

CS 451 / 551 / ECE 541

ADVANCED
COMPUTER ARCHITECTURE

SESSION no. 5

University of Idaho

BOB RINKER

JEB 235

RINKER@CS.UIDAHO.EDU

Addressing mode	Example instruction	Meaning	When used
Register	Add R4, R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4, #3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4, 100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4, (R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3, (R1 + R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	Add R1, (1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1, @(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$.
Autoincrement	Add R1, (R2) +	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$	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)	$\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1, 100(R2) [R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	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.

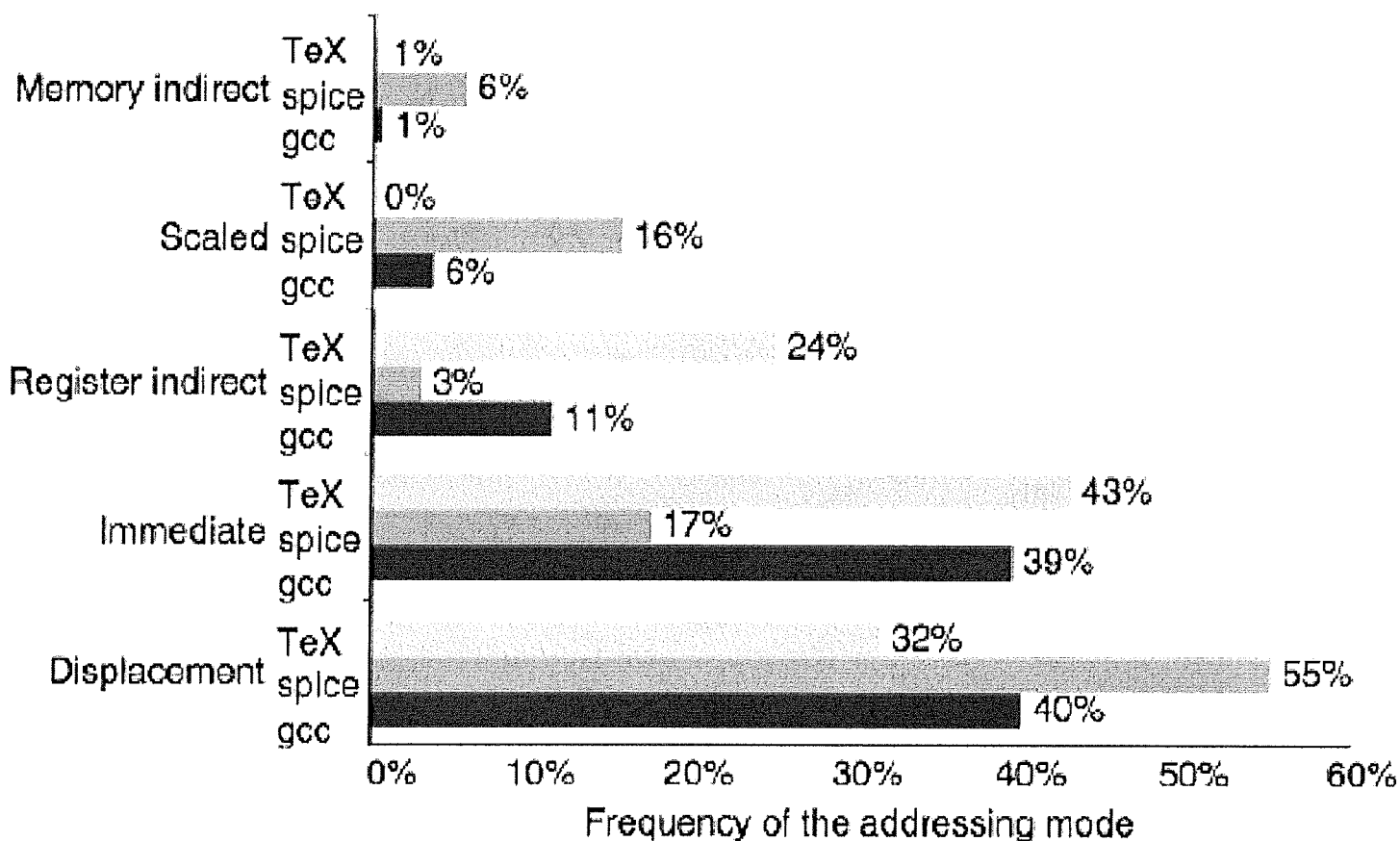


Figure A.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including 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.

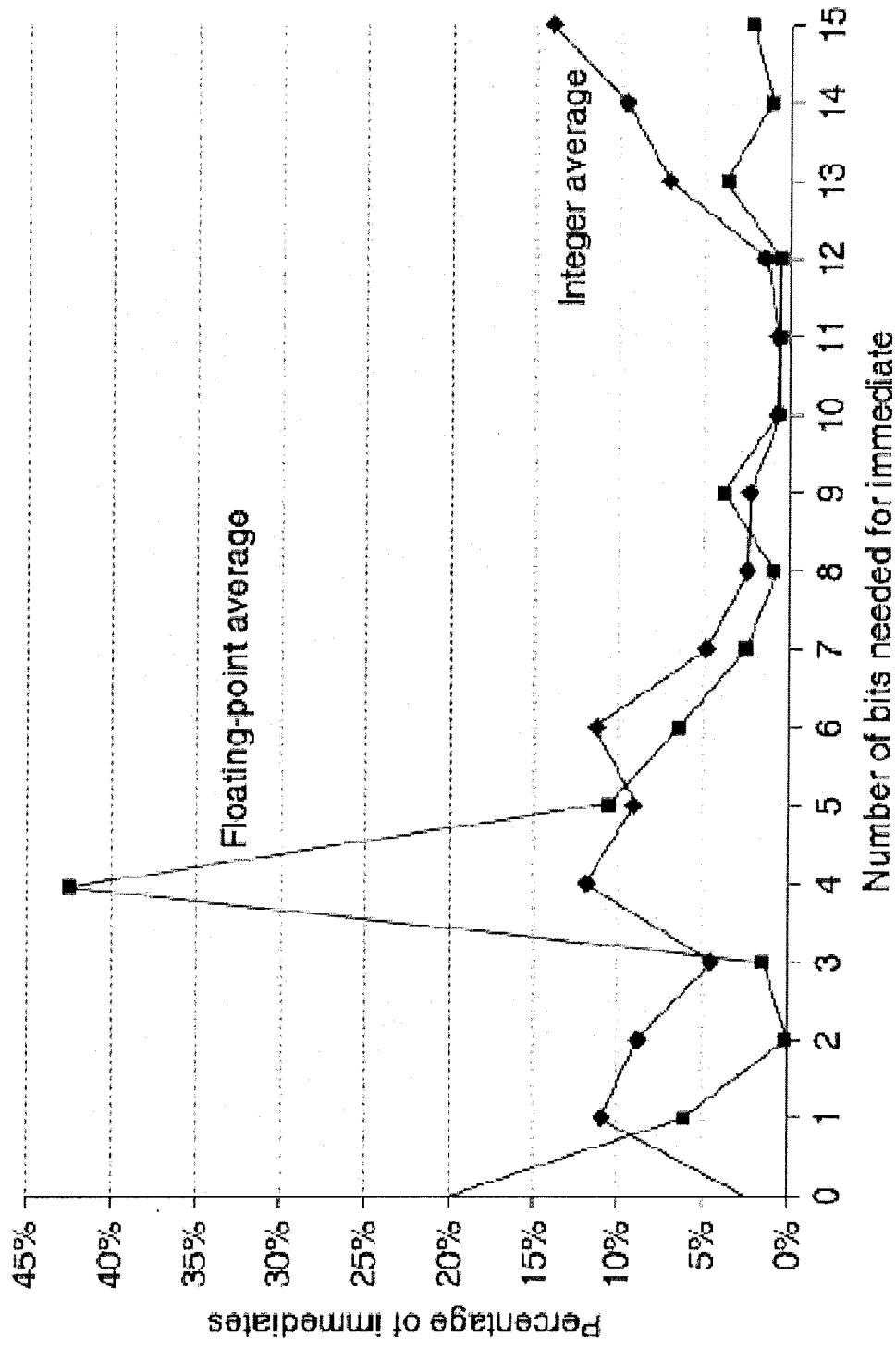


Figure A.10 The distribution of immediate values. The x-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for 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%.

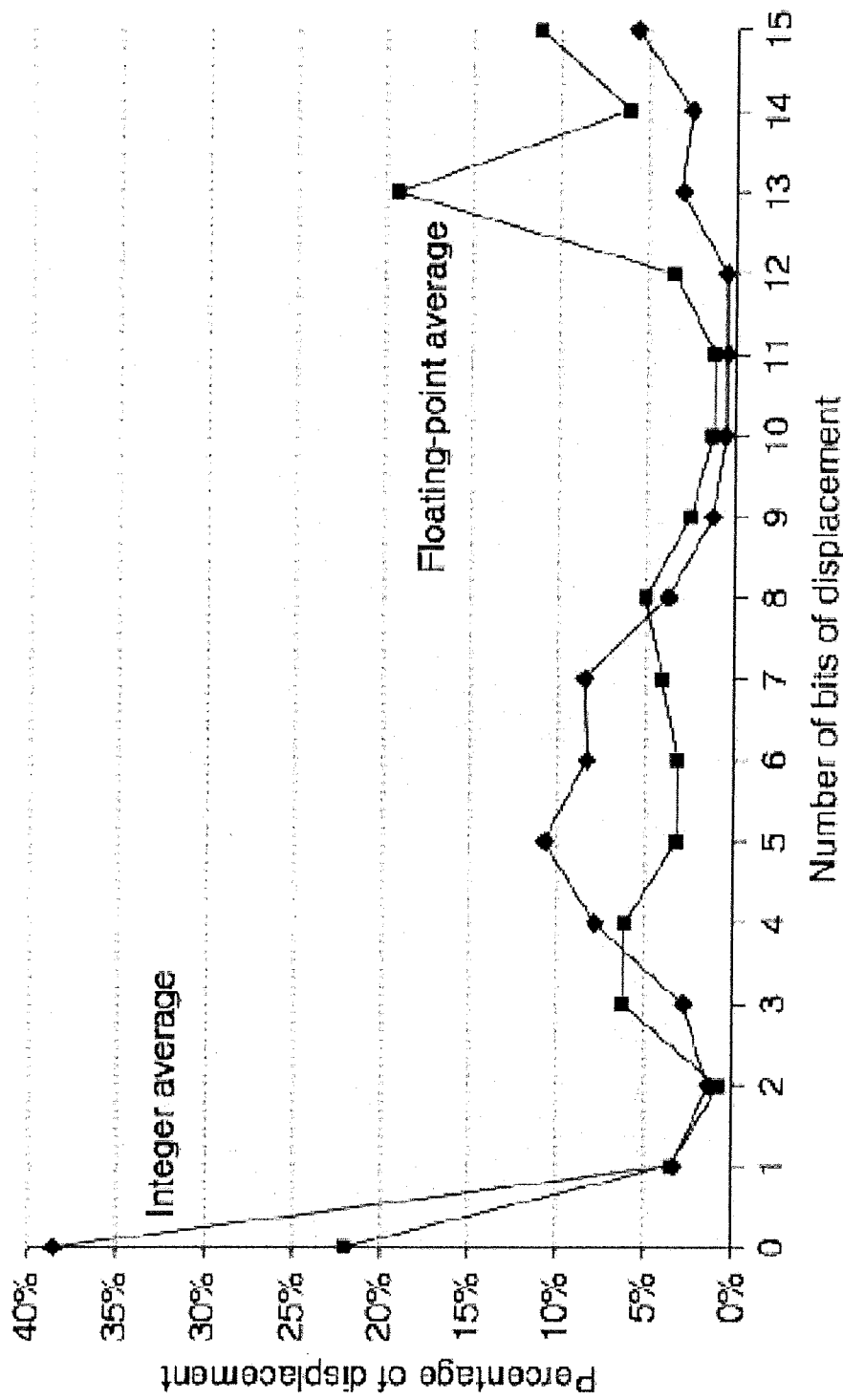


Figure A.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see Section A.8) as well as the overall addressing scheme the compiler uses. The x-axis is \log_2 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).

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)

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.

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

6066
4506

Figure A.12 Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any computer that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel—for example, performing eight 8-bit additions on two 64-bit operands.

Instruction	gap	gcc	gzip	mcf	perlbnk	Integer average
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	2.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
AND	4.3%	4.6%	9.4%	0.2%	1.2%	4%
OR	7.9%	8.5%	4.8%	17.6%	8.7%	9%
XOR	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

Figure A.27 MIPS dynamic instruction mix for five SPECint2000 programs. Note that integer register-register move instructions are included in the OR instruction. Blank entries have the value 0.0%.

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.

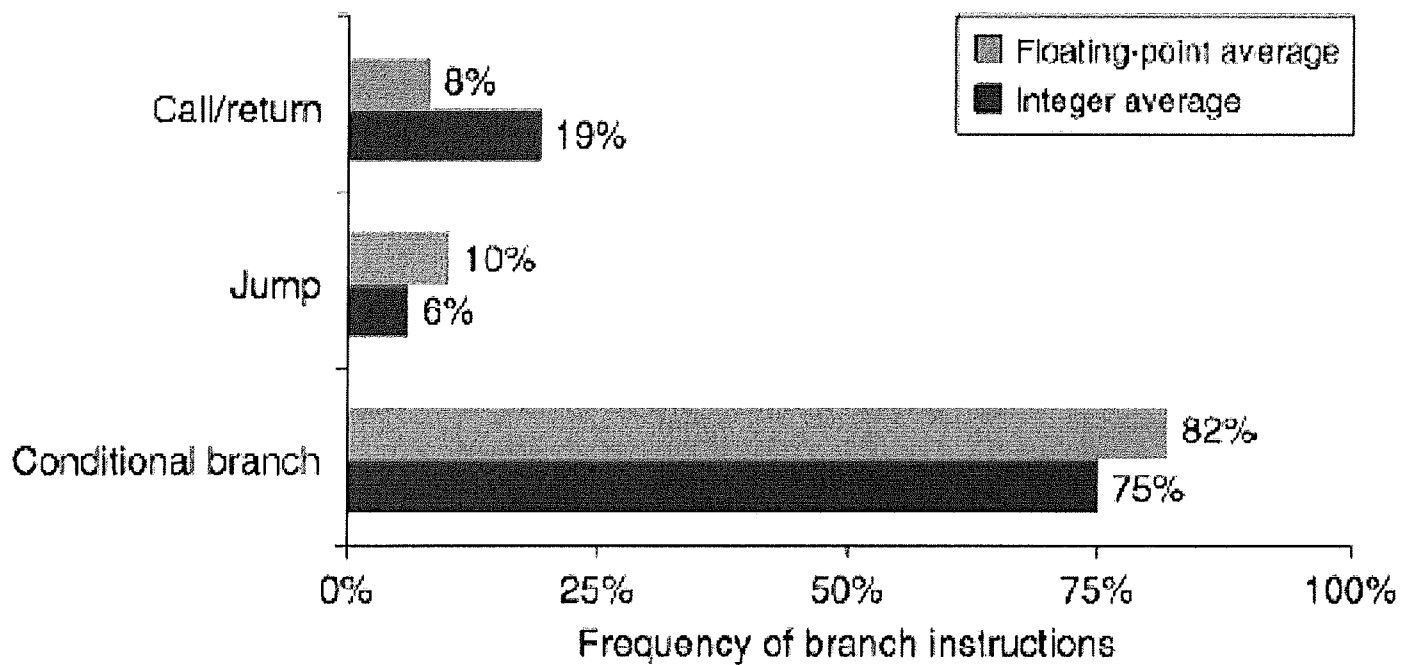


Figure A.14 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure A.8.

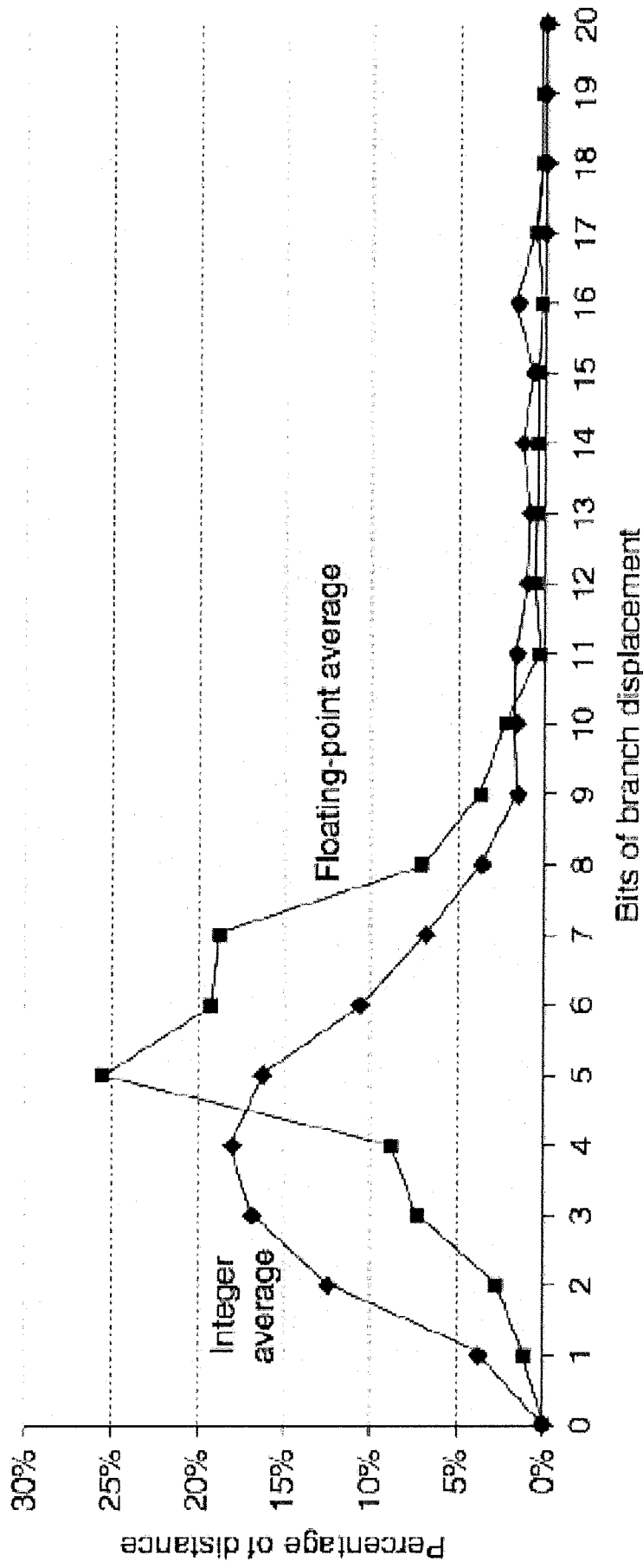
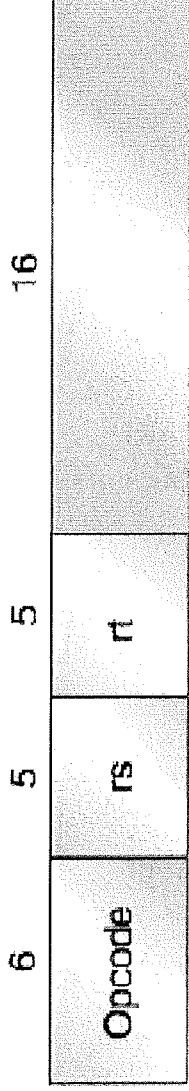


Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4 to 8 bits. This result tells us that short displacement fields often suffice for branches and that 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.

I-type instruction

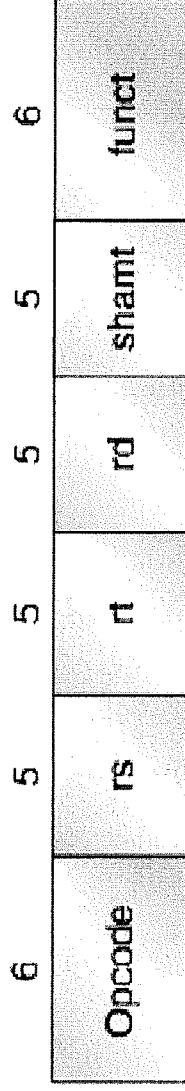


Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

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

LOAD
PADD
RADD

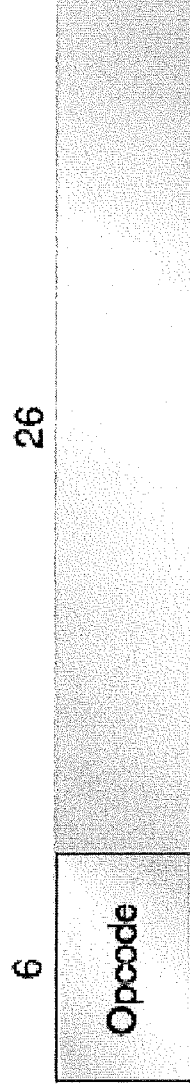
R-type instruction



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



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.

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

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

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	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{48} \text{ \#\# Mem}[40+\text{Regs}[R3]]$
L.S F0, 50(R3)	Load FP single	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R3]] \text{ \#\# } 0^{32}$
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.

Instruction type/opcode	Instruction meaning
<i>Data transfers</i>	
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
<i>Arithmetic/Logical</i>	
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
<i>Control</i>	
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, MOYZ	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
<i>Floating point</i>	
<i>FP operations on DP and SP formats</i>	
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

Figure A.26 Subset of the instructions in MIPS64. Figure A.22 lists the formats of these instructions. SP = single precision; DP = double precision. This list can also be found on the back inside cover.

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

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.