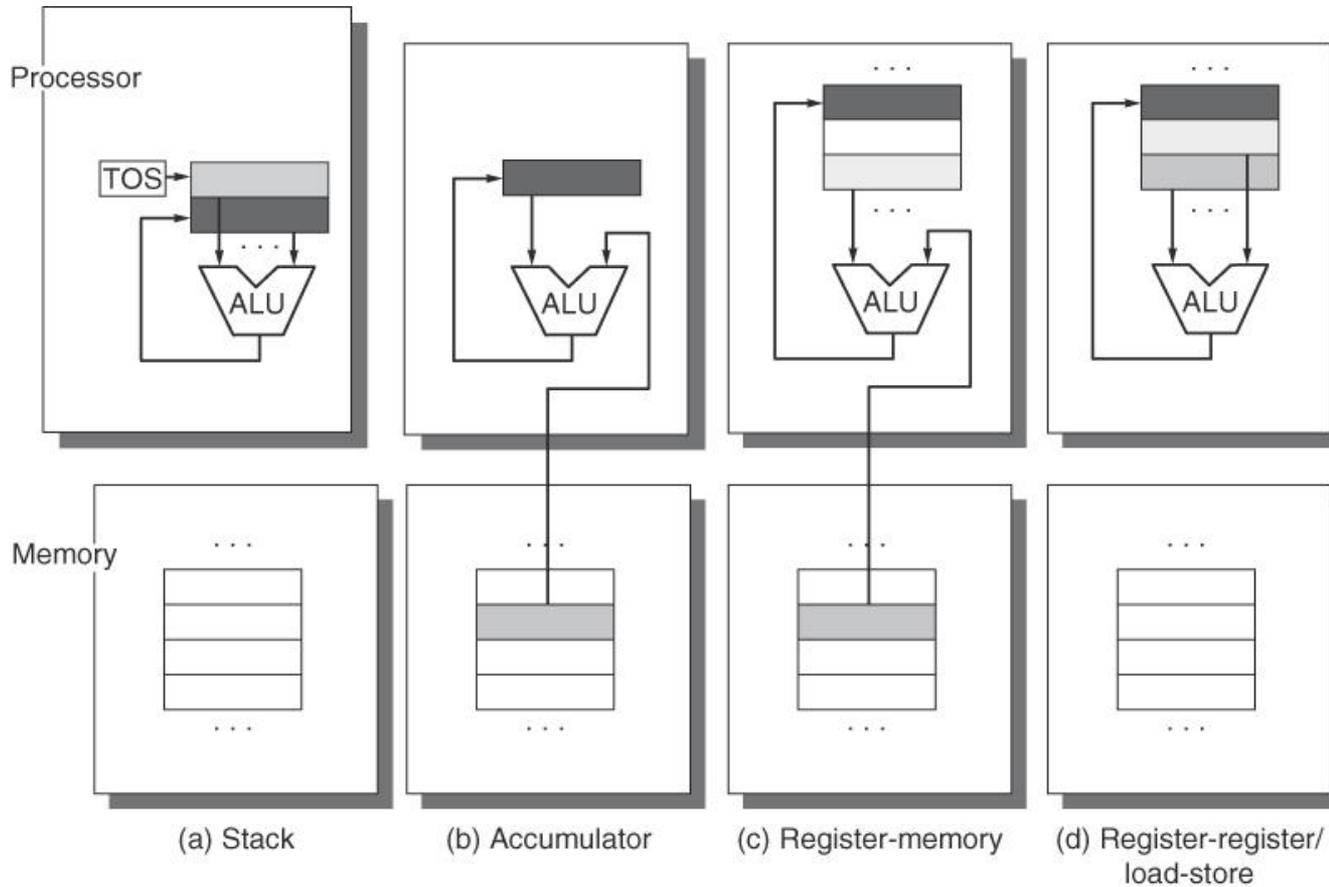
**Figure A.1 Operand locations for four instruction set architecture classes.** The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS) points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).

| 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]] <br> 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] - $d$ <br> 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. |

**Figure A.6** Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable $d$ designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A-36.
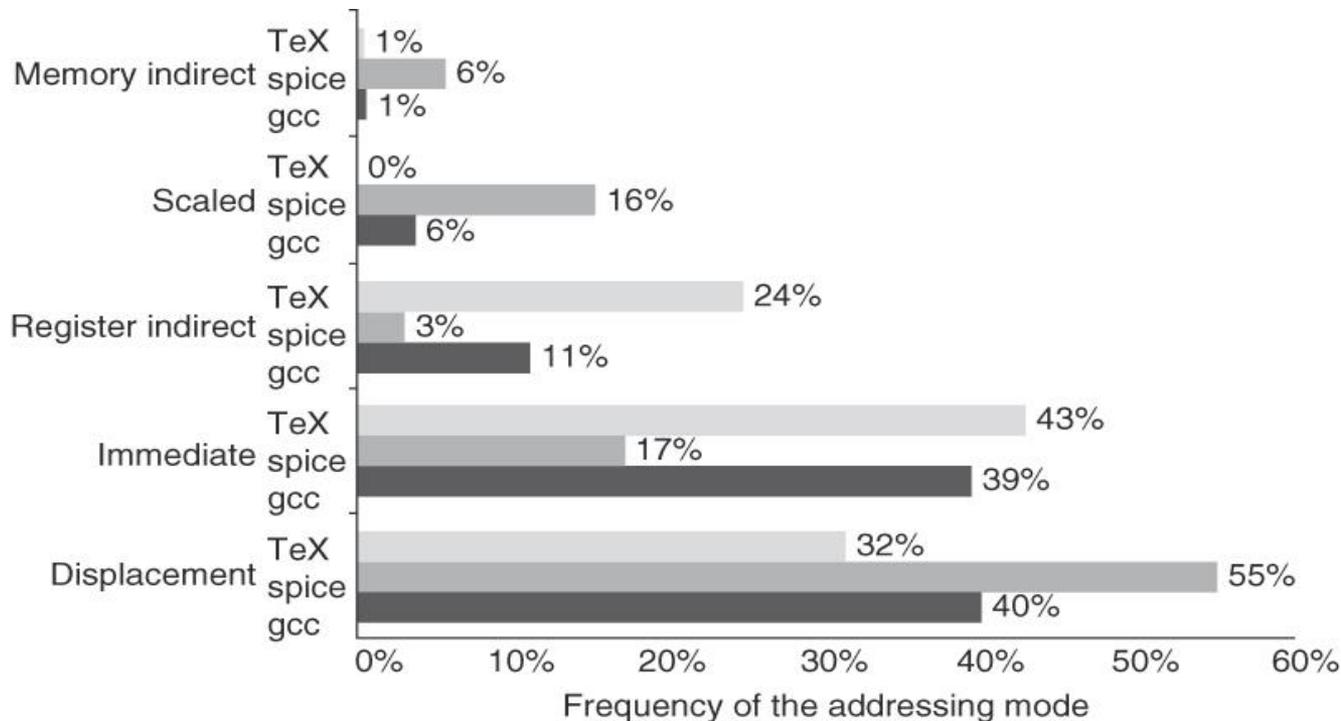
**Figure A.7 Summary of use of memory addressing modes (including immediates).** These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bits). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown.
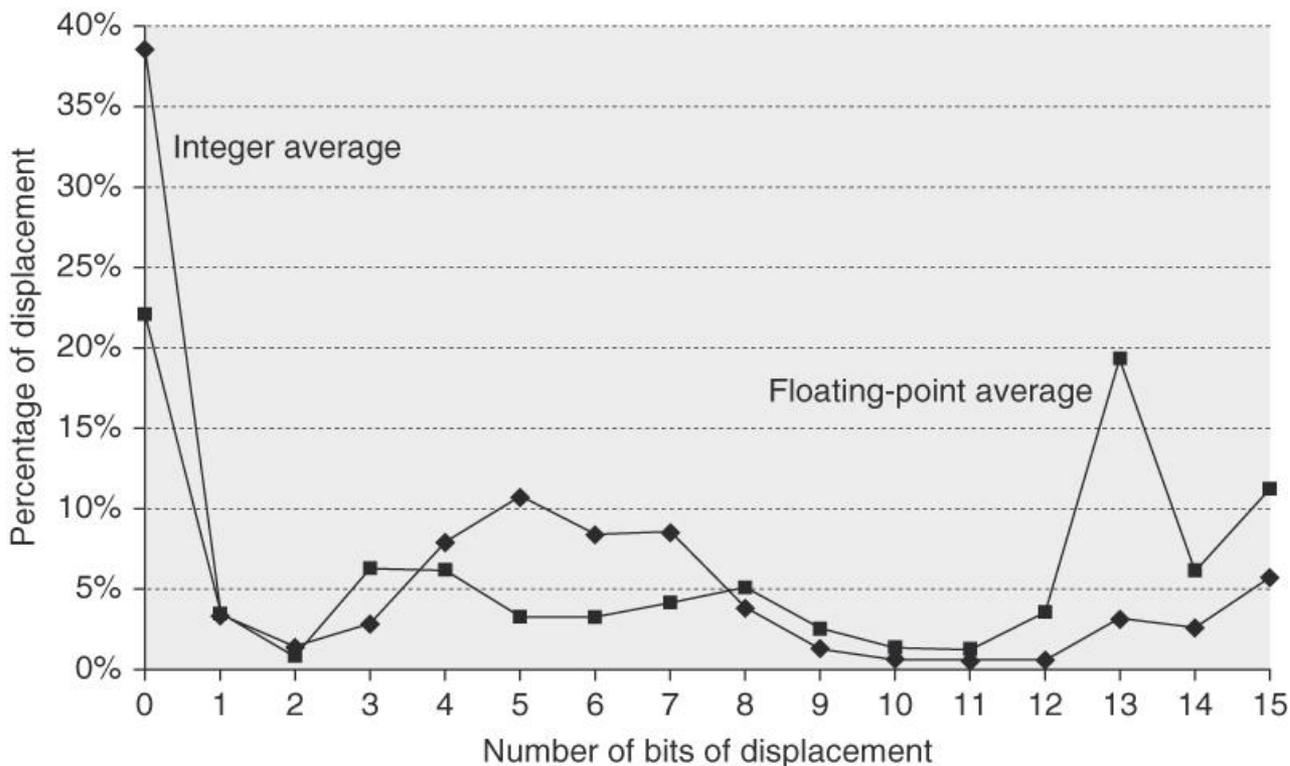
**Figure A.8 Displacement values are widely distributed.** There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see Section A.8) as well as the overall addressing scheme the compiler uses. The *x*-axis is log2 of the displacement, that is, the size of a field needed to represent the magnitude of the displacement. Zero on the *x*-axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Since these data were collected on a computer with 16-bit displacements, they cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see Section A.8) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).
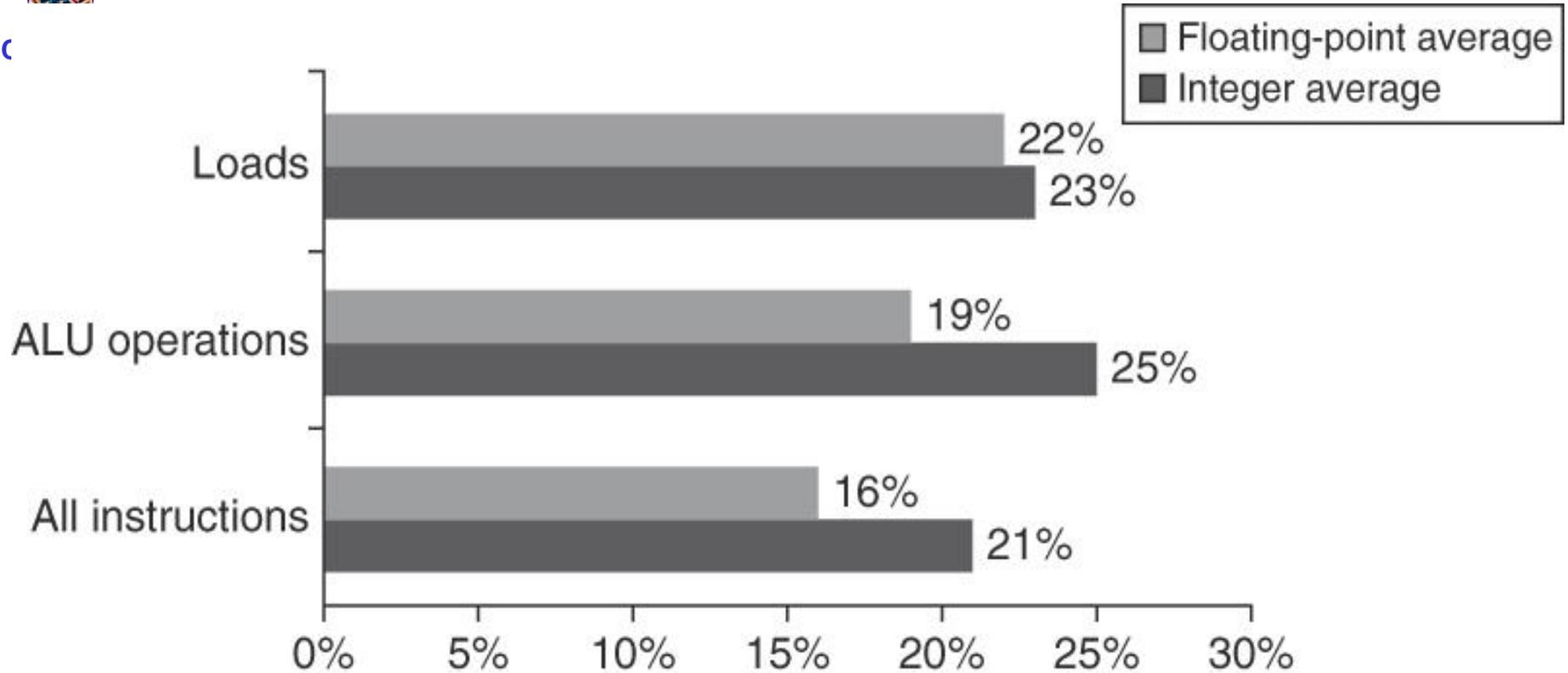
**Figure A.9 About one-quarter of data transfers and ALU operations have an immediate operand.** The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) The programs and computer used to collect these statistics are the same as in Figure A.8.
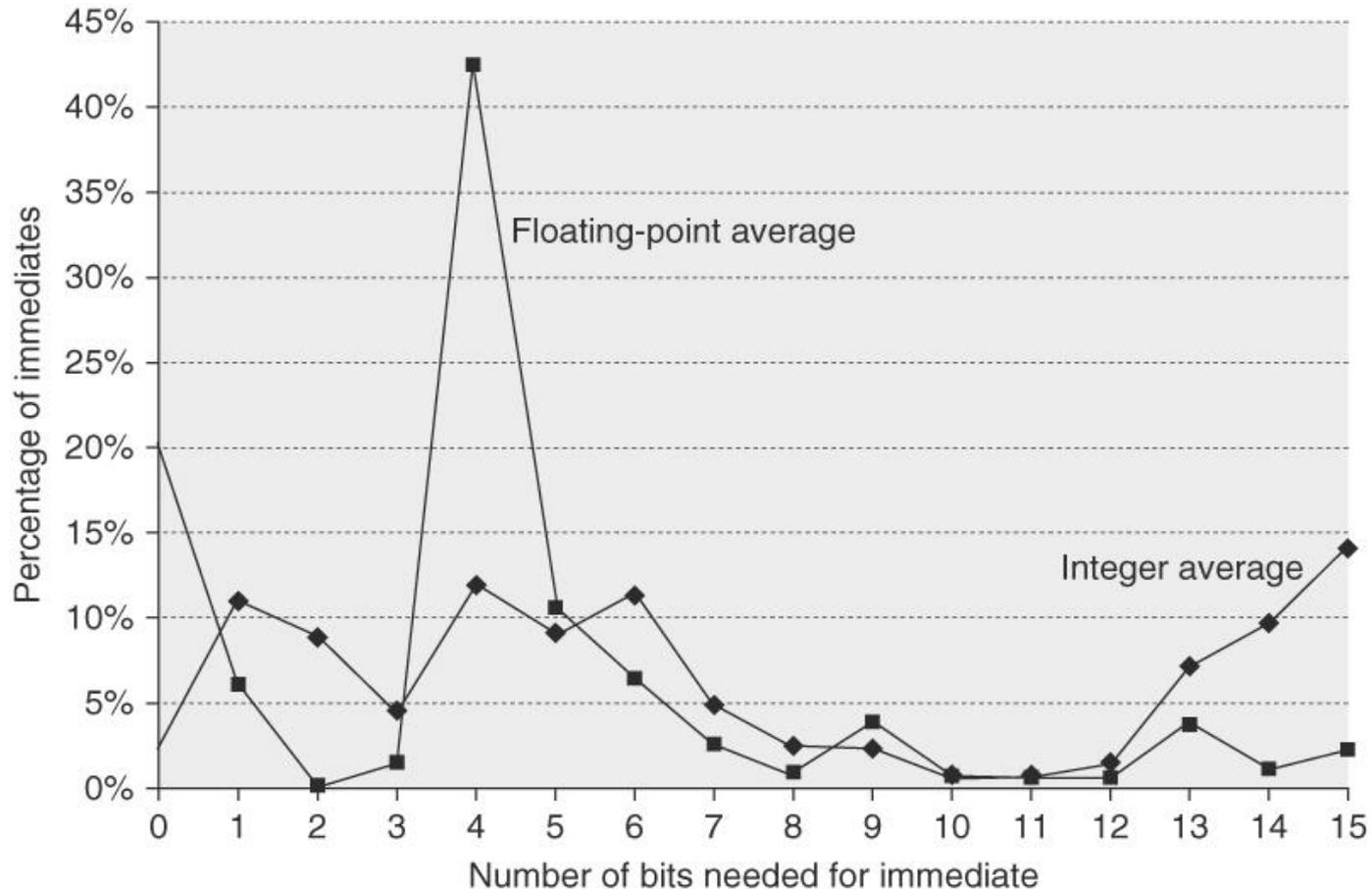
**Figure A.10 The distribution of immediate values.** The *x*-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for CFP2000. These measurements were taken on an Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure A.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits. Thus, 16 bits would capture about 80% and 8 bits about 50%.
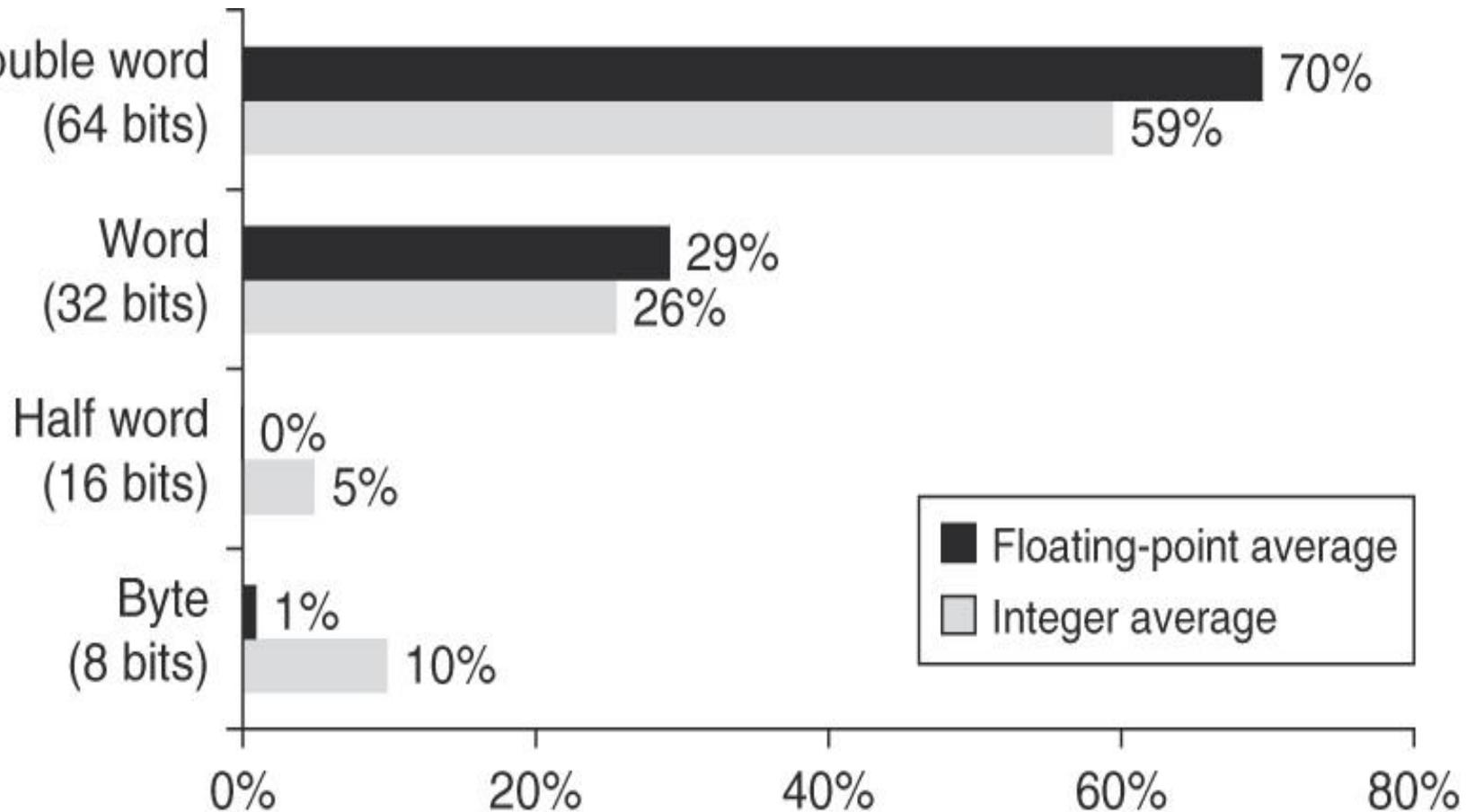
**Figure A.11 Distribution of data accesses by size for the benchmark programs.** The double-word data type is used for double-precision floating point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.
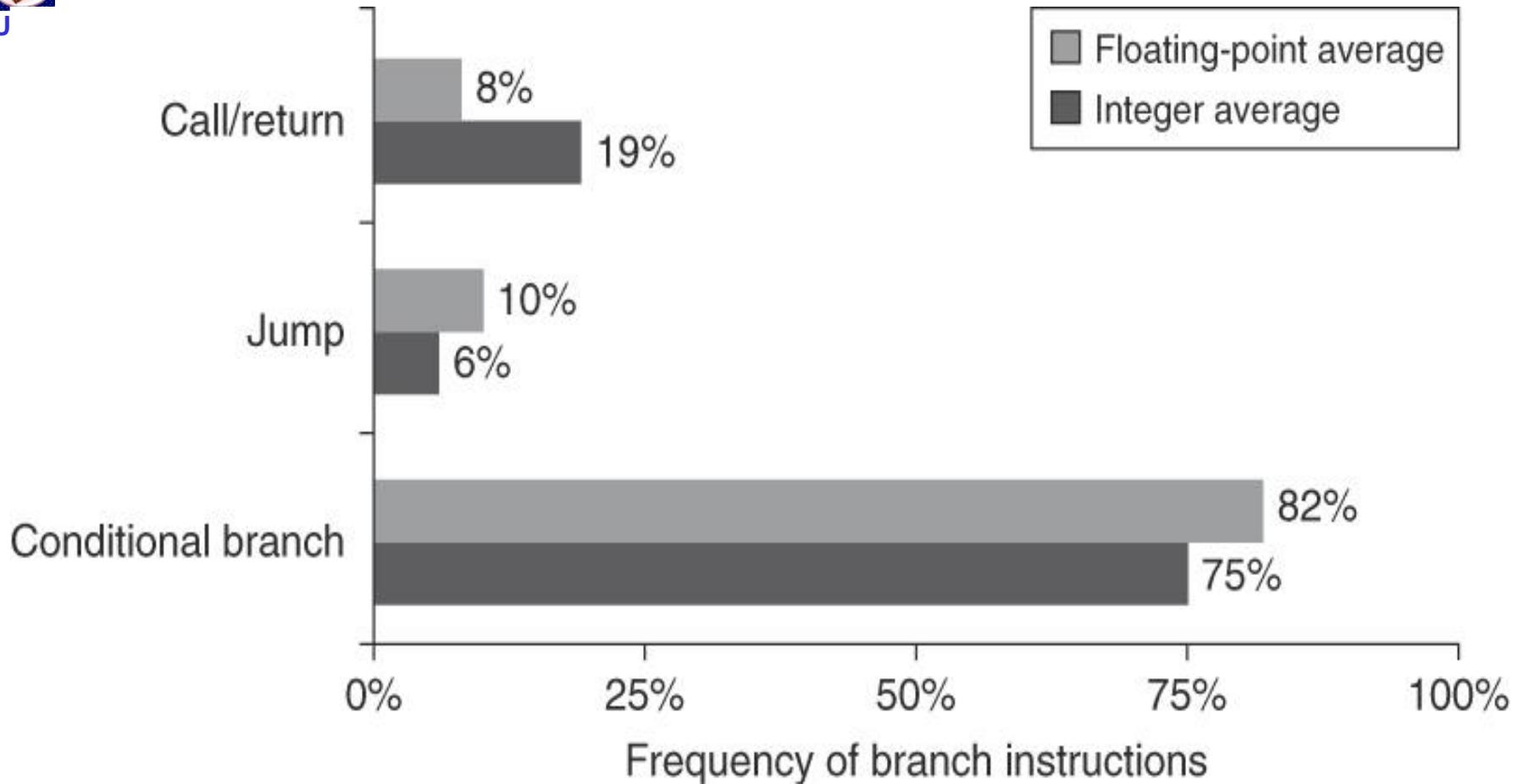
**Figure A.14 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches.** Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure A.8.
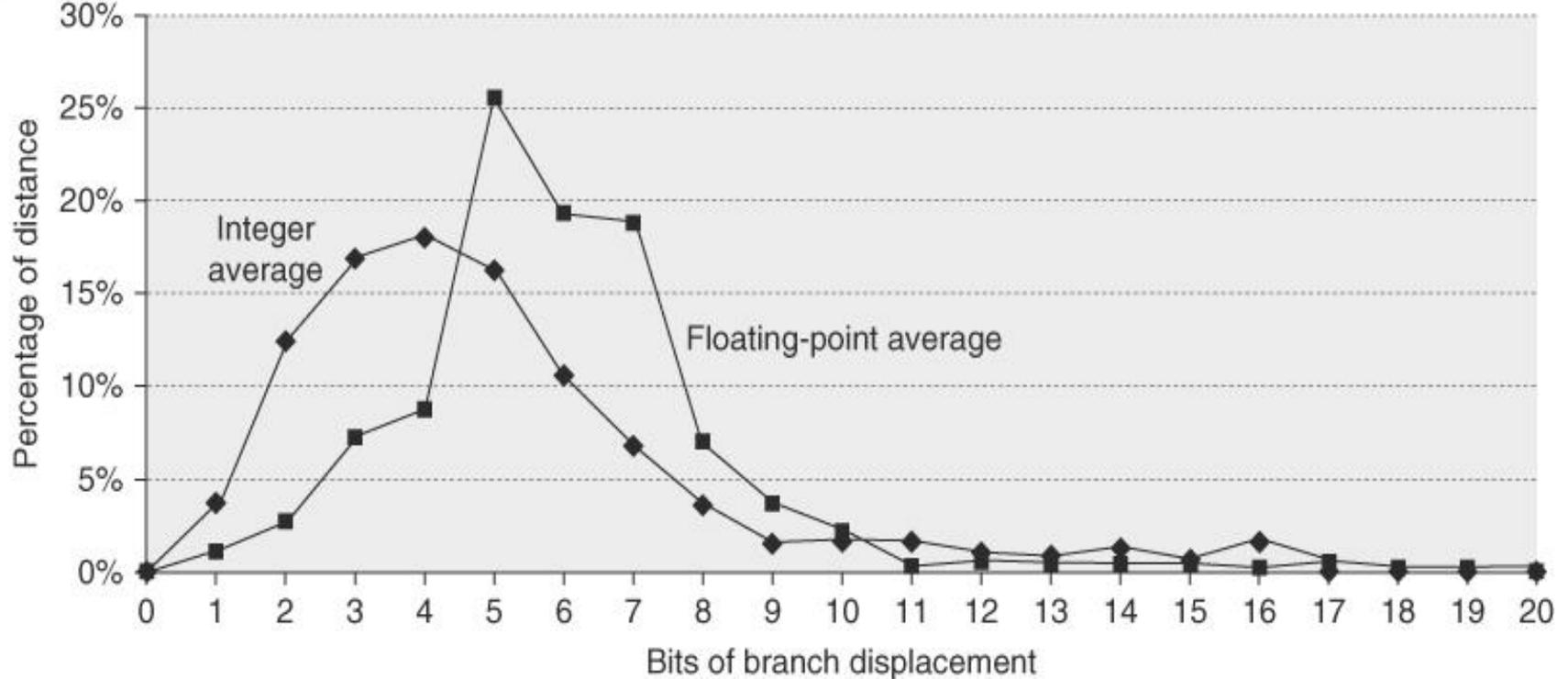
**Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction.** The most frequent branches in the integer programs are to targets that can be encoded in 4 to 8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.
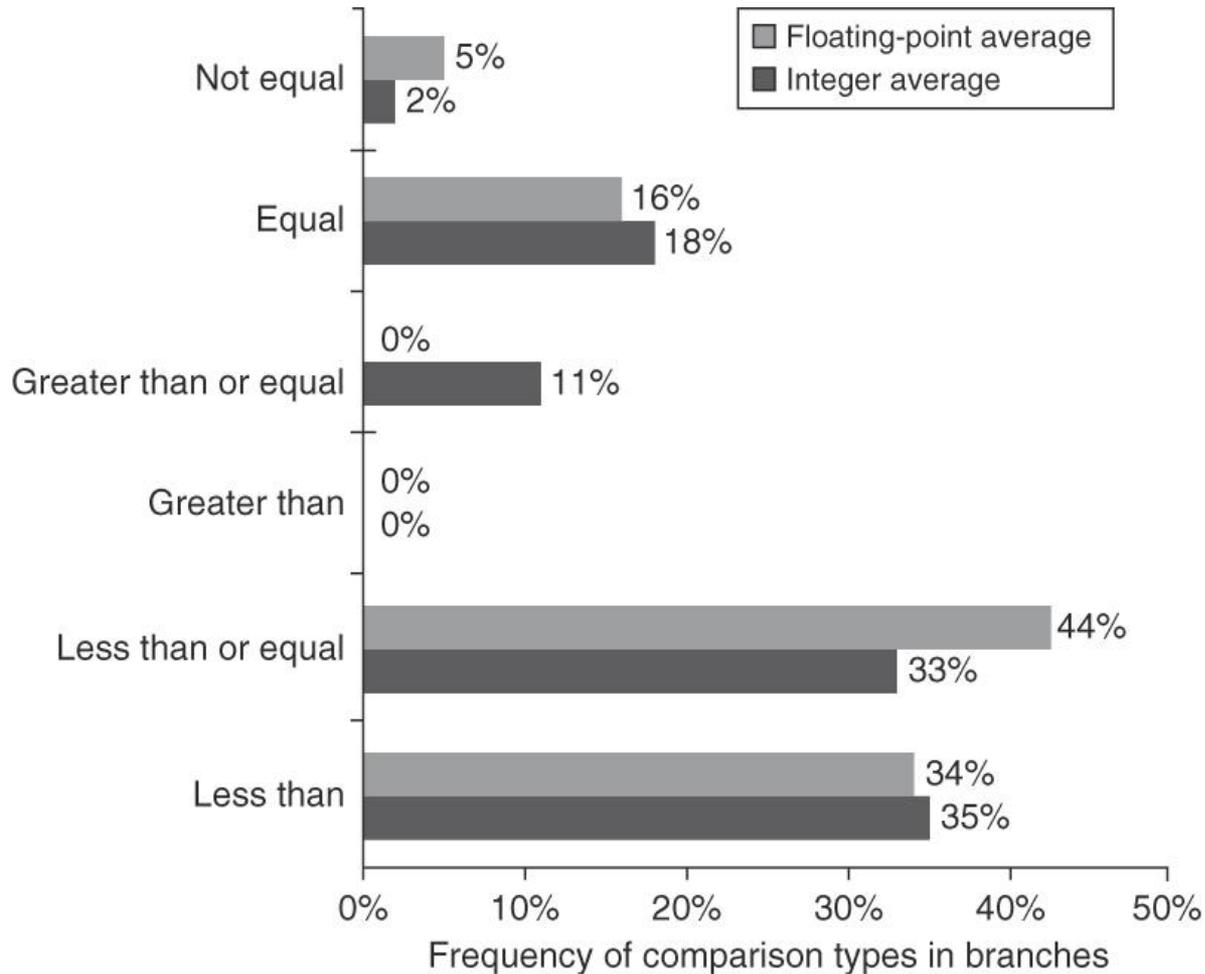
**Figure A.17 Frequency of different types of compares in conditional branches.** Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure A.8.

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

| Operation and no. of operands | Address specifier 1 | Address field 1 | $\cdots$ | Address specifier $n$ | Address field $n$ |
|---|---|---|---|---|---|

| Operation | Address field 1 | Address field 2 | Address field 3 |
|---|---|---|---|
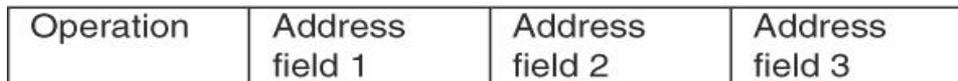
(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

| Operation | Address specifier | Address field |
|---|---|---|

| Operation | Address specifier 1 | Address specifier 2 | Address field |
|---|---|---|---|

| Operation | Address specifier | Address field 1 | Address field 2 |
|---|---|---|---|
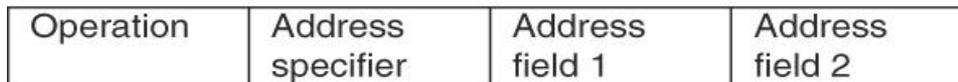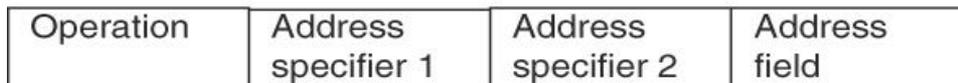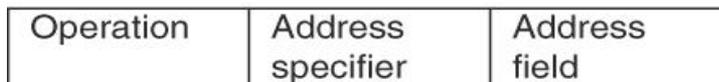
(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

**Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid.** The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

# Revisão: ISA MIPS64

# Conjunto de instruções: MIPS64

- Referência: Apêndice A (CAQA5)
- MIPS64 é usado em todo o curso como base para análise de todas as questões e problemas

## I-type instruction

| 6 | 5 | 5 | 16 |
|---|---|---|---|
| Opcode | rs | rt | Immediate |
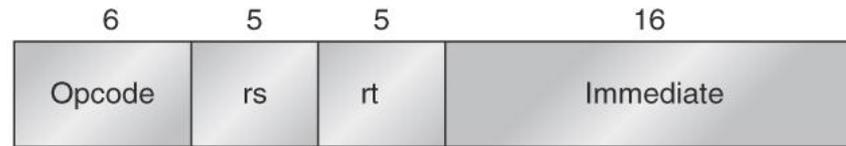
Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd=0, rs=destination, immediate=0)

## R-type instruction

| 6 | 5 | 5 | 5 | 5 | 6 |
|---|---|---|---|---|---|
| Opcode | rs | rt | rd | shamt | funct |
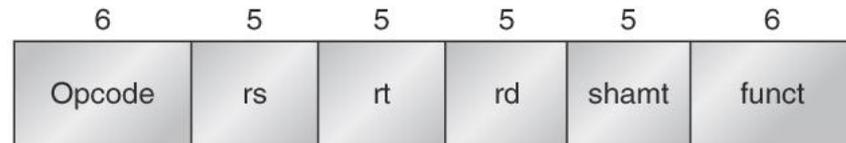
Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

## J-type instruction

| 6 | 26 |
|---|---|
| Opcode | Offset added to PC |

Jump and jump and link
Trap and return from exception

**Figure A.22 Instruction layout for MIPS.** All instructions are encoded in one of three types, with common fields in the same location in each format.

## Basic instruction formats

| R | opcode | rs | rt | rd | shamt | funct |
|---|--------|-----|-----|-----|-------|-------|

31        26 25        21 20        16 15        11 10        6 5        0

| I | opcode | rs | rt | immediate |
|---|--------|-----|-----|-----------|

31        26 25        21 20        16 15

| J | opcode | address |
|---|--------|---------|

31        26 25
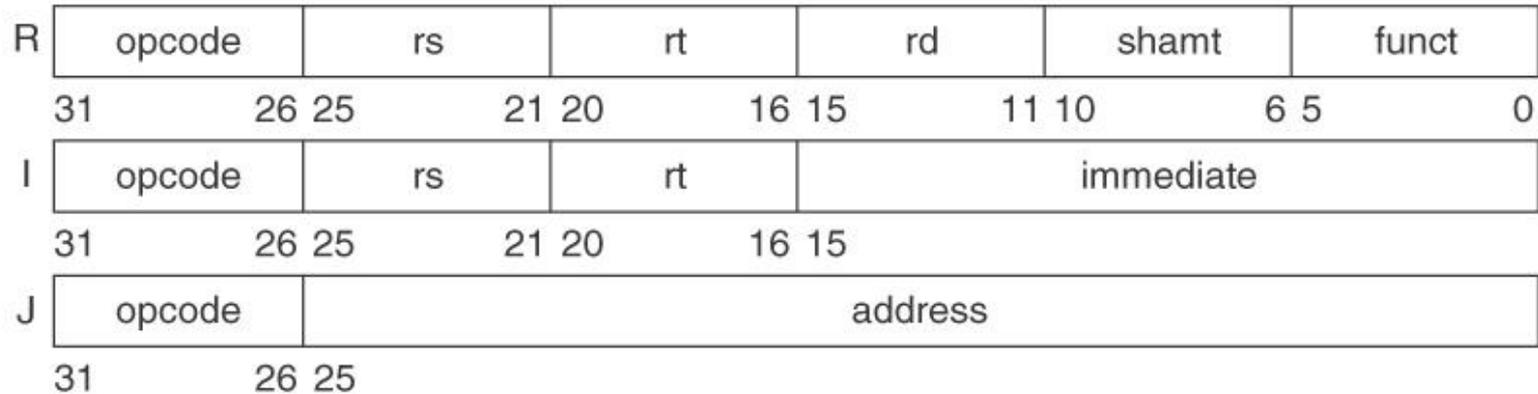
## Floating-point instruction formats

| FR | opcode | fmt | ft | fs | fd | funct |
|----|--------|-----|-----|-----|-----|-------|

31        26 25        21 20        16 15        11 10        6 5        0

| FI | opcode | fmt | ft | immediate |
|----|--------|-----|-----|-----------|

31        26 25        21 20        16 15
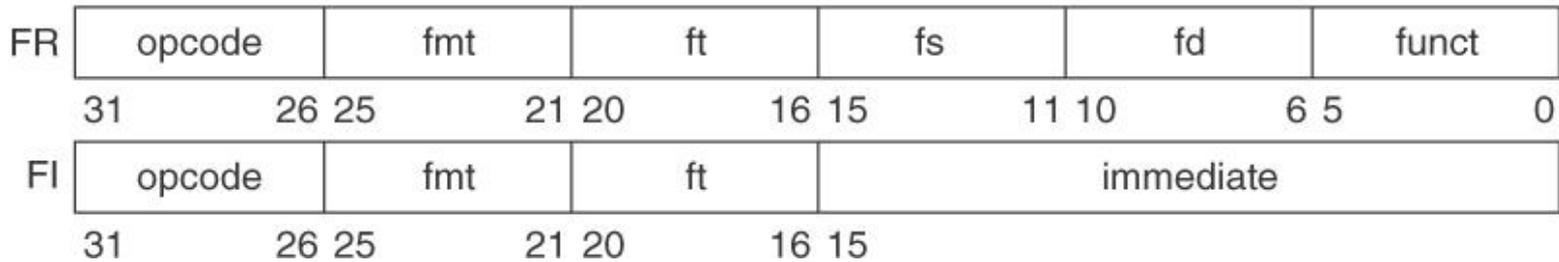
Figure 1.6 MIPS64 instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as DADDU, DSUBU, and so on. The I format is for data transfers, branches, and immediate instructions, such as LD, SD, BEQZ, and DADDIs. The J format is for jumps, the FR format for floating-point operations, and the FI format for floating-point branches.

# Registradores

- **Integer Registers**
  - 32 registradores de 64 bits: GPR (general purpose registers) → R0, R1, ... , R31
  - R0 = 0 (sempre)

- **Floating Point Registers**
  - 32 registradores de 64 bits: FPR → F0, F1, ... , F31
  - permitem armazenar 32 números FP de precisão simples (32b) ou dupla (64b)
  - se FPR contém nº de precisão simples → metade não é usada

- Há instruções para FPR ←→ GPR e para usar 2 dados c/ precisão simples em um único FPR

# Tipos de dados

- Bytes
- Meia palavra: 16 bits
- Palavra: 32 bits
- Palavra dupla: 64 bits
  - inteiros
  - FP precisão simples (32b) ou dupla (64b)
- Operações sobre dados
  - Palavra e palavra dupla
  - loads de bytes, meia palavras e palavras → MSB completados com zeros ou sign extend

# Modos de endereçamento

- ## Imediato (16bits) e displacement
  - Endereço = conteúdo registrador + imediato

- ## Para endereçamento indireto registrador Ri
  - fazer imediato = 0

- ## Endereço de 64 bits, byte addressable
  - Mode bit$\rightarrow$ SW pode selecionar big/little endian

# Operações

- 4 classes:
  - loads e stores ; ALU; controle de fluxo; ponto flutuante

- Notação
  - bits dos registradores: 0 (MSB) $\rightarrow$ 63 (LSB)
  - Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]] : transferência de 64 bits da posição de memória 30+R2
  - Regs[R4]$_0$ : bit 0 de R4
  - Regs[R3]$_{56..63}$ : byte menos significativo de R3
  - $0^{48}$ : campo com 48 zeros
  - a ## b:  a concatenado com b

# Load e Store

| Example instruction | Instruction name | Meaning |
|---|---|---|
| LD   R1,30(R2) | Load double word | $\text{Regs}[R1] \leftarrow_{64} \text{Mem}[30+\text{Regs}[R2]]$ |
| LD   R1,1000(R0) | Load double word | $\text{Regs}[R1] \leftarrow_{64} \text{Mem}[1000+0]$ |
| LW   R1,60(R2) | Load word | $\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[60+\text{Regs}[R2]]_0)^{32} \,\#\#\, \text{Mem}[60+\text{Regs}[R2]]$ |
| LB   R1,40(R3) | Load byte | $\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]]_0)^{56} \,\#\#\,$ $\text{Mem}[40+\text{Regs}[R3]]$ |
| LBU R1,40(R3) | Load byte unsigned | $\text{Regs}[R1] \leftarrow_{64} 0^{56} \,\#\#\, \text{Mem}[40+\text{Regs}[R3]]$ |
| LH   R1,40(R3) | Load half word | $\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]]_0)^{48} \,\#\#\,$ $\text{Mem}[40+\text{Regs}[R3]] \,\#\#\, \text{Mem}[41+\text{Regs}[R3]]$ |
| L.S F0,50(R3) | Load FP single | $\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R3]] \,\#\#\, 0^{32}$ |
| L.D F0,50(R2) | Load FP double | $\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R2]]$ |
| SD   R3,500(R4) | Store double word | $\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{64} \text{Regs}[R3]$ |
| SW   R3,500(R4) | Store word | $\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{32} \text{Regs}[R3]_{32..63}$ |
| S.S F0,40(R3) | Store FP single | $\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{32} \text{Regs}[F0]_{0..31}$ |
| S.D F0,40(R3) | Store FP double | $\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{64} \text{Regs}[F0]$ |
| SH   R3,502(R2) | Store half | $\text{Mem}[502+\text{Regs}[R2]] \leftarrow_{16} \text{Regs}[R3]_{48..63}$ |
| SB   R2,41(R3) | Store byte | $\text{Mem}[41+\text{Regs}[R3]] \leftarrow_{8} \text{Regs}[R2]_{56..63}$ |

**Figure A.23** The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

# ALU

| Example instruction | Instruction name | Meaning |
|---|---|---|
| DADDU   R1,R2,R3 | Add unsigned | $Regs[R1] \leftarrow Regs[R2]+Regs[R3]$ |
| DADDIU R1,R2,#3 | Add immediate unsigned | $Regs[R1] \leftarrow Regs[R2]+3$ |
| LUI     R1,#42 | Load upper immediate | $Regs[R1] \leftarrow 0^{32}\#\#42\#\#0^{16}$ |
| DSLL    R1,R2,#5 | Shift left logical | $Regs[R1] \leftarrow Regs[R2]<<5$ |
| SLT     R1,R2,R3 | Set less than | if $(Regs[R2]<Regs[R3])$ $Regs[R1] \leftarrow 1$ else $Regs[R1] \leftarrow 0$ |

**Figure A.24** Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

# Controle de fluxo

| Example instruction | Instruction name | Meaning |
|---|---|---|
| J name | Jump | $PC_{36..63} \leftarrow name$ |
| JAL name | Jump and link | $Regs[R31] \leftarrow PC+8;\ PC_{36..63} \leftarrow name;$ <br> $((PC+4)-2^{27}) \le name < ((PC+4)+2^{27})$ |
| JALR R2 | Jump and link register | $Regs[R31] \leftarrow PC+8;\ PC \leftarrow Regs[R2]$ |
| JR R3 | Jump register | $PC \leftarrow Regs[R3]$ |
| BEQZ R4,name | Branch equal zero | $if\ (Regs[R4]==0)\ PC \leftarrow name;$ <br> $((PC+4)-2^{17}) \le name < ((PC+4)+2^{17})$ |
| BNE R3,R4,name | Branch not equal zero | $if\ (Regs[R3] != Regs[R4])\ PC \leftarrow name;$ <br> $((PC+4)-2^{17}) \le name < ((PC+4)+2^{17})$ |
| MOVZ R1,R2,R3 | Conditional move if zero | $if\ (Regs[R3]==0)\ Regs[R1] \leftarrow Regs[R2]$ |

**Figure A.25 Typical control flow instructions in MIPS.** All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggest; since MIPS instructions are all 32 bits long, the byte branch address is multiplied by 4 to get a longer distance.

# Ponto flutuante

| Floating point | FP operations on DP and SP formats |
|---|---|
| ADD.D,ADD.S,ADD.PS | Add DP, SP numbers, and pairs of SP numbers |
| SUB.D,SUB.S,SUB.PS | Subtract DP, SP numbers, and pairs of SP numbers |
| MUL.D,MUL.S,MUL.PS | Multiply DP, SP floating point, and pairs of SP numbers |
| MADD.D,MADD.S,MADD.PS | Multiply-add DP, SP numbers, and pairs of SP numbers |
| DIV.D,DIV.S,DIV.PS | Divide DP, SP floating point, and pairs of SP numbers |
| CVT._._ | Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs. |
| C.__.D,C.__.S | DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register |

| Instruction type/opcode | Instruction meaning |
|---|---|
| *Data transfers* | *Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-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, load word unsigned, store word (to/from integer registers) |
| LD,SD | Load double word, store double word (to/from integer registers) |
| L.S,L.D,S.S,S.D | Load SP float, load DP float, store SP float, store DP float |
| MFC0,MTC0 | Copy from/to GPR to/from a special register |
| MOV.S,MOV.D | Copy one SP or DP FP register to another FP register |
| MFC1,MTC1 | Copy 32 bits to/from FP registers from/to integer registers |
| *Arithmetic/logical* | *Operations on integer or logical data in GPRs; signed arithmetic trap on overflow* |
| DADD,DADDI,DADDU,DADDIU | Add, add immediate (all immediates are 16 bits); signed and unsigned |
| DSUB,DSUBU | Subtract; signed and unsigned |
| DMUL,DMULU,DDIV, DDIVU,MADD | Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values |
| AND,ANDI | And, and immediate |
| OR,ORI,XOR,XORI | Or, or immediate, exclusive or, exclusive or immediate |
| LUI | Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends |
| DSLL,DSRL,DSRA,DSLLV, DSRLV,DSRAV | Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic |
| SLT,SLTI,SLTU,SLTIU | Set less than, set less than immediate; signed and unsigned |
| *Control* | *Conditional branches and jumps; PC-relative or through register* |
| BEQZ,BNEZ | Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4 |
| BEQ,BNE | Branch GPR equal/not equal; 16-bit offset from PC + 4 |
| BC1T,BC1F | Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4 |
| MOVN,MOVZ | Copy GPR to another GPR if third GPR is negative, zero |
| J,JR | Jumps: 26-bit offset from PC + 4 (J) or target in register (JR) |
| JAL,JALR | Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR) |
| TRAP | Transfer to operating system at a vectored address |
| ERET | Return to user code from an exception; restore user mode |
| *Floating point* | *FP operations on DP and SP formats* |
| ADD.D,ADD.S,ADD.PS | Add DP, SP numbers, and pairs of SP numbers |
| SUB.D,SUB.S,SUB.PS | Subtract DP, SP numbers, and pairs of SP numbers |
| MUL.D,MUL.S,MUL.PS | Multiply DP, SP floating point, and pairs of SP numbers |
| MADD.D,MADD.S,MADD.PS | Multiply-add DP, SP numbers, and pairs of SP numbers |
| DIV.D,DIV.S,DIV.PS | Divide DP, SP floating point, and pairs of SP numbers |
| CVT._._ | Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs. |
| C.__.D,C.__.S | DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register |

# ISA de outros computadores

- Ver apêndice K