Lecture 9
Floating Point

School of Computer Science and Engineering
Soongsil University

3. Arithmetic for Computers

3.1 Introduction
3.2 Addition and Subtraction
3.3 Multiplication
3.4 Division
3.5 Floating Point
3.6 Parallelism and Computer Arithmetic: Subword Parallelism
3.7 Real Stuff: x86 Streaming SIMD Extensions and Advanced Vector Extensions
3.8 Going Faster: Subword Parallelism and Matrix Multiply
3.9 Fallacies and Pitfalls
3.10 Concluding Remarks
3.11 Historical Perspective and Further Reading
3.12 Exercises

3.5 Floating Point

- **Real numbers**

  $3.14159265\ldots_{ten}$ (pi), $2.71828\ldots_{ten}$ (e),

  0.000000001_{ten} , $0.1_{ten} \times 10^{-8}$ or $1.0_{ten} \times 10^{-9}$,

  $3{,}155{,}760{,}000_{ten}$, $0.00315576 \times 10^{12}$ or 3.15576×10^{9}

- **Scientific notation**
  - $a \times 10^{b}$ (a times ten raised to the power of b)
    - where the exponent b is an integer, and the significand (or mantissa) a is any real number
    - $1.0_{ten} \times 10^{-9}$, 315.576×10^{7}
  - A notation that renders numbers with a single digit to the left of the decimal point (in our text)

- **Normalized number**
  - A number in scientific notation that has no leading 0s (i.e. $1 \le |a| < 10$).

Floating-Point Representation

- **Floating point numbers in binary form**

  $\pm 1.xxxxxxxxx_{\text{two}} \times 2^{yyyy}$

- **Sign**
  - 1 bit
- **Exponent**
  - 8 bits (including the sign of the exponent)
- **Fraction (=significand=mantissa)**
  - 23 bits, fraction
  - sign and magnitude representation

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>s</td><td colspan="8">exponent</td><td colspan="23">fraction</td></tr>
<tr><td>1 bit</td><td colspan="8">8 bits</td><td colspan="23">23 bits</td></tr>
</table>

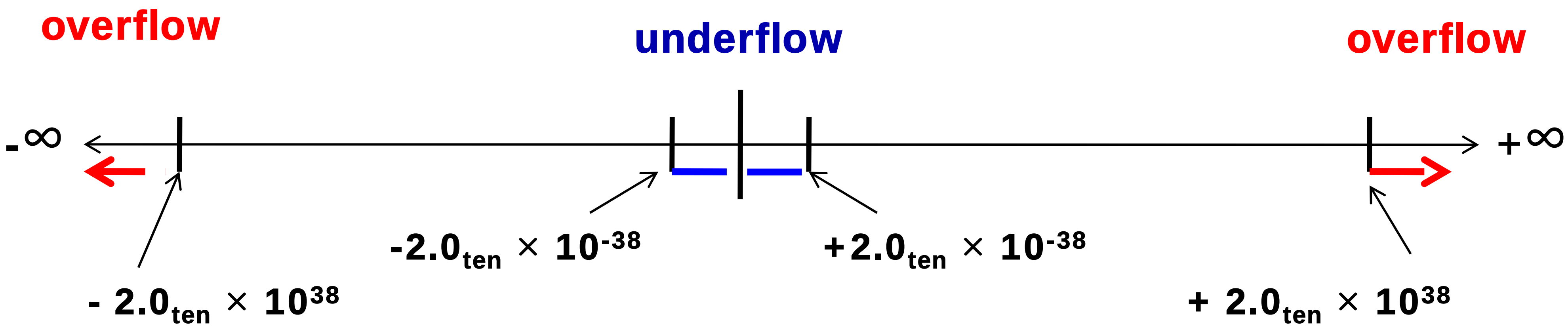
Floating Point Numbers

- **General form of floating-point numbers**

  $(-1)^S \times F \times 2^E$

- **Tradeoff between accuracy and range**
  - Large significand … increased accuracy
  - Large exponent … increased range of numbers
- **Range of floating-point numbers in MIPS**

  $\mathbf{2.0_{ten} \times 10^{-38} \sim 2.0_{ten} \times 10^{38}}$



ANSI / IEEE Std 754-1985

- **IEEE standard for binary floating-point arithmetic**
- **Hidden-bit scheme**

  $(-1)^s \times (1 + \text{fraction}) \times 2^E$

  $= (-1)^s \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$
- **In this book,**
  - significand … represent the 24/53-bit number that is 1 plus the fraction
  - fraction … represent the 23/52-bit number
- **32-bit single format**
  - 1-bit sign, 8-bit exponent, 23-bit fraction
- **64-bit double format**
  - 1-bit sign, 11-bit exponent, 52-bit fraction

Double and Quad Precision

- **Double precision floating-point number**
  - 11 exponent bits
  - 52 fraction bits

$$2.0_{ten} \times 10^{-308} \sim 2.0_{ten} \times 10^{308}$$

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>s</td><td colspan="11">exponent</td><td colspan="20">fraction</td></tr>
<tr><td>1 bit</td><td colspan="11">11 bits</td><td colspan="20">20 bits</td></tr>
<tr><td colspan="32">fraction (continued)</td></tr>
<tr><td colspan="32">32 bits</td></tr>
</table>

- **Quad precision floating-point number**
  - IEEE 754-2008 binary128 standard
  - 15 exponent bits
  - 112 significand bits

IEEE 754 Encoding

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	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

Sorting Floating Point Numbers

- Keep encoding that is somewhat compatible with 2's complement
  - e.g., 0.0 in FP is 0 in two's complement
  - Can compare two FP numbers in the same way as comparing 2's complement integers

- Placing the sign in the most significant bit
- Placing exponent before the significand
- But what with the negative exponents ?

Example: 2's complement exponents

- **$1.0_{two} \times 2^{-1}$**

  0 11111111 00000000000000000000000

- **$1.0_{two} \times 2^{+1}$**

  0 00000001 00000000000000000000000

- **$1.0_{two} \times 2^{-1}$ looks like a bigger one than $1.0_{two} \times 2^{+1}$**

  Undesirable !

Biased Notation

- Can reuse integer comparison hardware
  - If the most negative exponent = 00…000
  - and the most positive exponent = 11...111
- $(-1)^{\text{Sign}} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - } \textbf{Bias})}$
- **Exponent biases in IEEE 754**
  - 127 for single precision

    $-126 \le \text{exponent} \le +127$
  - 1023 for double precision

    $-1022 \le \text{exponent} \le +1023$

Biased Exponent with Bias=127

	How it is interpreted		How it is encoded	
	Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
∞, NaN	For infinities		11111111	255
	127	01111111	11111110	254
	...	...	...	...
	2	00000010	10000001	129
Getting closer to zero	1	00000001	10000000	128
	0	00000000	01111111	127
	-1	11111111	01111110	126
	-2	11111110	01111101	125
	...	...	...	...
	-126	10000010	00000001	1
Zero	For Denorms	10000001	00000000	0

Example: Floating-Point Representation

- **Show the IEEE 754 single and double precision representations of -0.75_{ten} .**

[Answer]

- $-0.75_{ten} = -0.11_{two} = -1.1_{two} \times 2^{-1}$
- Single: exponent = -1 + bias = -1 + 127 = 126

  $(-1)^1 \times (1 + .1000...00) \times 2^{(126-127)}$

  = 1 01111110 10000000000000000000000

  = 1011 1111 0100 0000 0000 0000 0000 0000

  = $BF400000_{hex}$
- Double: exponent = -1 + bias = -1 + 1023 = 1022

  $(-1)^1 \times (1 + .1000...00) \times 2^{(1022-1023)}$

  = 1 01111111110 1000000000000000000000000000000000000000...

  = $BFE8000000000000_{hex}$

Example: Converting Binary to Decimal Floating Point

- **What decimal number is represented by**

  1 10000001 01000000000000000000000 ?

[Answer]

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent - Bias})}$

$= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)}$

$= -1 \times 1.25 \times 2^2$

$= -1.25 \times 4$

$= -5.0$

Floating-Point Addition

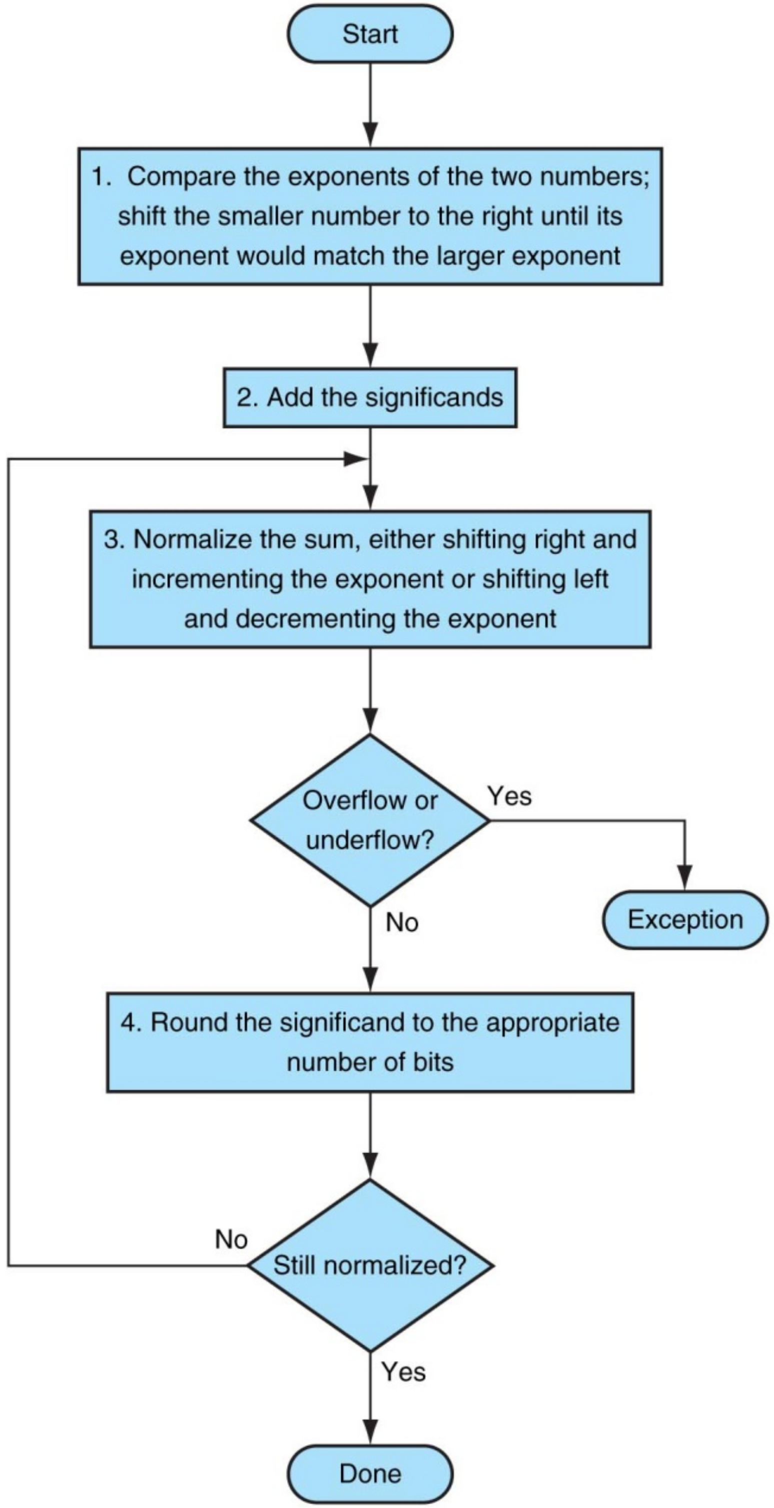


Figure 3.14

Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

- Step 0: Restore the hidden bit in F1 and in F2
- Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming $E1 \geq E2$) keeping track of (three of) the bits shifted out in G R and S
- Step 2: Add the resulting F2 to F1 to form F3
- Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
  - If F1 and F2 have the same sign $\rightarrow$ F3 $\in [1,4)$ $\rightarrow$ 1 bit right shift F3 and increment E3 (check for overflow)
  - If F1 and F2 have different signs $\rightarrow$ F3 may require *many* left shifts each time decrementing E3 (check for underflow)
- Step 4: Round F3 and possibly normalize F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

Example: Floating Point Addition

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - Step 0:
  - Step 1:
  - Step 2:
  - Step 3:
  - Step 4:
  - Step 5:

Example: Floating Point Addition

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

- Step 0: Hidden bits restored in the representation above
- Step 1: Shift significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)
- Step 2: Add significands

  $$1.0000 + (-0.111) = 1.0000 – 0.111 = 0.001$$
- Step 3: Normalize the sum, checking for exponent over/underflow

  $$0.001 \text{ x } 2^{-1} = 0.010 \text{ x } 2^{-2} = .. = 1.000 \text{ x } 2^{-4}$$
- Step 4: The sum is already rounded, so we're done
- Step 5: Rehide the hidden bit before storing

  0 01111011 00000000000000000000000

Floating-Point Adder

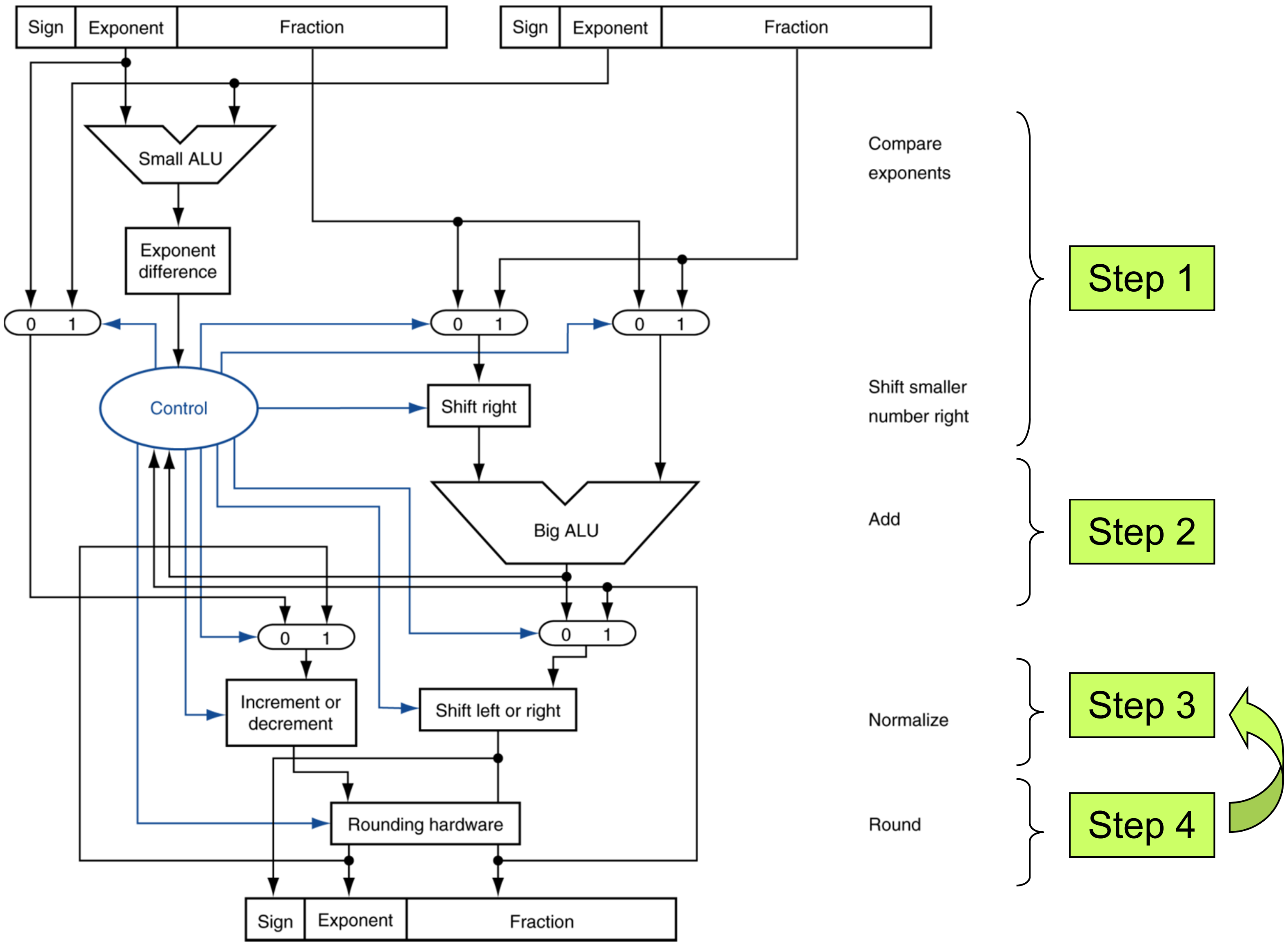


Figure 3.15

Floating-Point Multiplication

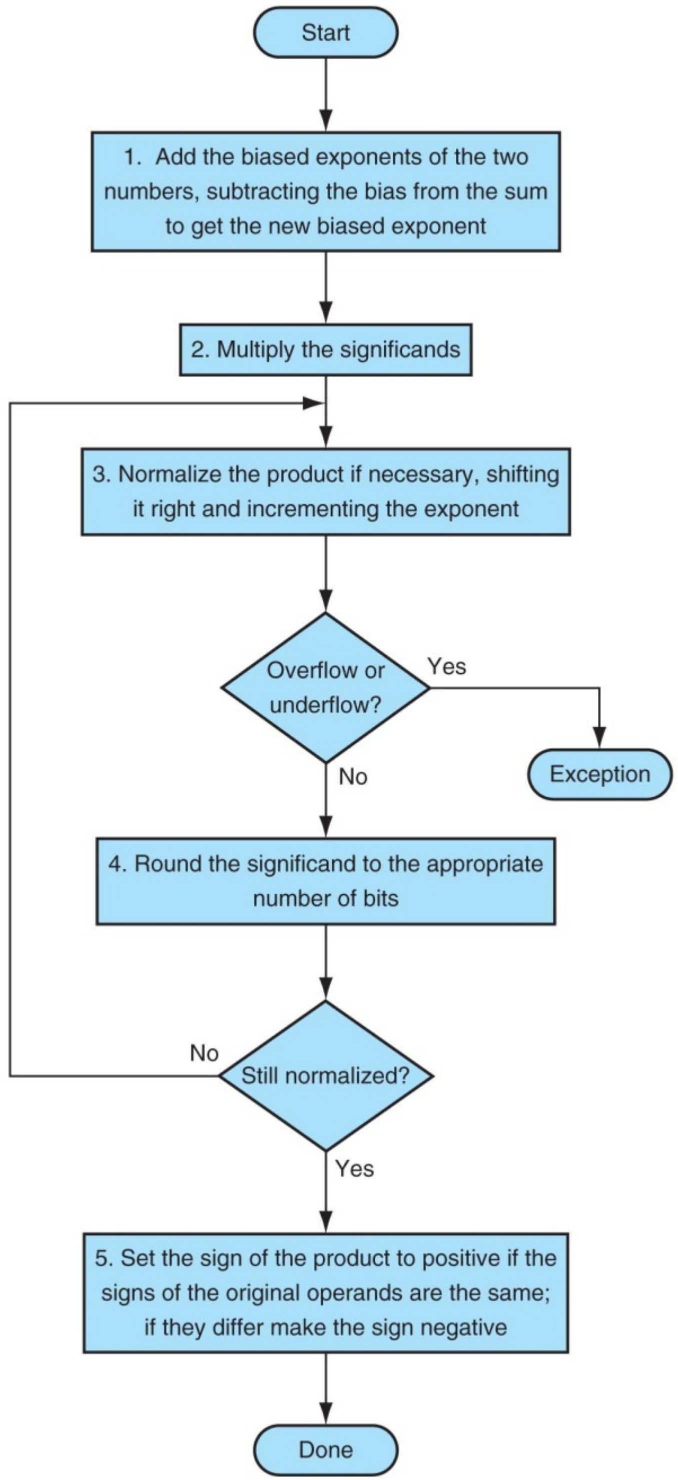


Figure 3.16

Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \text{ x } (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

- Step 0: Restore the hidden bit in F1 and in F2
- Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3

  also determine the sign of the product (which depends on the sign of the operands (most significant bits))
- Step 2: Multiply F1 by F2 to form a double precision F3
- Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
  - Since F1 and F2 come in normalized $\rightarrow$ F3 $\in [1,4)$ $\rightarrow$ 1 bit right shift F3 and increment E3
  - Check for overflow/underflow
- Step 4: Round F3 and possibly normalize F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

Example: Floating Point Multiplication

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \text{ x } (-0.4375 = -1.1100 \times 2^{-2})$$

  - Step 0:
  - Step 1:
  - Step 2:
  - Step 3:
  - Step 4:
  - Step 5:

Example: Floating Point Multiplication

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \text{ x } (-0.4375 = -1.1100 \times 2^{-2})$$

- Step 0: Hidden bits restored in the representation above
- Step 1: Add the exponents (not in bias would be -1 + (-2) = -3 and in bias would be (-1+127) + (-2+127) – 127 = (-1 -2) + (127+127-127) = -3 + 127 = 124
- Step 2: Multiply the significands
  1.0000 x 1.110 = 1.110000
- Step 3: Normalized the product, checking for exp over/underflow
  1.110000 x 2^{-3} is already normalized
- Step 4: The product is already rounded, so we're done
- Step 5: Rehide the hidden bit before storing
  1 01111100 11000000000000000000000

Floating-Point Instructions

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
Condi-tional 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

Figure 3.17

3.10 Concluding Remarks

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	bc1t	0.0%	0.2%
load word	lw	18.6%	5.8%	branch on floating-point false	bc1f	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

Supplement

Accurate Arithmetic

- **Guard bit**
  - Used to provide one fraction bit when shifting left to normalize a result
  - e.g., when normalizing fraction after division or subtraction
- **Round bit**
  - Used to improve rounding accuracy
- **Sticky bit**
  - Used to support Round to nearest even
  - $0.50...00_{ten}$ vs. $0.50...01_{ten}$
  - Is set to a 1 whenever a 1 bit shifts (right) through it
  - e.g., when aligning fraction during addition/subtraction

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

Example: Rounding with Guard Digits

- Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$.
- Significant digit = 3 decimal digits
- **Round to the nearest number with and without guard and round digits.**

[Answer]

1) Without guard and round digits

$$2.34 \times 10^2 + 0.02 \times 10^2 = 2.36 \times 10^2$$

2) With guard and round digits

2.56×10^0 -> 0.0256×10^2 (guard = 5, round = 6)

$$2.3400 \times 10^2 + 0.0256 \times 10^2 = 2.3656 \times 10^2$$

=> rounded to 2.37×10^2

Elaboration

- **Rounding modes in IEEE 754**
  1) Always round up
     - Toward +∞
  2) Always round down (toward -∞)
     - Toward -∞
  3) Truncate
     - Toward 0
     - Round down if positive, up if negative
  4) Round to nearest even
     - When the Guard || Round || Sticky are 100
     - Always creates a 0 in the least significant bit of fraction

Examples

	7.3	7.5	8.5	7.7	-7.3	-7.5	-8.5	-7.7
Round up	8	8	9	8	-7	-7	-8	-7
Round down	7	7	8	7	-8	-8	-9	-8
Truncate	7	7	8	7	-7	-7	-8	-7
Round to nearest even	7	8	8	8	-7	-8	-8	-8

	+0001.01	-0001.01	+0101.10	+0100.10	-0011.10
Round up	+0010	-0001	+0110	+0101	-0011
Round down	+0001	-0010	+0101	+0100	-0100
Truncate	+0001	-0001	+0101	+0100	-0011
Round to nearest even	+0001	-0010	+0110	+0100	-0100