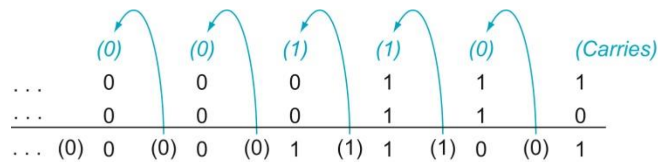


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of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

Copyright © 2021 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 © 2021 Elsevier Inc. All rights reserved.

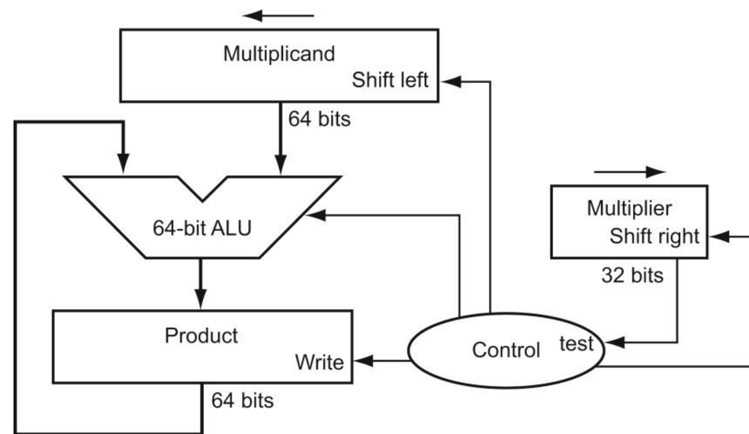


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted 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 © 2021 Elsevier Inc. All rights reserved.

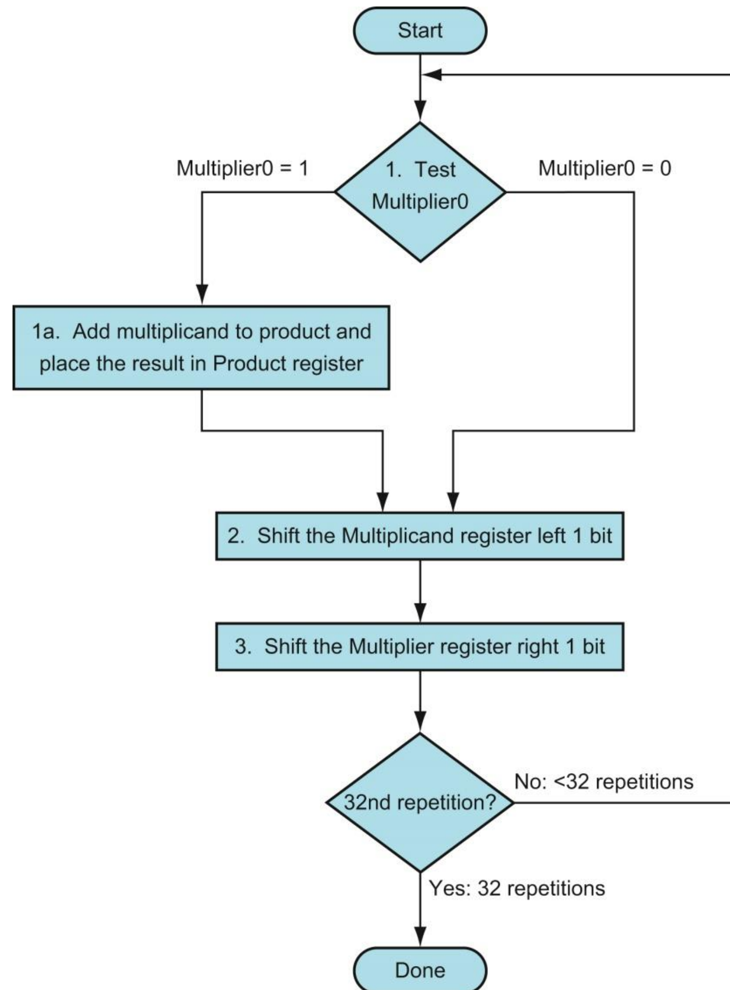


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are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

Copyright © 2021 Elsevier Inc. All rights reserved.

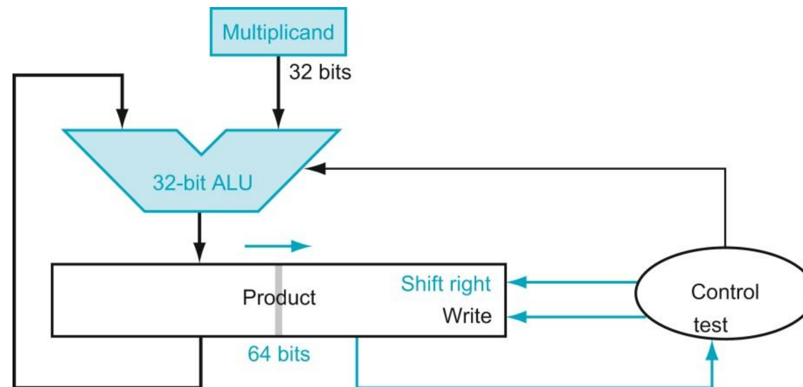


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 32 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 65 bits to hold the carry-out of the adder. These changes are highlighted in color.

Copyright © 2021 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 © 2021 Elsevier Inc. All rights reserved.

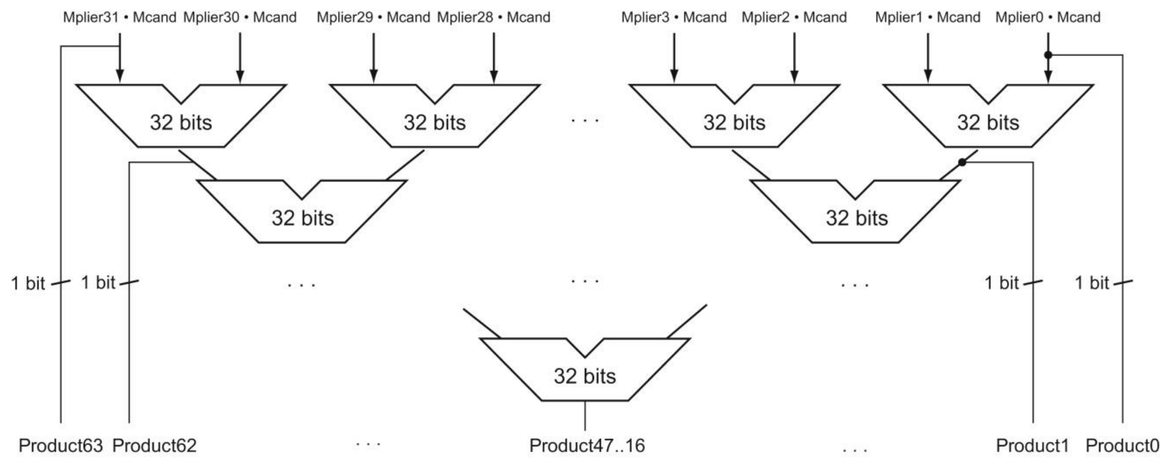


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.

Copyright © 2021 Elsevier Inc. All rights reserved.

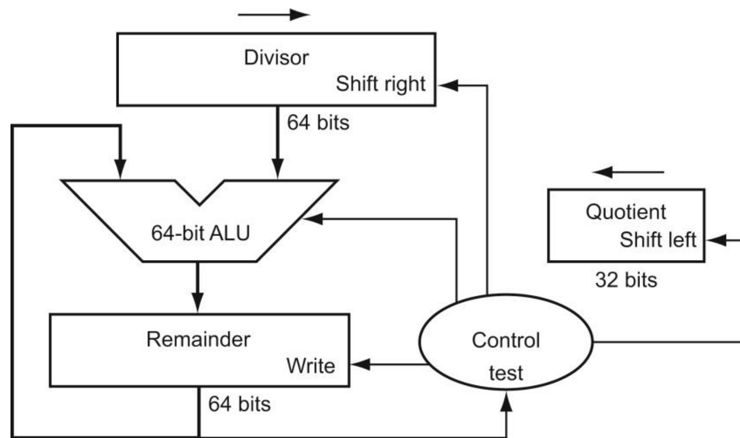


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized 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 © 2021 Elsevier Inc. All rights reserved.

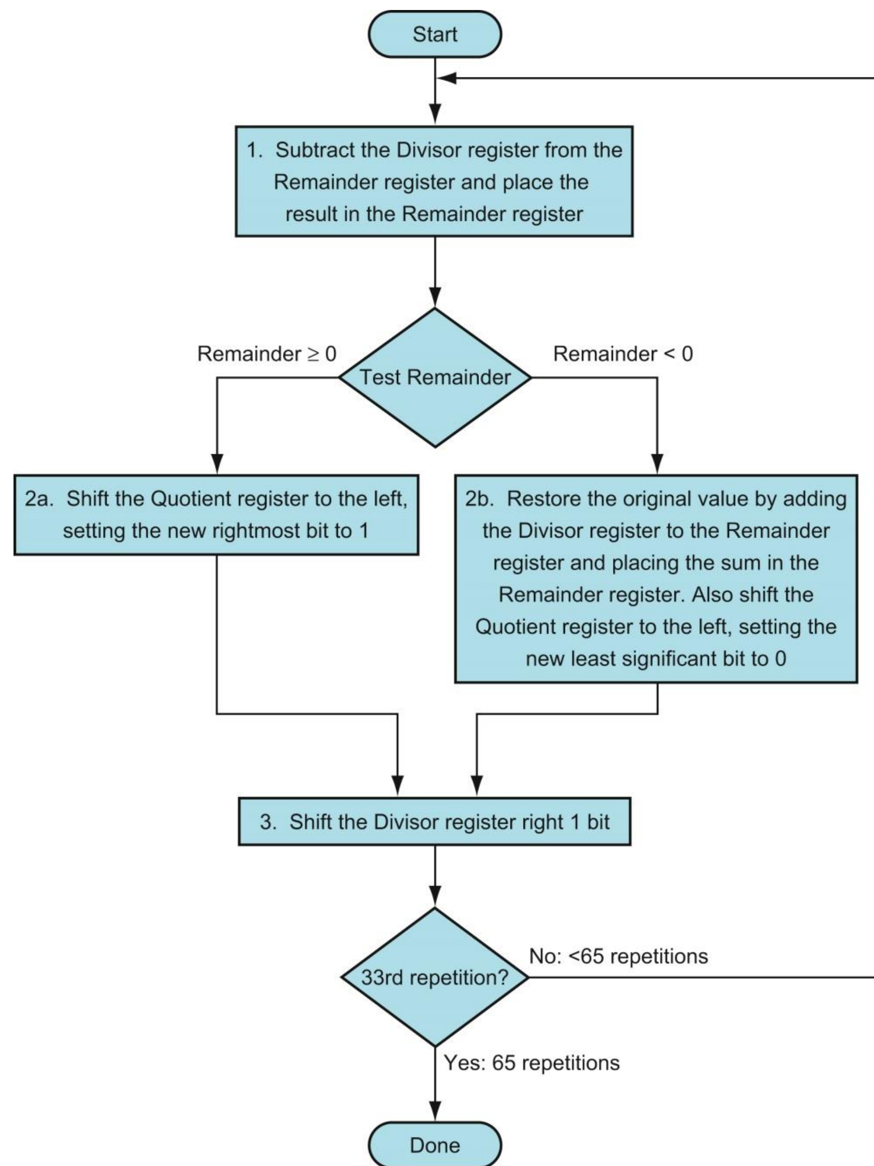


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient 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 © 2021 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	①110 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	①111 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	①111 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	⓪000 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	⓪000 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 © 2021 Elsevier Inc. All rights reserved.

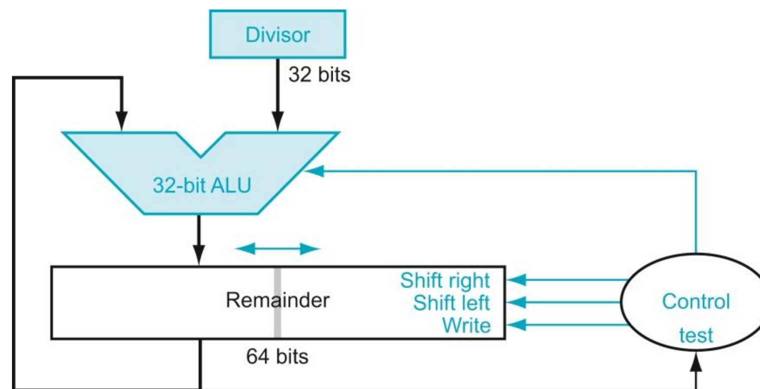


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of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

Copyright © 2021 Elsevier Inc. All rights reserved.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
	Set if less than	slt x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, unsigned	sltu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, immediate	slti x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Set if less than immediate, unsigned	sltiu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Multiply	mul x5, x6, x7	x5 = x6 × x7	Lower 32 bits of 64-bit product
	Multiply high	mulh x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed-unsigned product
	Divide	div x5, x6, x7	x5 = x6 / x7	Divide signed 32-bit numbers
	Divide unsigned	divu x5, x6, x7	x5 = x6 / x7	Divide unsigned 32-bit numbers
	Remainder	rem x5, x6, x7	x5 = x6 % x7	Remainder of signed 32-bit division
	Remainder unsigned	remu x5, x6, x7	x5 = x6 % x7	Remainder of unsigned 32-bit division
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Uns. byte halfword from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	auipc x5, 0x12345	x5 = PC + 0x12345000	Used for PC-relative data addressing
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Uncondit-ional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 3.12 RISC-V core architecture. RISC-V machine language is listed in the RISC-V Reference Data Card at the front of this book.

Copyright © 2021 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 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 233. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

Copyright © 2021 Elsevier Inc. All rights reserved.

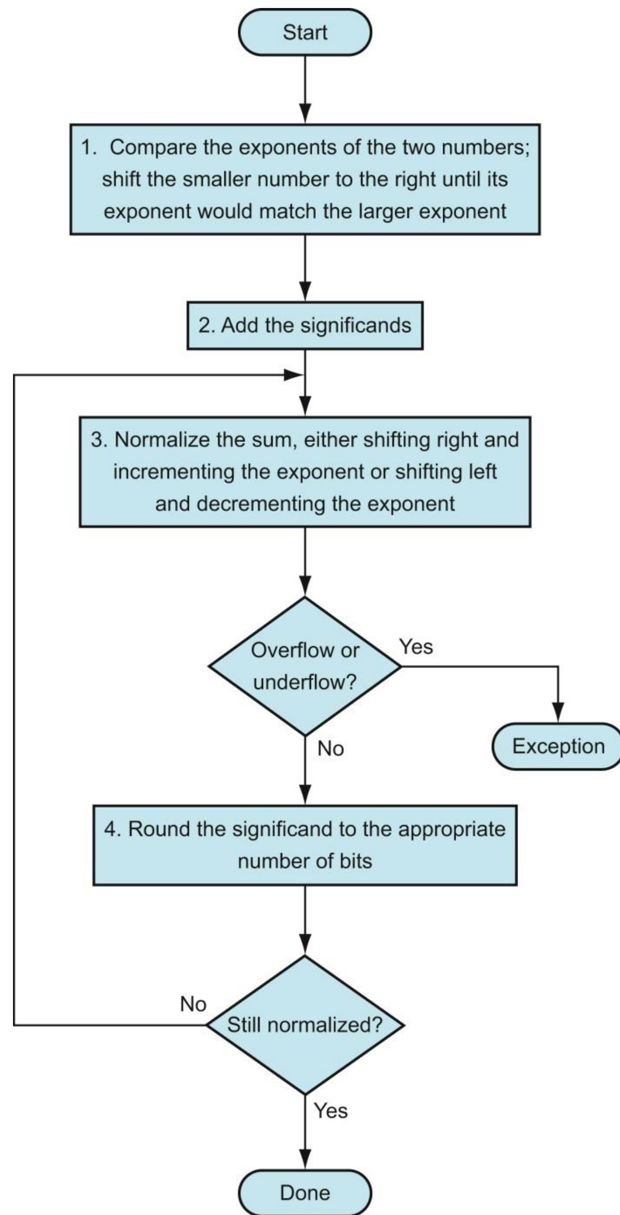


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Copyright © 2021 Elsevier Inc. All rights reserved.

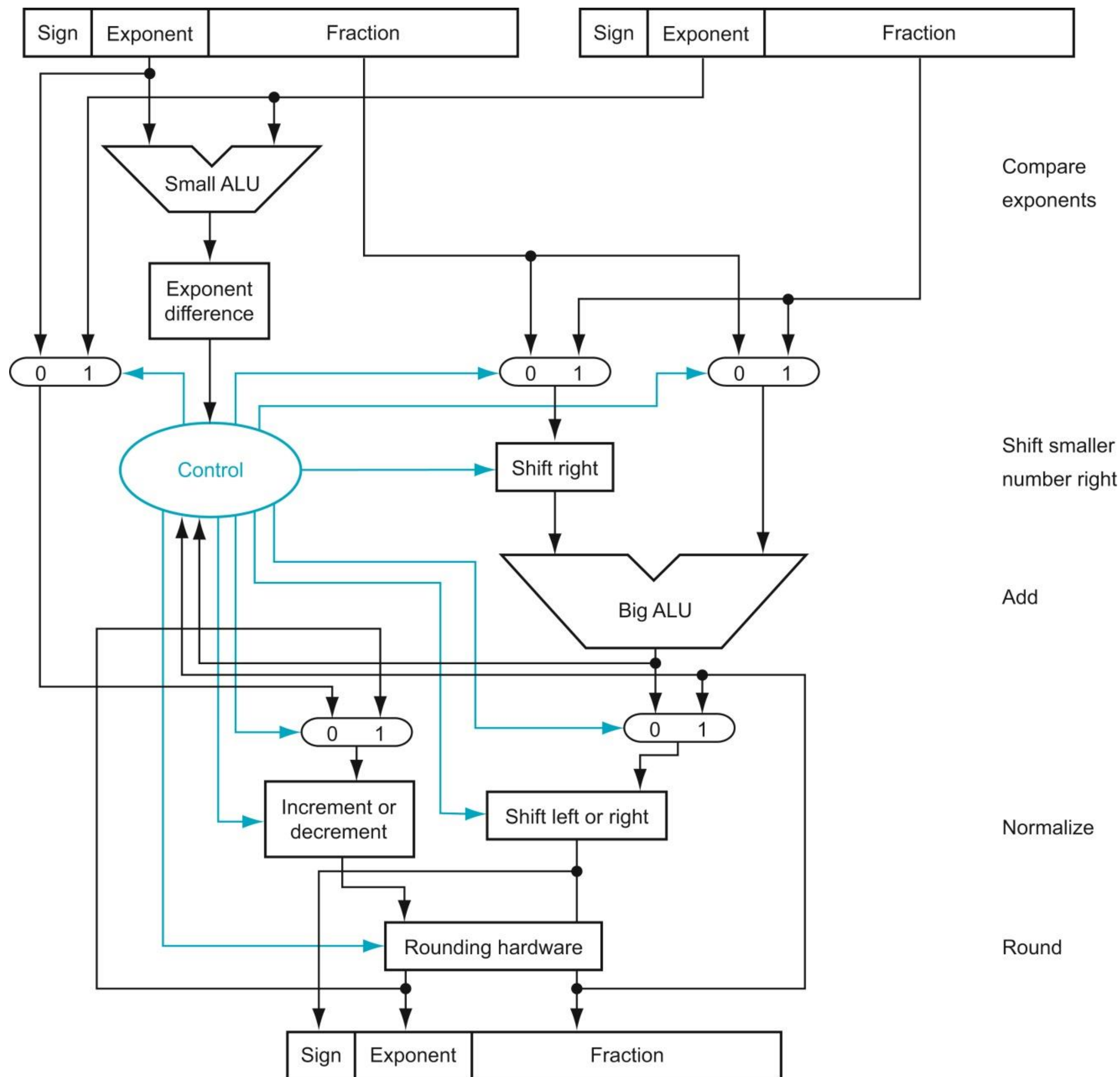


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference 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 © 2021 Elsevier Inc. All rights reserved.

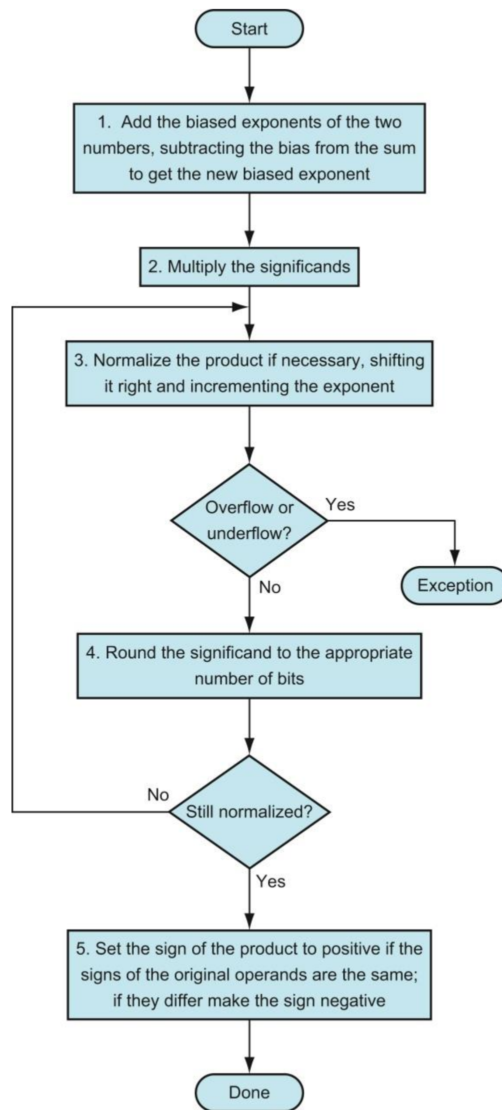


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Copyright © 2021 Elsevier Inc. All rights reserved.

RISC-V floating-point operands

32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

FIGURE 3.17 RISC-V floating-point architecture revealed thus far. This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book.

Copyright © 2021 Elsevier Inc. All rights reserved.

<table>
<tr><th>Data transfer</th><th>Arithmetic</th><th>Compare</th></tr>
<tr><td rowspan="2">MOV[AU]{SS|PS|SD|PD} xmm, {mem|xmm}</td><td>ADD{SS|PS|SD|PD} xmm,{mem|xmm}</td><td>CMP{SS|PS|SD|PD}</td></tr>
<tr><td>SUB{SS|PS|SD|PD} xmm,{mem|xmm}</td><td></td></tr>
<tr><td rowspan="2">MOV[HL]{PS|PD} xmm, {mem|xmm}</td><td>MUL{SS|PS|SD|PD} xmm,{mem|xmm}</td><td></td></tr>
<tr><td>DIV{SS|PS|SD|PD} xmm,{mem|xmm}</td><td></td></tr>
<tr><td></td><td>SQRT{SS|PS|SD|PD} {mem|xmm}</td><td></td></tr>
<tr><td></td><td>MAX{SS|PS|SD|PD} {mem|xmm}</td><td></td></tr>
<tr><td></td><td>MIN{SS|PS|SD|PD} {mem|xmm}</td><td></td></tr>
</table>

FIGURE 3.18 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. The table uses regular expressions to show the variations of instructions. Thus, MOV[AU]{SS|PS|SD|PD} represents the eight instructions MOVASS,MOVAPS,MOVASD,MOVAPD,MOVUSS,MOVUPS,MOVUSD, and MOVUPD. We use square brackets [] to show single-letter alternatives: 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. We use the curly brackets {} with a vertical bar | to show multiple letter 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.

Copyright © 2021 Elsevier Inc. All rights reserved.

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t 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 3.19 Optimized version of DGEMM using C intrinsics to generate AVX512 subword-parallel instructions for the x86. Figure 3.20 shows the assembly language produced by the compiler for the inner loop.

Copyright © 2021 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,%r11                 # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)              # Store %xmm0 into C element
```

FIGURE 3.20 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.19. Note the similarities to Figure 2.44 of Chapter 2, with the primary difference being that the original floating-point operations are now using ZMM registers and the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision, and it is performing a single multiply-add instruction instead of separate multiply and add instruction.

Copyright © 2021 Elsevier Inc. All rights reserved.

A Closer Look

Warning: Intel Inside

BUSINESS

MARKETS ◆ HIGH TECH ◆ ECONOMY

TALKING BUSINESS

PERSONAL TECHNOLOGY

Intel to work on software patch for Pentium bug

Grove says he's sorry about Pentium bug

INFOWORLD

The Voice of Personal Computing in the Enterprise

Bug Dodge Booed

Intel Knocked For Response On Flaw

Chip Shot

Computer Giants' War Over Flaw in Pentium Jolts the PC Industry

Flawed Chip Bruises Intel

Investors react, stock plunges

INTEL'S MISTAKE

BUSINESS

Some Scientists Are Angry Over Flaw In Pentium Chip, and Intel's Response

Intel's Pentium Problem Persists

TECHNOLOGY

Intel's Grove Airs Apology for Pentium Over the Internet

Pentium woes continue

Faulty FPU flubs math in certain equations

Multithreading gets lost on P100 systems

IBM to stop shipping Pentium PCs

FIGURE 3.21 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 opening comedic monologue of the *David Letterman Late Show* on television. ("You know what goes great with those defective Pentium chips? Defective Pentium salsa!") Intel eventually took a $500 million write-off to replace the buggy chips.

Copyright © 2021 Elsevier Inc. All rights reserved.

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load word	lw	12.65%	27.01%
Add registers	add	7.57%	34.58%
Load fl. pt. double	fld	6.83%	41.41%
Store word	sw	5.81%	47.22%
Branch if not equal	bne	4.14%	51.36%
Shift left immediate	slli	3.65%	55.01%
Fused mul-add double	fmadd.d	3.49%	58.50%
Branch if equal	beq	3.27%	61.77%
Add immediate word	addiw	2.86%	64.63%
Store fl. pt. double	fsd	2.24%	66.87%
Multiply fl. pt. double	fmul.d	2.02%	68.89%

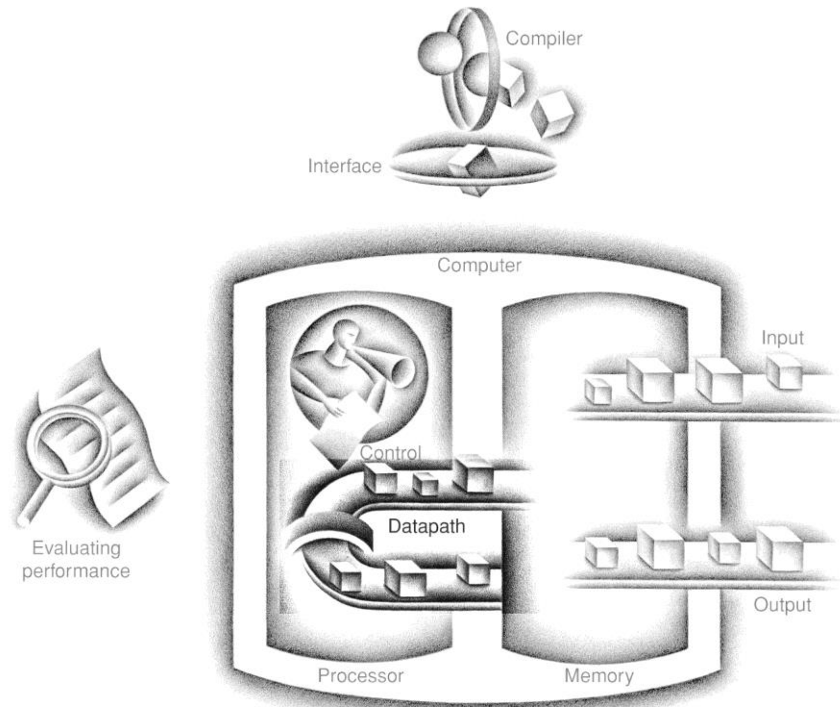
FIGURE 3.22 The frequency of the RISC-V instructions for the SPEC CPU2006 benchmarks. The 17 most popular instructions, which collectively account for 76% of all instructions executed, are included in the table. Pseudoinstructions are converted into RISC-V before execution, and hence do not appear here, explaining in part the popularity of addi.

Copyright © 2021 Elsevier Inc. All rights reserved.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate *(signed or unsigned)*	neg*s*	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left *(unaligned)*	lwl	I	load address	la	rd,addr
load word right *(unaligned)*	lwr	I	load double	ld	rd,addr
store word left *(unaligned)*	swl	I	store double	sd	rd,addr
store word right *(unaligned)*	swr	I	unaligned load word	ulw	rd,addr
load linked (*atomic update*)	ll	I	unaligned store word	usw	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or *uns.*)	madd*s*	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or *uns.*)	msub*s*	I	branch on compare *(signed or unsigned)*	bx*s*	rs,rt,L
branch on ≥ zero and link	bgezal	I	(x = lt, le, gt, ge)		
branch on < zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare *(signed or unsigned)*	sx*s*	rd,rs,rt
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			load to floating point *(s or d)*	l.*f*	rd,addr
branch compare reg likely	bxl	I	store from floating point *(s or d)*	s.*f*	rd,addr
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			

FIGURE 3.23 Floating point format for IEEE 754 single-precision (fp32), IEEE 754 half-precision (fp16), and Brain float 16. Google's TPUv3 hardware uses Brain float 16 (see Section 6.11).

Copyright © 2021 Elsevier Inc. All rights reserved.



Copyright © 2021 Elsevier Inc. All rights reserved.

Step	Action	Multiplier	Multiplicand	Product
0	Initial Vals	001 010	000 000 110 010	000 000 000 000
1	lsb=0, no op	001 010	000 000 110 010	000 000 000 000
	Lshift Mcand	001 010	000 001 100 100	000 000 000 000
	Rshift Mplier	000 101	000 001 100 100	000 000 000 000
2	Prod=Prod+Mcand	000 101	000 001 100 100	000 001 100 100
	Lshift Mcand	000 101	000 011 001 000	000 001 100 100
	Rshift Mplier	000 010	000 011 001 000	000 001 100 100
3	lsb=0, no op	000 010	000 011 001 000	000 001 100 100
	Lshift Mcand	000 010	000 110 010 000	000 001 100 100
	Rshift Mplier	000 001	000 110 010 000	000 001 100 100
4	Prod=Prod+Mcand	000 001	000 110 010 000	000 111 110 100
	Lshift Mcand	000 001	001 100 100 000	000 111 110 100
	Rshift Mplier	000 000	001 100 100 000	000 111 110 100
5	lsb=0, no op	000 000	001 100 100 000	000 111 110 100
	Lshift Mcand	000 000	011 001 000 000	000 111 110 100
	Rshift Mplier	000 000	011 001 000 000	000 111 110 100
6	lsb=0, no op	000 000	110 010 000 000	000 111 110 100
	Lshift Mcand	000 000	110 010 000 000	000 111 110 100
	Rshift Mplier	000 000	110 010 000 000	000 111 110 100

Copyright © 2021 Elsevier Inc. All rights reserved.

Step	Action	Multiplicand	Product/Multiplier
0	Initial Vals	110 010	000 000 001 010
1	lsb=0, no op	110 010	000 000 001 010
	Rshift Product	110 010	000 000 000 101
2	Prod=Prod+Mcand	110 010	110 010 000 101
	Rshift Mplier	110 010	011 001 000 010
3	lsb=0, no op	110 010	011 001 000 010
	Rshift Mplier	110 010	001 100 100 001
4	Prod=Prod+Mcand	110 010	111 110 100 001
	Rshift Mplier	110 010	011 111 010 000
5	lsb=0, no op	110 010	011 111 010 000
	Rshift Mplier	110 010	001 111 101 000
6	lsb=0, no op	110 010	001 111 101 000
	Rshift Mplier	110 010	000 111 110 100

Copyright © 2021 Elsevier Inc. All rights reserved.

Step	Action	Quotient	Divisor	Remainder
0	Initial Vals	000 000	010 001 000 000	000 000 111 100
1	Rem=Rem–Div	000 000	010 001 000 000	101 111 111 100
	Rem<0,R+D,Q<<	000 000	010 001 000 000	000 000 111 100
	Rshift Div	000 000	001 000 100 000	000 000 111 100
2	Rem=Rem–Div	000 000	001 000 100 000	111 000 011 100
	Rem<0,R+D,Q<<	000 000	001 000 100 000	000 000 111 100
	Rshift Div	000 000	000 100 010 000	000 000 111 100
3	Rem=Rem–Div	000 000	000 100 010 000	111 100 101 100
	Rem<0,R+D,Q<<	000 000	000 100 010 000	000 000 111 100
	Rshift Div	000 000	000 010 001 000	000 000 111 100
4	Rem=Rem–Div	000 000	000 010 001 000	111 110 110 100
	Rem<0,R+D,Q<<	000 000	000 010 001 000	000 000 111 100
	Rshift Div	000 000	000 001 000 100	000 000 111 100
5	Rem=Rem–Div	000 000	000 001 000 100	111 111 111 000
	Rem<0,R+D,Q<<	000 000	000 001 000 100	000 000 111 100
	Rshift Div	000 000	000 000 100 010	000 000 111 100
6	Rem=Rem–Div	000 000	000 000 100 010	000 000 011 010
	Rem>0,Q<<1	000 001	000 000 100 010	000 000 011 010
	Rshift Div	000 001	000 000 010 001	000 000 011 010
7	Rem=Rem–Div	000 001	000 000 010 001	000 000 001 001
	Rem>0,Q<<1	000 011	000 000 010 001	000 000 001 001
	Rshift Div	000 011	000 000 001 000	000 000 001 001

Copyright © 2021 Elsevier Inc. All rights reserved.

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	010 001	000 000 111 100
1	R<<	010 001	000 001 111 000
	Rem=Rem–Div	010 001	111 000 111 000
	Rem<0,R+D	010 001	000 001 111 000
2	R<<	010 001	000 011 110 000
	Rem=Rem–Div	010 001	110 010 110 000
	Rem<0,R+D	010 001	000 011 110 000
3	R<<	010 001	000 111 100 000
	Rem=Rem–Div	010 001	110 110 110 000
	Rem<0,R+D	010 001	000 111 100 000
4	R<<	010 001	001 111 000 000
	Rem=Rem–Div	010 001	111 110 000 000
	Rem<0,R+D	010 001	001 111 000 000

Copyright © 2021 Elsevier Inc. All rights reserved.

Step	Action	Divisor	Remainder/Quotient
5	R<<	010 001	011 110 000 000
	Rem=Rem–Div	010 001	111 110 000 000
	Rem>0,R0=1	010 001	001 101 000 001
6	R<<	010 001	011 010 000 010
	Rem=Rem–Div	010 001	001 001 000 010
	Rem>0,R0=1	010 001	001 001 000 011

Copyright © 2021 Elsevier Inc. All rights reserved.

Answer	sign	exp	Exact?
1 01111101 00000000000000000000000	–	−2	Yes

Copyright © 2021 Elsevier Inc. All rights reserved.