Chapter 3

Arithmetic for Computers

Copyright © 2014 Elsevier Inc. All rights reserved.

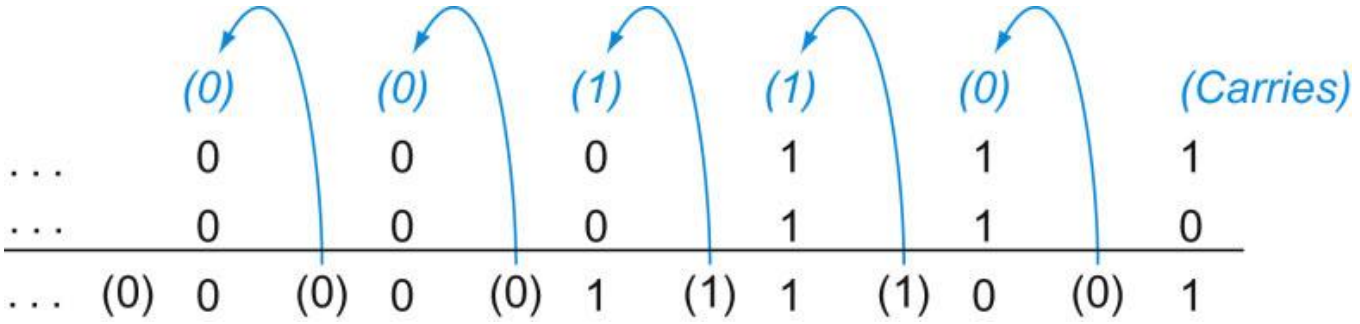


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 1 1 1 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 1 1 1 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 1 0 1 0, yielding a 1 sum and no carry.

Copyright © 2014 Elsevier Inc. All rights reserved.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

Copyright © 2014 Elsevier Inc. All rights reserved.

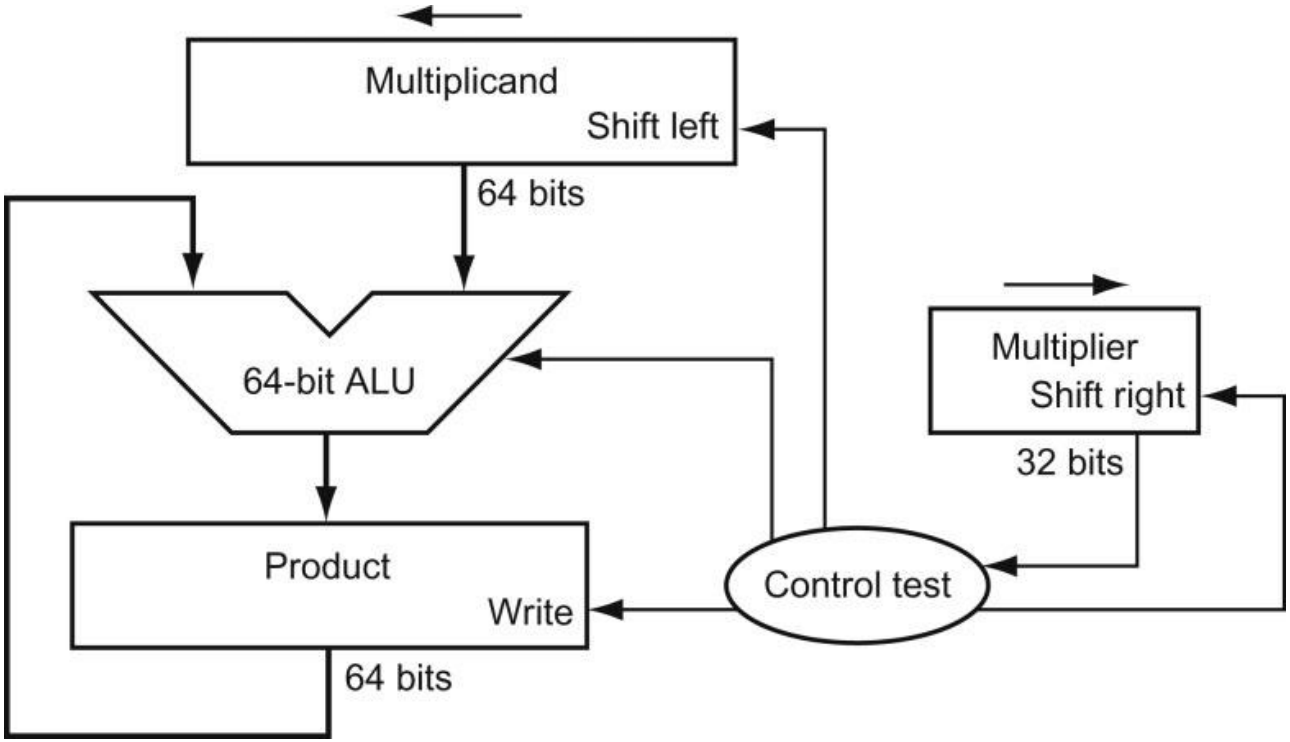


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Copyright © 2014 Elsevier Inc. All rights reserved.

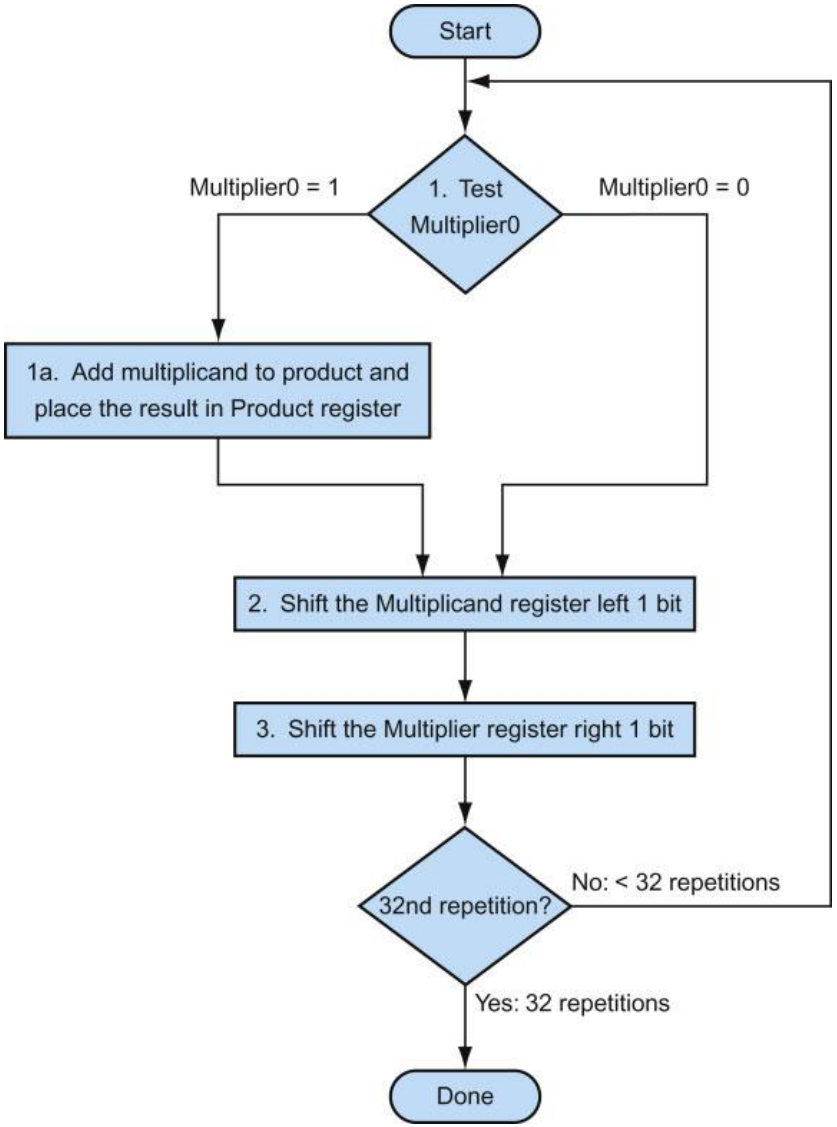


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

Copyright © 2014 Elsevier Inc. All rights reserved.

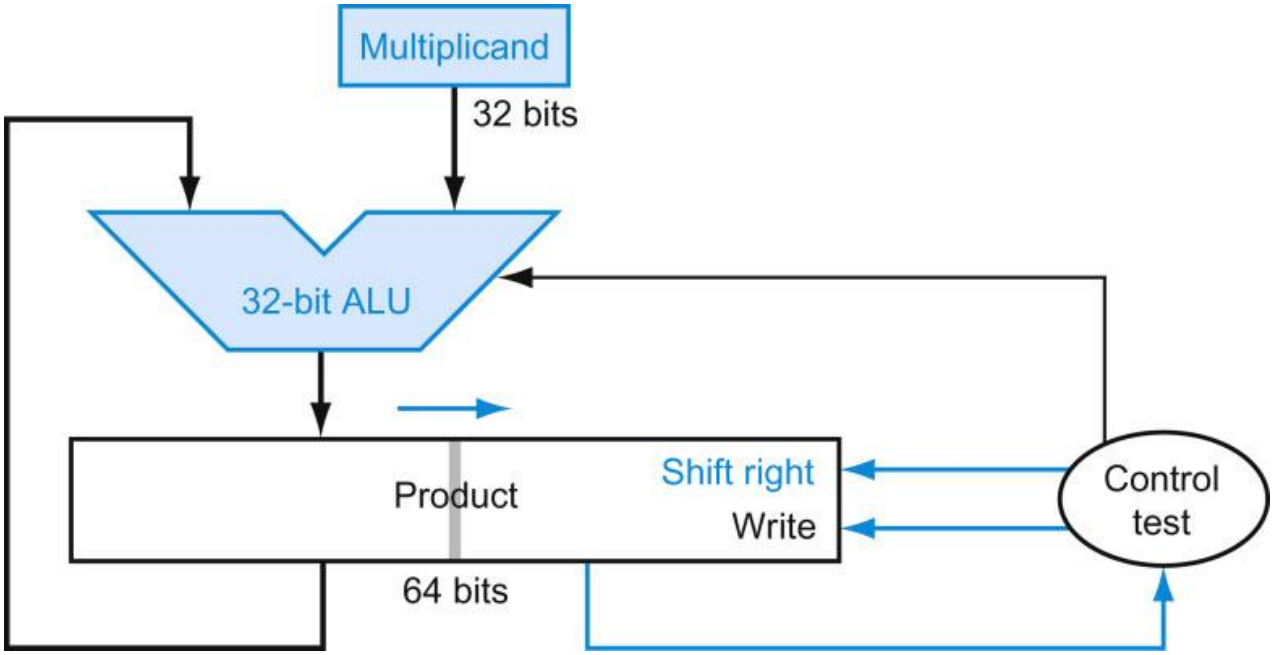


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Copyright © 2014 Elsevier Inc. All rights reserved.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Copyright © 2014 Elsevier Inc. All rights reserved.

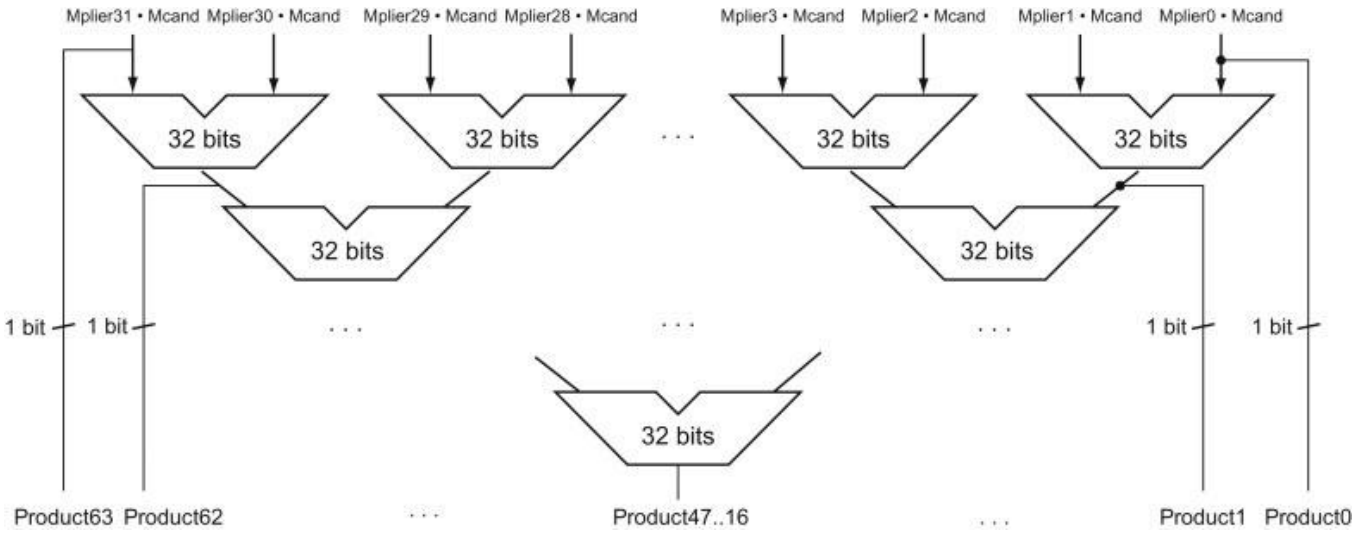


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.

Copyright © 2014 Elsevier Inc. All rights reserved.

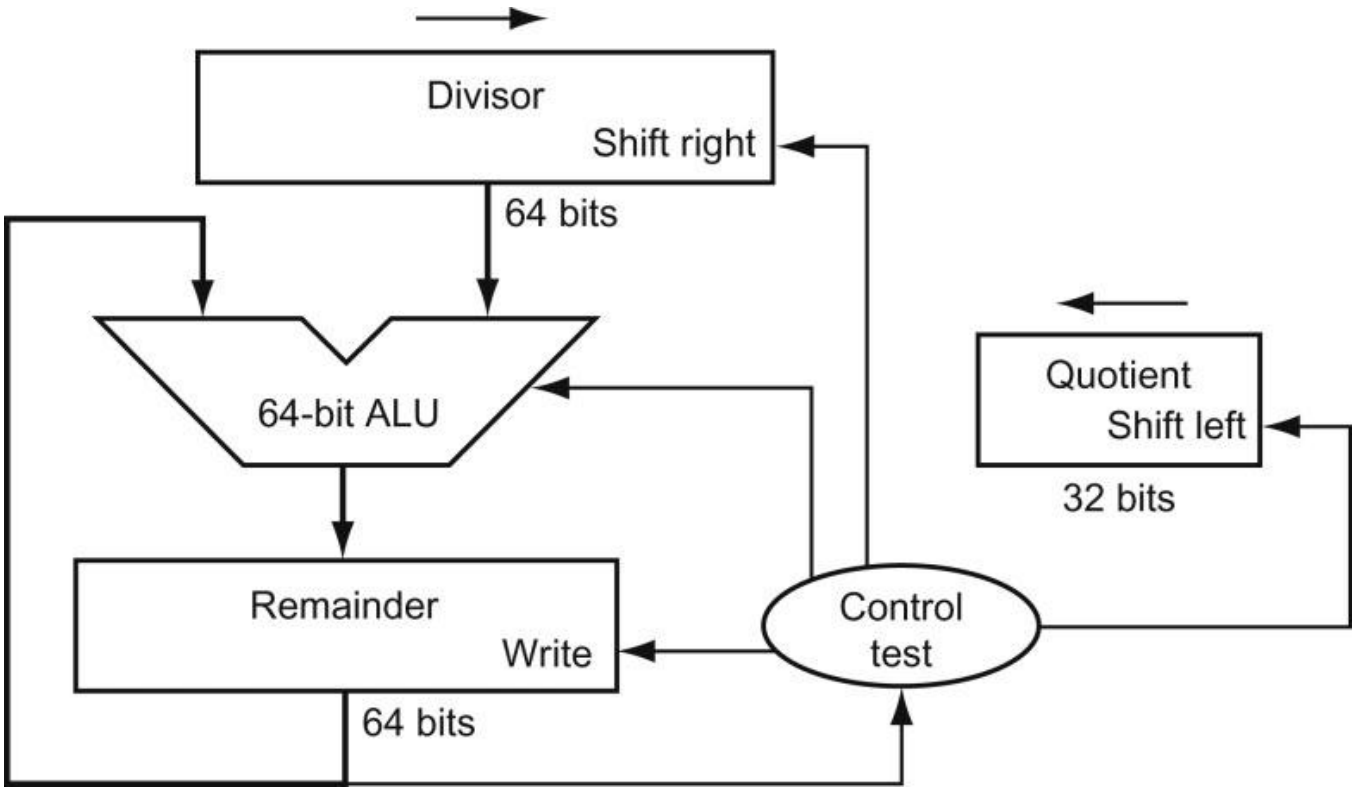


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Copyright © 2014 Elsevier Inc. All rights reserved.

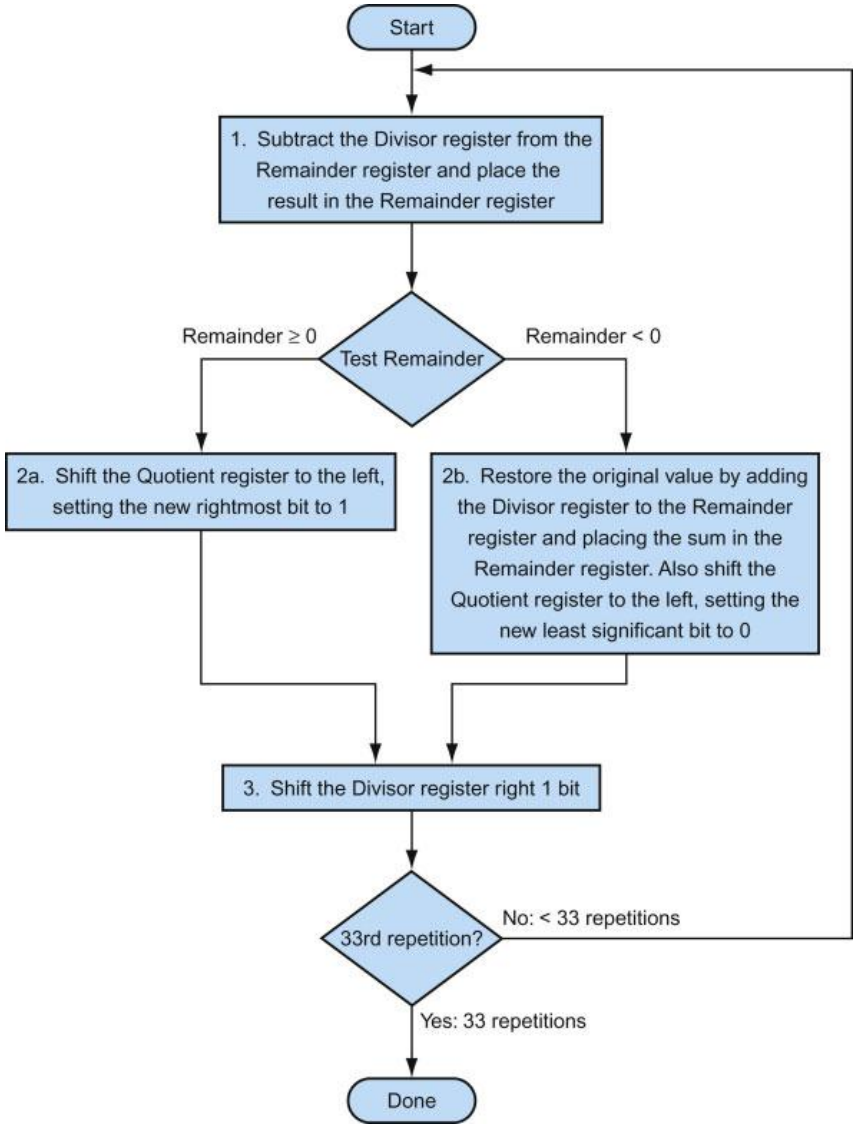


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

Copyright © 2014 Elsevier Inc. All rights reserved.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	(1)111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	(1)111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	(0)000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	(0)000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Copyright © 2014 Elsevier Inc. All rights reserved.

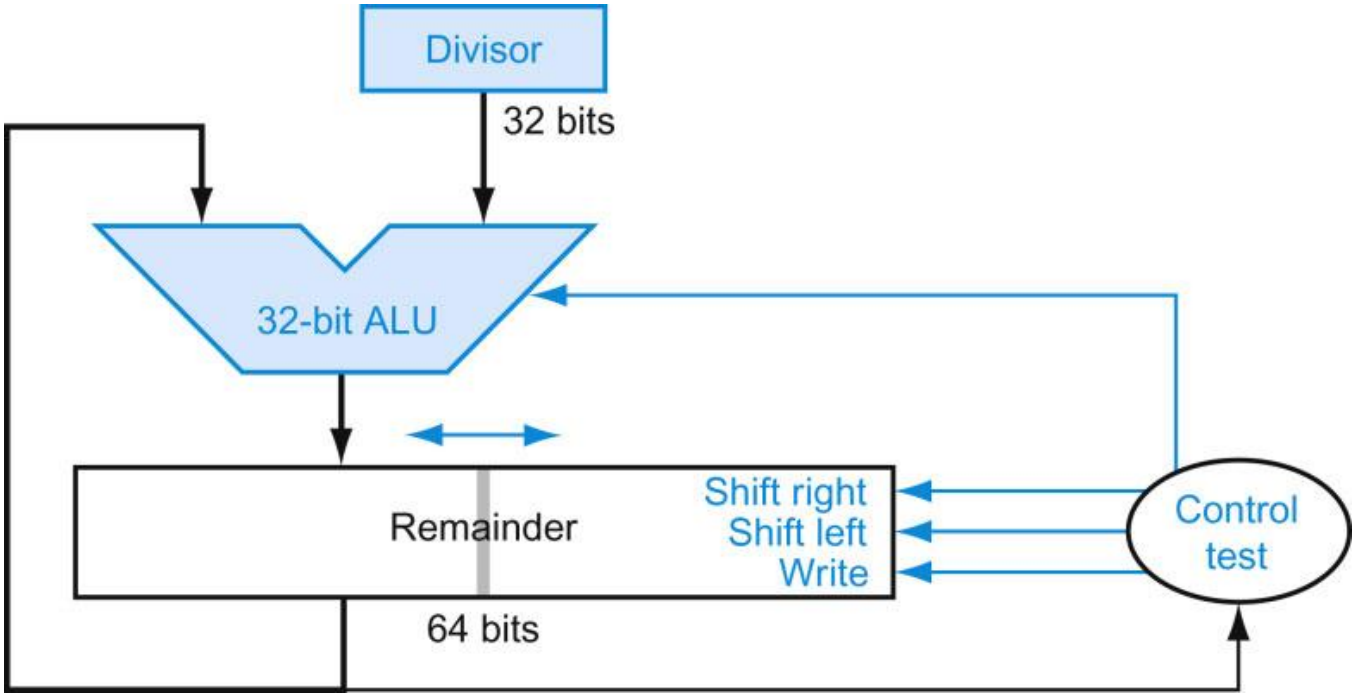


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.5, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

Copyright © 2014 Elsevier Inc. All rights reserved.

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
Condi-tional 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
Uncondi-tional 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.

Copyright © 2014 Elsevier Inc. All rights reserved.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Copyright © 2014 Elsevier Inc. All rights reserved.

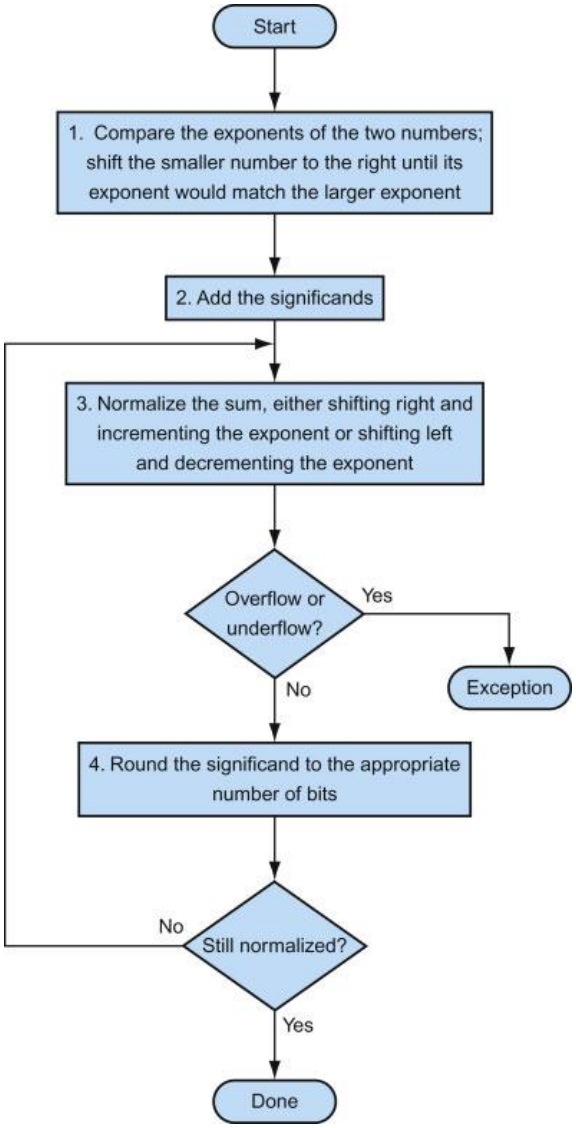


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Copyright © 2014 Elsevier Inc. All rights reserved.

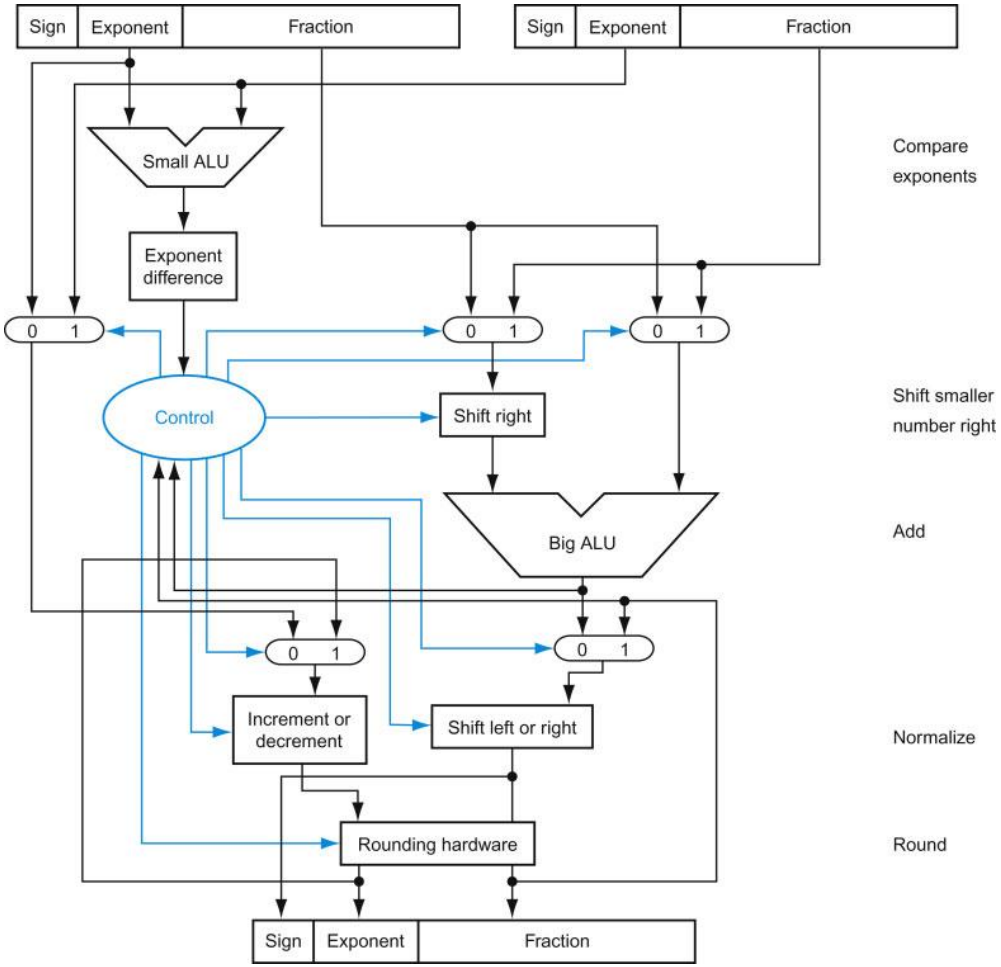


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.

Copyright © 2014 Elsevier Inc. All rights reserved.

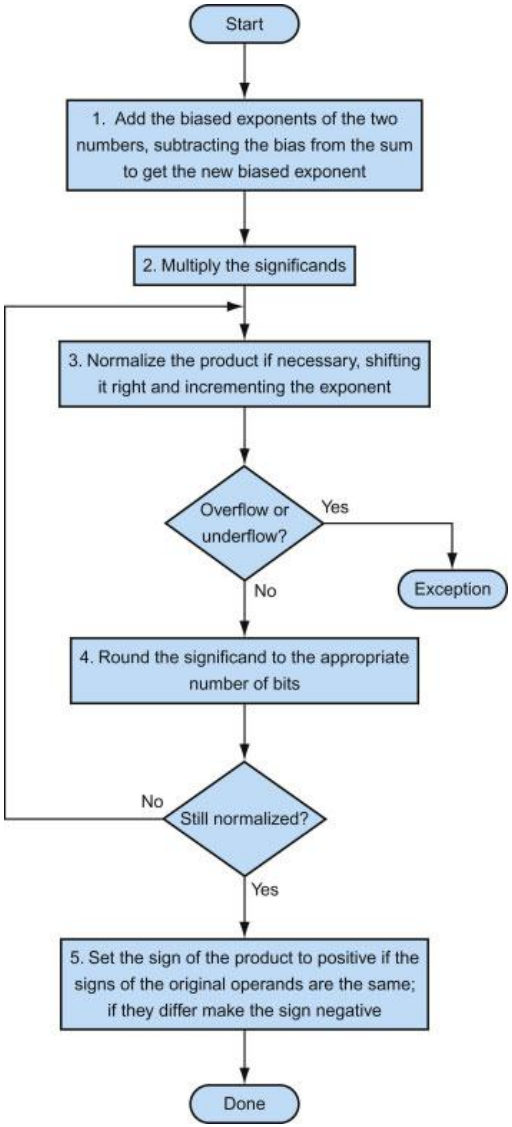


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Copyright © 2014 Elsevier Inc. All rights reserved.

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

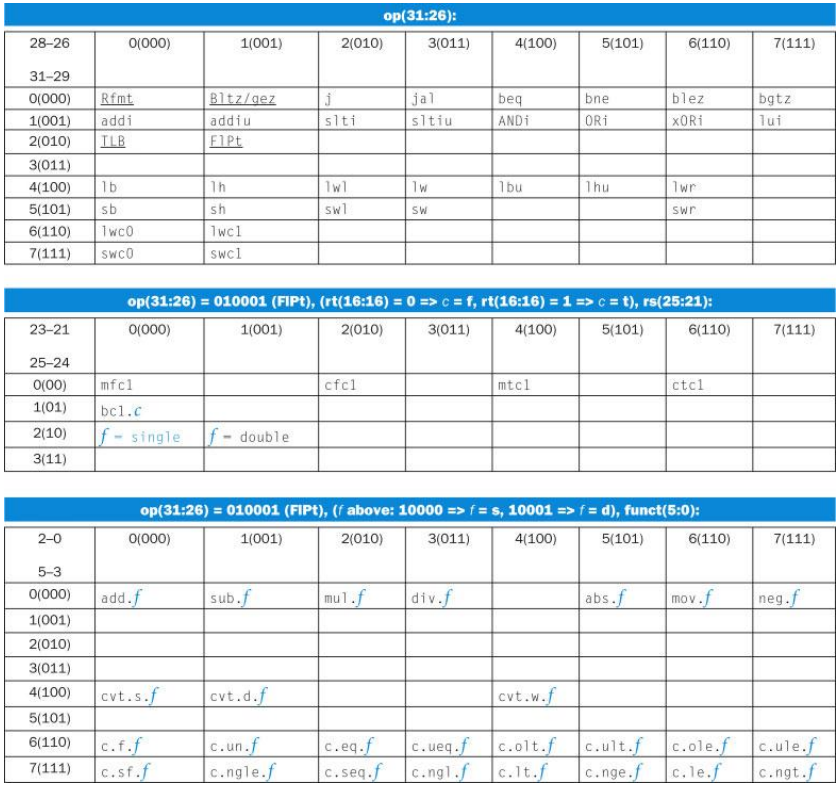
Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

FIGURE 3.17 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, for more detail. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

Copyright © 2014 Elsevier Inc. All rights reserved.



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)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	ANDi	ORi	xORi	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => c = f, rt(16:16) = 1 => c = t), 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)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.c							
2(10)	f = single	f = double						
3(11)								

op(31:26) = 010001 (FlPt), (f above: 10000 => f = s, 10001 => f = d), 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)	add.f	sub.f	mul.f	div.f		abs.f	mov.f	neg.f
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.f	cvt.d.f			cvt.w.f			
5(101)								
6(110)	c.f.f	c.un.f	c.eq.f	c.ueq.f	c.olt.f	c.ult.f	c.ole.f	c.ule.f
7(111)	c.sf.f	c.ngle.f	c.seq.f	c.ngl.f	c.lt.f	c.nge.f	c.le.f	c.ngt.f

FIGURE 3.18 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure, lw is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op 5 010001_{two}) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction) is 000001_{two} and the op field (bits 31–26) is 010001_{two}. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (*f* 5 *s*, so rs 5 10000) or double precision (*f* 5 *d*, so rs 5 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 5 1 5 .bc1.t) or false (bit 16 5 0 5 . bc1.f). Instructions in color are described in Chapter 2 or this chapter, with Appendix A covering all instructions. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

Copyright © 2014 Elsevier Inc. All rights reserved.

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3.4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3.4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

Copyright © 2014 Elsevier Inc. All rights reserved.

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm,mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm,mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm,mem/xmm	
	DIV{SS/PS/SD/PD} xmm,mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURE 3.20 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

Copyright © 2014 Elsevier Inc. All rights reserved.

```
1.  void dgemm (int n, double* A, double* B, double* C)
2.  {
3.     for (int i = 0; i < n; ++i)
4.       for (int j = 0; j < n; ++j)
5.       {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for( int k = 0; k < n; k++ )
8.           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.      }
11. }
```

FIGURE 3.21 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply (GEMM). Because we are passing the matrix dimension as the parameter n, this version of DGEMM uses single dimensional versions of matrices C, A, and B and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

Copyright © 2014 Elsevier Inc. All rights reserved.

```
 1.  vmovsd (%r10),%xmm0              # Load 1 element of C into %xmm0
 2.  mov    %rsi,%rcx                 # register %rcx = %rsi
 3.  xor    %eax,%eax                 # register %eax = 0
 4.  vmovsd (%rcx),%xmm1              # Load 1 element of B into %xmm1
 5.  add    %r9,%rcx                  # register %rcx = %rcx + %r9
 6.  vmulsd (%r8,%rax,8),%xmm1,%xmm1  # Multiply %xmm1, element of A
 7.  add    $0x1,%rax                 # register %rax = %rax + 1
 8.  cmp    %eax,%edi                 # compare %eax to %edi
 9.  vaddsd %xmm1,%xmm0,%xmm0         # Add %xmm1, %xmm0
10.  jg     30 <dgemm+0x30>           # jump if %eax > %edi
11.  add    $0x1,%r11d                # register %r11 = %r11 + 1
12.  vmovsd %xmm0,(%r10)              # Store %xmm0 into C element
```

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the Elaboration in Section 3.7).

Copyright © 2014 Elsevier Inc. All rights reserved.

```
1.  #include <x86intrin.h>
2.  void dgemm (int n, double* A, double* B, double* C)
3.  {
4.    for ( int i = 0; i < n; i+=4 )
5.      for ( int j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( int k = 0; k < n; k++ )
8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                  _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                 _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }
```

FIGURE 3.23 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86. Figure 3.24 shows the assembly language produced by the compiler for the inner loop.

Copyright © 2014 Elsevier Inc. All rights reserved.

```
 1.  vmovapd (%r11),%ymm0             # Load 4 elements of C into %ymm0
 2.  mov    %rbx,%rcx                 # register %rcx = %rbx
 3.  xor    %eax,%eax                 # register %eax = 0
 4.  vbroadcastsd (%rax,%r8,1),%ymm1  # Make 4 copies of B element
 5.  add    $0x8,%rax                 # register %rax = %rax + 8
 6.  vmulpd (%rcx),%ymm1,%ymm1        # Parallel mul %ymm1,4 A elements
 7.  add    %r9,%rcx                  # register %rcx = %rcx + %r9
 8.  cmp    %r10,%rax                 # compare %r10 to %rax
 9.  vaddpd %ymm1,%ymm0,%ymm0         # Parallel add %ymm1, %ymm0
10.  jne    50 <dgemm+0x50>           # jump if not %r10 != %rax
11.  add    $0x1,%esi                 # register % esi = % esi + 1
12.  vmovapd %ymm0,(%r11)             # Store %ymm0 into 4 C elements
```

FIGURE 3.24 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.23. Note the similarities to Figure 3.22, with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

Copyright © 2014 Elsevier Inc. All rights reserved.

FIGURE 3.25 A sampling of newspaper and magazine articles from November 1994, including the *New York Times, San Jose Mercury News, San Francisco Chronicle, and Infoworld.* The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

Copyright © 2014 Elsevier Inc. All rights reserved.

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			

FIGURE 3.26 The MIPS instruction set. This book concentrates on the instructions in the left column. This information is also found in columns 1 and 2 of the MIPS Reference Data Card at the front of this book.

Copyright © 2014 Elsevier Inc. All rights reserved.

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 *uns.*)	madd*s*	R
multiply and subtract (S or *uns.*)	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 uns.)*	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 uns.)*	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

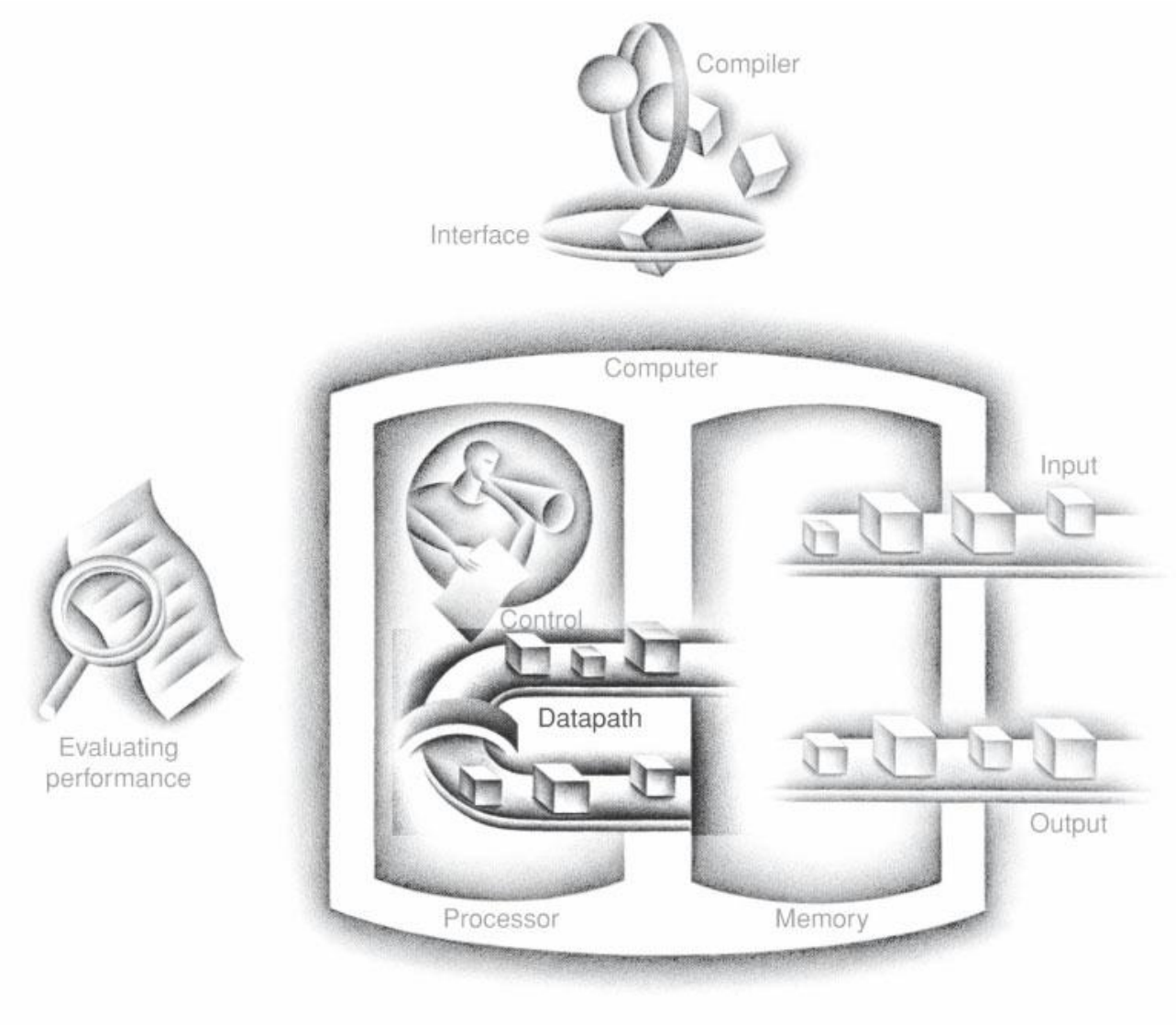
FIGURE 3.27 Remaining MIPS-32 and Pseudo MIPS instruction sets. *f* means single (s) or double (d) precision floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.f/ msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*). The underscore represents the letter to include to represent that datatype.

Copyright © 2014 Elsevier Inc. All rights reserved.

Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	add	0.0%	0.0%	FP add double	add.d	0.0%	10.6%
add immediate	addi	0.0%	0.0%	FP subtract double	sub.d	0.0%	4.9%
add unsigned	addu	5.2%	3.5%	FP multiply double	mul.d	0.0%	15.0%
add immediate unsigned	addiu	9.0%	7.2%	FP divide double	div.d	0.0%	0.2%
subtract unsigned	subu	2.2%	0.6%	FP add single	add.s	0.0%	1.5%
AND	AND	0.2%	0.1%	FP subtract single	sub.s	0.0%	1.8%
AND immediate	ANDi	0.7%	0.2%	FP multiply single	mul.s	0.0%	2.4%
OR	OR	4.0%	1.2%	FP divide single	div.s	0.0%	0.2%
OR immediate	ORi	1.0%	0.2%	load word to FP double	l.d	0.0%	17.5%
NOR	NOR	0.4%	0.2%	store word to FP double	s.d	0.0%	4.9%
shift left logical	sll	4.4%	1.9%	load word to FP single	l.s	0.0%	4.2%
shift right logical	srl	1.1%	0.5%	store word to FP single	s.s	0.0%	1.1%
load upper immediate	lui	3.3%	0.5%	branch on floating-point true	bclt	0.0%	0.2%
load word	lw	18.6%	5.8%	branch on floating-point false	bclf	0.0%	0.2%
store word	sw	7.6%	2.0%	floating-point compare double	c.x.d	0.0%	0.6%
load byte	lbu	3.7%	0.1%	multiply	mul	0.0%	0.2%
store byte	sb	0.6%	0.0%	shift right arithmetic	sra	0.5%	0.3%
branch on equal (zero)	beq	8.6%	2.2%	load half	lhu	1.3%	0.0%
branch on not equal (zero)	bne	8.4%	1.4%	store half	sh	0.1%	0.0%
jump and link	jal	0.7%	0.2%				
jump register	jr	1.1%	0.2%				
set less than	slt	9.9%	2.3%				
set less than immediate	slti	3.1%	0.3%				
set less than unsigned	sltu	3.4%	0.8%				
set less than imm. uns.	sltiu	1.1%	0.1%				

FIGURE 3.28 The frequency of the MIPS instructions for SPEC CPU2006 integer and floating point. All instructions that accounted for at least 0.2% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here.

Copyright © 2014 Elsevier Inc. All rights reserved.



Unn Fig. 1

Copyright © 2014 Elsevier Inc. All rights reserved.

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten}
\end{array}
$$

Unn Fig. 2

Copyright © 2014 Elsevier Inc. All rights reserved.

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

Unn Fig. 3

Copyright © 2014 Elsevier Inc. All rights reserved.

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

Unn Fig. 4

Copyright © 2014 Elsevier Inc. All rights reserved.

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

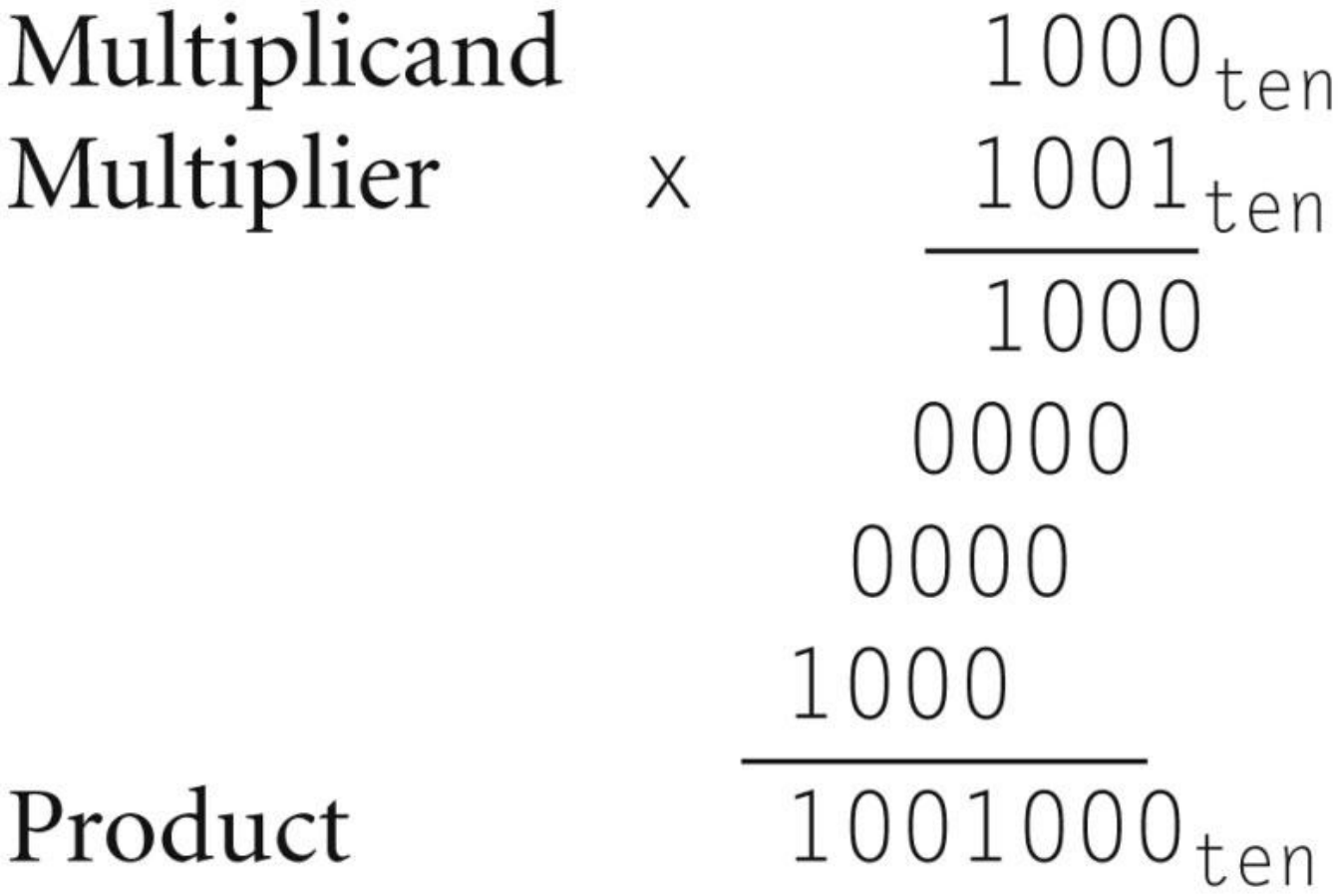
Unn Fig. 5

Copyright © 2014 Elsevier Inc. All rights reserved.

```
addu $t0, $t1, $t2      # $t0 = sum
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2      # (2^32 - $t1 - 1) < $t2
                        # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

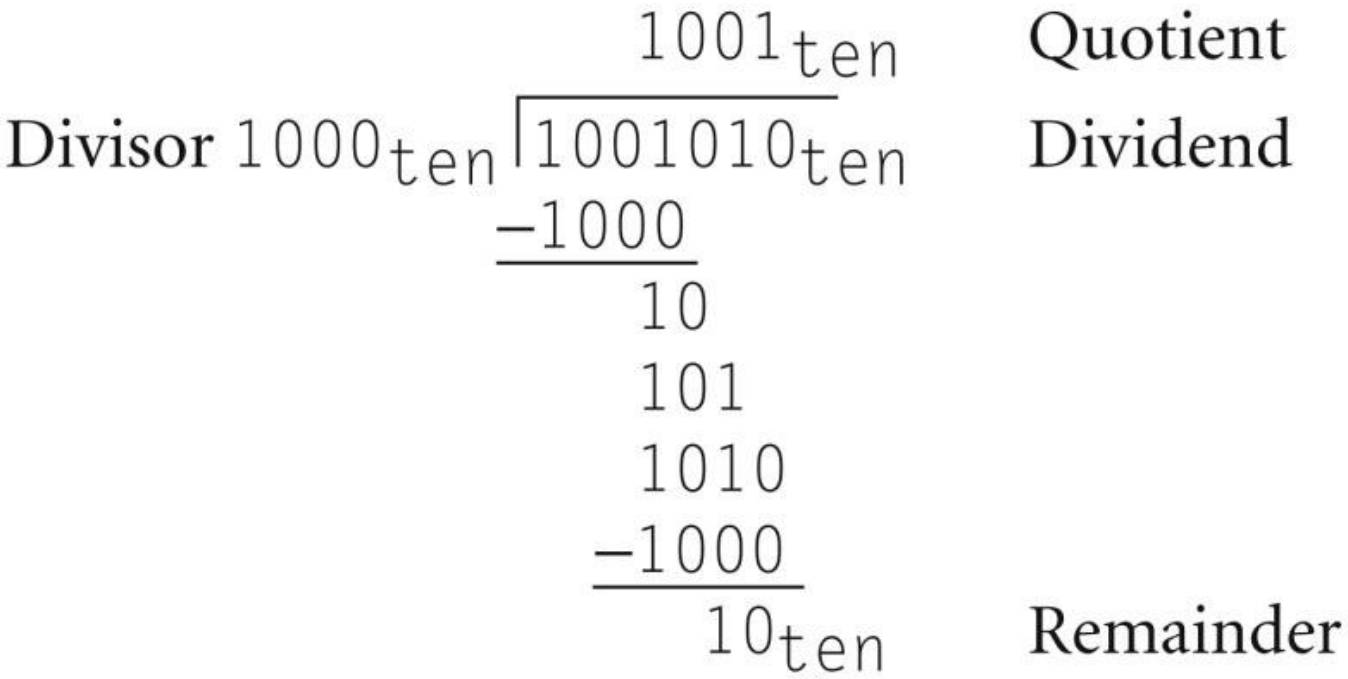
Unn Fig. 6

Copyright © 2014 Elsevier Inc. All rights reserved.



Unn Fig. 7

Copyright © 2014 Elsevier Inc. All rights reserved.

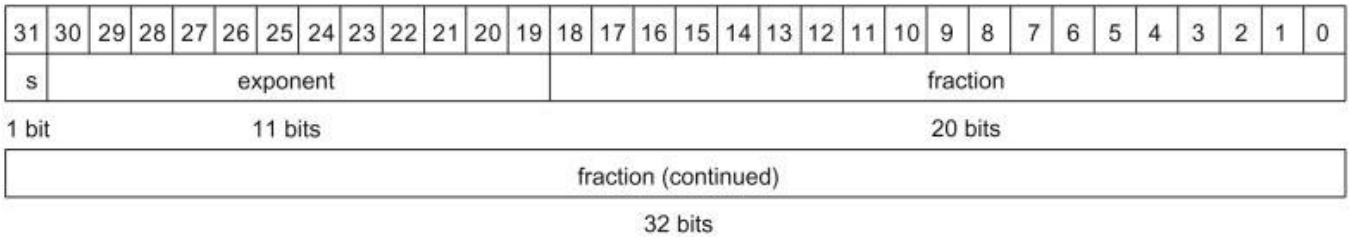


Unn Fig. 8

Copyright © 2014 Elsevier Inc. All rights reserved.

Unn Fig. 9

Copyright © 2014 Elsevier Inc. All rights reserved.



Unn Fig. 10

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

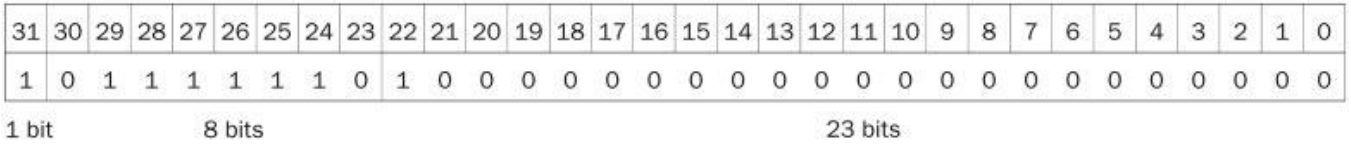
Unn Fig. 11

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

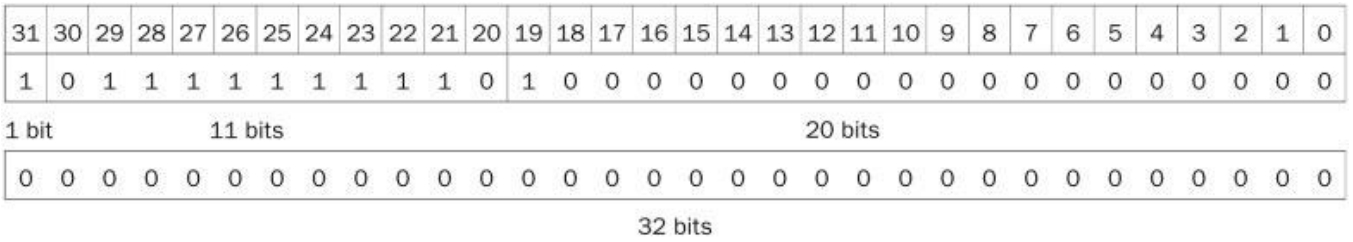
Unn Fig. 12

Copyright © 2014 Elsevier Inc. All rights reserved.



Unn Fig. 13

Copyright © 2014 Elsevier Inc. All rights reserved.



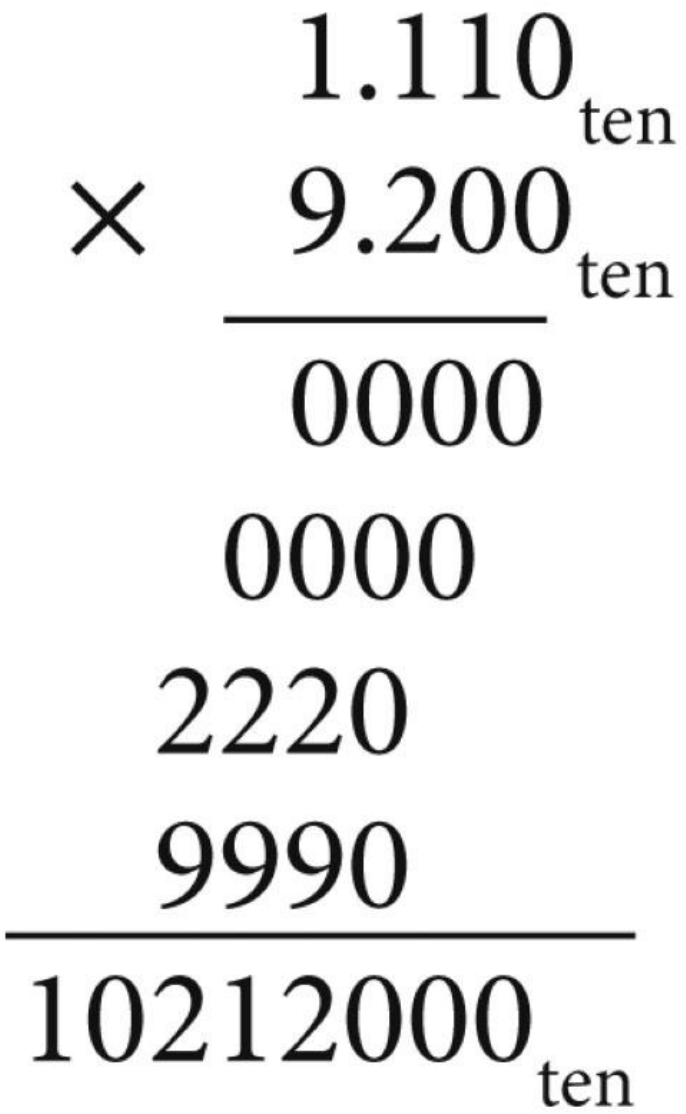
Unn Fig. 14

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Unn Fig. 15

Copyright © 2014 Elsevier Inc. All rights reserved.



Unn Fig. 16

Copyright © 2014 Elsevier Inc. All rights reserved.

$$
\begin{array}{r}
1.000_{\text{two}} \\
\times \quad 1.110_{\text{two}} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{\text{two}}
\end{array}
$$

Unn Fig. 17

Copyright © 2014 Elsevier Inc. All rights reserved.

```
void mm (double c[][], double a[][], double b[][])
{
        int i, j, k;
        for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
        for (k = 0; k != 32; k = k + 1)
          c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

Unn Fig. 18

Copyright © 2014 Elsevier Inc. All rights reserved.

```
    mm:...
       li     $t1, 32  # $t1 = 32 (row size/loop end)
       li     $s0, 0   # i = 0; initialize 1st for loop
L1:    li     $s1, 0   # j = 0; restart 2nd for loop
L2:    li     $s2, 0   # k = 0; restart 3rd for loop
```

Unn Fig. 19

Copyright © 2014 Elsevier Inc. All rights reserved.

```
L3: sll $t0, $s2, 5     # $t0 = k * 2^5 (size of row of b)
    addu $t0, $t0, $s1  # $t0 = k * size(row) + j
    sll $t0, $t0, 3     # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0  # $t0 = byte address of b[k][j]
    l.d $f16, 0($t0)    # $f16 = 8 bytes of b[k][j]
```

Unn Fig. 20

Copyright © 2014 Elsevier Inc. All rights reserved.

```
sll     $t0, $s0, 5     # $t0 = i * 2^5 (size of row of a)
addu    $t0, $t0, $s2   # $t0 = i * size(row) + k
sll     $t0, $t0, 3     # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0   # $t0 = byte address of a[i][k]
l.d     $f18, 0($t0)    # $f18 = 8 bytes of a[i][k]
```

Unn Fig. 21

Copyright © 2014 Elsevier Inc. All rights reserved.

```
addiu   $s1, $s1, 1     # $j = j + 1
bne     $s1, $t1, L2    # if (j != 32) go to L2
addiu   $s0, $s0, 1     # $i = i + 1
bne     $s0, $t1, L1    # if (i != 32) go to L1
…
```

Unn Fig. 22

Copyright © 2014 Elsevier Inc. All rights reserved.

Table 1

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Table 2

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Table 3

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 bit				8 bits					23 bits																						

Table 4

Copyright © 2014 Elsevier Inc. All rights reserved.

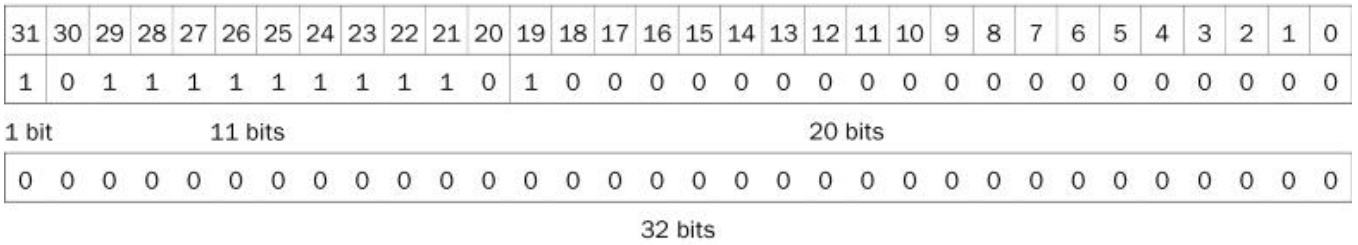


Table 5

Copyright © 2014 Elsevier Inc. All rights reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Table 6

Copyright © 2014 Elsevier Inc. All rights reserved.

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
unsigned int	–	lw, sw, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	–	lb, sb, lui	add, addi, sub, mult, div AND, ANDi, OR, ORi, NOR, slt, slti
–	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Table 7

Copyright © 2014 Elsevier Inc. All rights reserved.

Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

Table 8

Copyright © 2014 Elsevier Inc. All rights reserved.