Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input. See Appendix B.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1` = `$s2` + 100	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	`$s1` = `$s2` + 100	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	`$s1` = `$epc`	Copy Exception PC + special regs
	multiply	`mult $s2,$s3`	Hi, Lo = `$s2` × `$s3`	64-bit signed product in Hi, Lo
	multiply unsigned	`multu $s2,$s3`	Hi, Lo = `$s2` × `$s3`	64-bit unsigned product in Hi, Lo
	divide	`div $s2,$s3`	Lo = `$s2` / `$s3`, Hi = `$s2` mod `$s3`	Lo = quotient, Hi = remainder
	divide unsigned	`divu $s2,$s3`	Lo = `$s2` / `$s3`, Hi = `$s2` mod `$s3`	Unsigned quotient and remainder
	move from Hi	`mfhi $s1`	`$s1` = Hi	Used to get copy of Hi
	move from Lo	`mflo $s1`	`$s1` = Lo	Used to get copy of Lo
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store conditional word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1`=0 or 1	Store word as 2nd half atomic swap
	load upper immediate	`lui $s1,100`	`$s1` = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	AND	`AND $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	OR	`OR $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	NOR	`NOR $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	Three reg. operands; bit-by-bit NOR
	AND immediate	`ANDi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND with constant
	OR immediate	`ORi $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1` != `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if (`$s2` < 100) `$s1` = 1; else `$s1`=0	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1`=0	Compare less than; natural numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if (`$s2` < 100) `$s1` = 1; else `$s1` = 0	Compare < constant; natural numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

FIGURE 3.12 MIPS core architecture. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the Hi and Lo registers to support multiply and divide. MIPS machine language is listed in the MIPS Reference Data Card at the front of this book.

MIPS Addressing Mode Summary

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

Multiple forms of addressing are generically called **addressing modes**. Figure 2.18 shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

1. *Immediate addressing,* where the operand is a constant within the instruction itself
2. *Register addressing,* where the operand is a register
3. *Base* or *displacement addressing,* where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. *PC-relative addressing,* where the branch address is the sum of the PC and a constant in the instruction
5. *Pseudodirect addressing,* where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

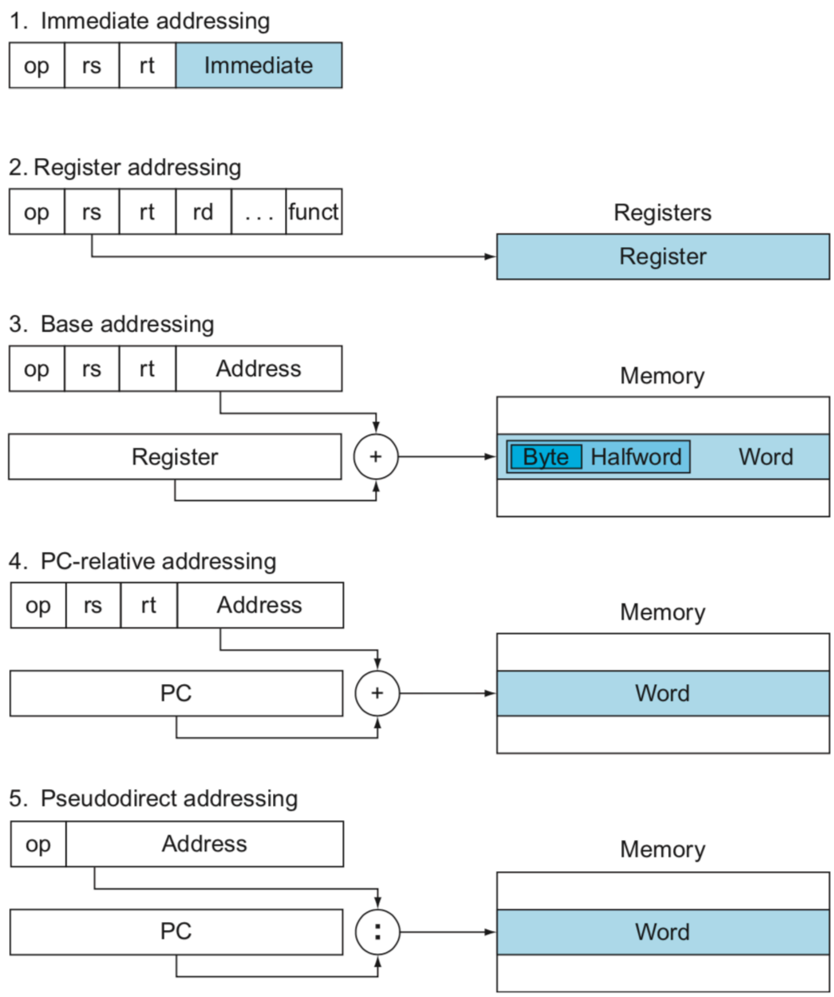

FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	`shamt`	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, `imm`. format
J-format	op	target address					Jump instruction format

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Remaining MIPS-32	Name	Format
exclusive or (*rs* ⊕ *rt*)	`xor`	R
exclusive or immediate	`xori`	I
shift right arithmetic	`sra`	R
shift left logical variable	`sllv`	R
shift right logical variable	`srlv`	R
shift right arithmetic variable	`srav`	R
move to Hi	`mthi`	R
move to Lo	`mtlo`	R
load halfword	`lh`	I
load byte	`lb`	I
load word left *(unaligned)*	`lwl`	I
load word right *(unaligned)*	`lwr`	I
store word left *(unaligned)*	`swl`	I
store word right *(unaligned)*	`swr`	I
load linked (*atomic update*)	`ll`	I
store cond. (*atomic update*)	`sc`	I
move if zero	`movz`	R
move if not zero	`movn`	R
multiply and add (S or *u*ns.)	`madd`*s*	R
multiply and subtract (S or *u*ns.)	`msub`*s*	I
branch on ≥ zero and link	`bgezal`	I
branch on < zero and link	`bltzal`	I
jump and link register	`jalr`	R
branch compare to zero	`bxz`	I
branch compare to zero likely	`bxzl`	I
(x = `lt`, `le`, `gt`, `ge`)		
branch compare reg likely	`bxl`	I
trap if compare reg	`tx`	R
trap if compare immediate	`txi`	I
(x = `eq`, `neq`, `lt`, `le`, `gt`, `ge`)		
return from exception	`rfe`	R
system call	`syscall`	I
break *(cause exception)*	`break`	I
move from FP to integer	`mfc1`	R
move to FP from integer	`mtc1`	R
FP move (*s or d*)	`mov.`*f*	R
FP move if zero (*s or d*)	`movz.`*f*	R
FP move if not zero (*s or d*)	`movn.`*f*	R
FP square root (*s or d*)	`sqrt.`*f*	R
FP absolute value (*s or d*)	`abs.`*f*	R
FP negate (*s or d*)	`neg.`*f*	R
FP convert (*w, s, or d*)	`cvt.`*f.f*	R
FP compare un (*s or d*)	`c.xn.`*f*	R

Pseudo MIPS	Name	Format
absolute value	`abs`	rd,rs
negate *(signed or unsigned)*	`neg`*s*	rd,rs
rotate left	`rol`	rd,rs,rt
rotate right	`ror`	rd,rs,rt
multiply and don't check oflw *(signed or uns.)*	`mul`*s*	rd,rs,rt
multiply and check oflw *(signed or uns.)*	`mulo`*s*	rd,rs,rt
divide and check overflow	`div`	rd,rs,rt
divide and don't check overflow	`divu`	rd,rs,rt
remainder *(signed or unsigned)*	`rem`*s*	rd,rs,rt
load immediate	`li`	rd,imm
load address	`la`	rd,addr
load double	`ld`	rd,addr
store double	`sd`	rd,addr
unaligned load word	`ulw`	rd,addr
unaligned store word	`usw`	rd,addr
unaligned load halfword *(signed or uns.)*	`ulh`*s*	rd,addr
unaligned store halfword	`ush`	rd,addr
branch	`b`	Label
branch on equal zero	`beqz`	rs,L
branch on compare *(signed or unsigned)*	`bx`*s*	rs,rt,L
(x = `lt`, `le`, `gt`, `ge`)		
set equal	`seq`	rd,rs,rt
set not equal	`sne`	rd,rs,rt
set on compare *(signed or unsigned)*	`sx`*s*	rd,rs,rt
(x = `lt`, `le`, `gt`, `ge`)		
load to floating point (*s or d*)	`l.`*f*	rd,addr
store from floating point (*s or d*)	`s.`*f*	rd,addr

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

op(31:26)

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)

23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								