Solutions for Chapter 3 Exercises

3.1 0000 0000 0000 0000 0001 0000 0000 0000_{two}

3.2 1111 1111 1111 1111 1111 1000 0000 0001_{two}

3.3 1111 1111 1110 0001 0111 1011 1000 0000_{two}

3.4 -250_{ten}

3.5 -17_{ten}

3.6 2147483631_{ten}

3.7

```
       addu    $t2, $zero, $t3  # copy $t3 into $t2
       bgez    $t3, next        # if $t3 >= 0 then done
       sub     $t2, $zero, $t3  # negate $t3 and place into $t2
Next:
```

3.9 The problem is that `A_lower` will be sign-extended and then added to `$t0`. The solution is to adjust `A_upper` by adding 1 to it if the most significant bit of `A_lower` is a 1. As an example, consider 6-bit two's complement and the address 23 = 010111. If we split it up, we notice that `A_lower` is 111 and will be sign-extended to 111111 = –1 during the arithmetic calculation. `A_upper_adjusted` = 011000 = 24 (we added 1 to 010 and the lower bits are all 0s). The calculation is then 24 + –1 = 23.

3.10 Either the instruction sequence

```
addu $t2, $t3, $t4
sltu $t2, $t2, $t4
```

or

```
addu $t2, $t3, $t4
sltu $t2, $t2, $t3
```

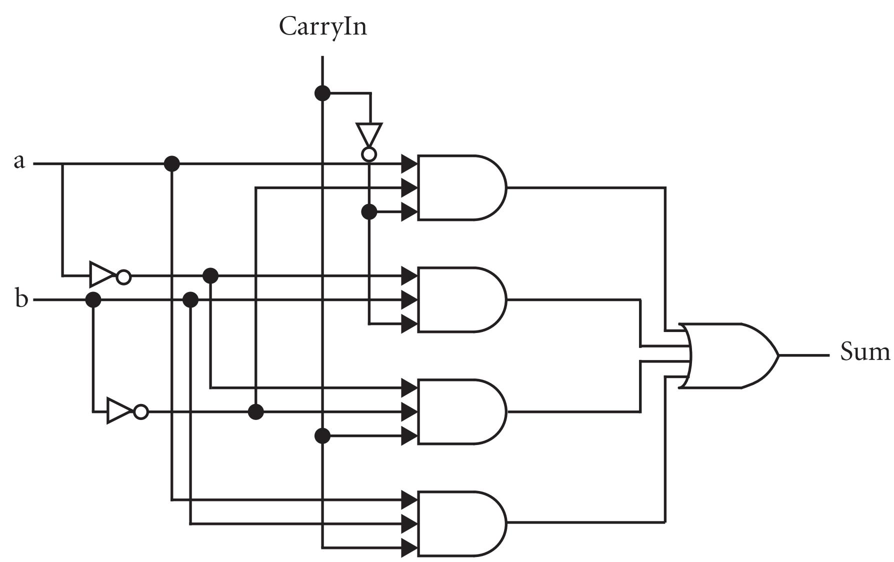
works.

3.12 To detect whether `$s0` < `$s1`, it's tempting to subtract them and look at the sign of the result. This idea is problematic, because if the subtraction results in an overflow, an exception would occur! To overcome this, there are two possible methods: You can subtract them as unsigned numbers (which never produces an exception) and then check to see whether overflow would have occurred. This method is acceptable, but it is lengthy and does more work than necessary. An alternative would be to check signs. Overflow can occur if `$s0` and (`-$s1`) share the same sign; that is, if `$s0` and `$s1` differ in sign. But in that case, we don't need to subtract them since the negative one is obviously the smaller! The solution in pseudocode would be

```
if ($s0<0) and ($s1>0) then
      $t0:=1
else if ($s0>0) and ($s1<0) then
      $t0:=0
else
      $t1:=$s0-$s1
      if ($t1<0) then
            $t0:=1
      else
            $t0:=0
```

3.13 Here is the equation:

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$



3.23

Current bits		Prev. bits	Operation	Reason
ai + 1	ai	ai – 1		
0	0	0	None	Middle of a string of 0s
0	0	1	Add the multiplicand	End of a string of 1s
0	1	0	Add the multiplicand	A string of one 1, so subtract the multiplicand at position i for the beginning of the string and add twice the multiplicand (twice to align with position i + l) for the end of the string; net result, add the multiplicand
0	1	1	Add twice the multiplicand	End of a string of 1s; must align add with 0 in position i + 1
1	0	0	Subtract twice the multiplicand	Beginning of a string of 1s; must subtract with 1 in position i + 1
1	0	1	Subtract the multiplicand	End of string of 1s, so add multiplicand, plus beginning of a string of 1s, so subtract twice the multiplicand; net result is to subtract the multiplicand
1	1	0	Subtract the multiplicand	Beginning of a string of 1s
1	1	1	None	Middle of a string of 1s

One example of 6-bit operands that run faster when Booth's algorithm looks at 3 bits at a time is $21_{ten} \times 27_{ten} = 567_{ten}$.

Two-bit Booth's algorithm:

010101	$= 21_{ten}$
×011011	$= 27_{ten}$
– 010101	10 string (always start with padding 0 to right of LSB)
000000	11 string, middle of a string of 1s, no operation
+ 010101	01 string, add multiplicand
– 010101	10 string, subtract multiplicand
000000	11 string
+ 010101	01 string
11111101011	two's complement with sign extension as needed
0000000000	zero with sign extension shown
000010101	positive multiplicand with sign extension
11101011	
0000000	
+ 010101	
01000110111	$= 567_{ten}$

Don't worry about the carry out of the MSB here; with additional sign extension for the addends, the sum would correctly have an extended positive sign. Now, using the 3-bit Booth's algorithm:

010101	$= 21_{ten}$
×011011	$= 27_{ten}$
– 010101	110 string (always start with padding 0 to right of LSB)
– 010101	101 string, subtract the multiplicand
+ 0101010	011 string, add twice the multiplicand (i.e., shifted left 1 place)
11111101011	two's complement of multiplicand with sign extension
111101011	two's complement of multiplicand with sign extension
+ 0101010	
01000110111	$= 567_{ten}$

Using the 3-bit version gives only 3 addends to sum to get the product versus 6 addends using the 2-bit algorithm.

Booth's algorithm can be extended to look at any number of bits *b* at a time. The amounts to add or subtract include all multiples of the multiplicand from 0 to $2^{(b-1)}$. Thus, for $b > 3$ this means adding or subtracting values that are other than powers of 2 multiples of the multiplicand. These values do not have a trivial "shift left by the power of 2 number of bit positions" method of computation.

3.25

```
l.d    $f0, -8($gp)
l.d    $f2, -16($gp)
l.d    $f4, -24($gp)
fmadd  $f0, $f0, $f2, $f4
s.d    $f0, -8($gp)
```

3.26 a.

x = 0100 0000 0110 0000 0000 0000 0010 0001

y = 0100 0000 1010 0000 0000 0000 0000 0000

Exponents

```
  100 0000 0
 +100 0000 1
 -----------
 1000 0000 1
 –011 1111 1
 -----------
  100 0001 0
```

```
x                          1.100 0000 0000 0000 0010 0001
y                         ×1.010 0000 0000 0000 0000 0000
                          -------------------------------
1 100 0000 0000 0000 0010 0001 000 0000 0000 0000 0000 0000
+  11 0000 0000 0000 0000 1000 010 0000 0000 0000 0000 0000
-----------------------------------------------------------
1.111 0000 0000 0000 0010 1001 010 0000 0000 0000 0000 0000
```

Round result for part b.

```
1.111 1100 0000 0000 0010 1001

z 0011 1100 1110 0000 0000 1010 1100 0000
```

Exponents

```
  100 0001 0
-  11 1100 1
------------
       100 1  --> shift 9 bits

 1.111 0000 0000 0000 0010 1001 010 0000 00
+   z          111 0000 0000 0101 011 0000 00
---------------------------------------------
1.111  0000 0111 0000 0010 1110 101
                                GRS
```

Result:

```
0100 0001 0111 0000 0111 0000 0100 1111
```

b.

```
1.111 1100 0000 0000 0000 1001  result from mult.
+  z          111 0000 0000 0101 011
----------------------------------------------
1.111 1100 0111 0000 0001 1110 011
                               GRS

0100 0001 0111 0000 0111 0000 0100 1110
```

3.27

a.

```
                                                         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  0 0 0 0  0 1 0 1  1 0 1 1
+ 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  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 0 0  0 1 1 0  1 0 0 0
```

b.

```
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 1  1 0 1 1
- 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  1 1 0 1
                                                             1   1
---------------------------------------------------------------------------
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 0  1 1 1 0
```

c.

```
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 1  1 0 1 1
x 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  1 1 0 1
---------------------------------------------------------------------------
                                                        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  0 0 0 0  0 1 0 1  1 0 1 1
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 1  0 1 1 0  1 1
+ 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  1
---------------------------------------------------------------------------
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 0  1 0 0 1  1 1 1 1
```

d.

```
                      0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 1 1two
                  -----------------------------------------------------------------------------
0 0 0 0  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 0 0  0 1 0 1  1 0 1 1
                 - 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 | |  0 1 | |
                 ----------------------------------------------------------------------- | |
                                                                            0 1 0  0 1 1 |
                 - 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 1  1 0 1 |
                 ------------------------------------------------------------------------- |
                                                                                0  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 0 0  0 0 0 0  1 1 0 1
                 ---------------------------------------------------------------------------
                                                                                0 0 0 0
```

3.28

a.

```
                                                 1 1 1  1   1    1 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
+ 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 1 1 0  0 0 1 0  1 0 1 0
```

b.

```
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
- 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
                                                     1  1 1 1 1  1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 0 0  0 1 1 1  1 1 0 0
```

c.

```
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
× 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
  ---------------------------------------------------------------------
  6 6 6 6  6 6 6 6  6 5 5 5  4 4 4 4  3 4 3 3  4 3 3 2  1 1 1 1  1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  0 1 1 0  0 1 1 0  1 0 1 0  0 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 0  1 1 0 0  1 1 0 1  0 1 0 0  1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
  1 1 1 1  1 1 1 1  1 1 1 0  1 1 0 0  1 1 0 1  0 1 0 0  1 1
  1 1 1 1  1 1 1 1  1 1 0 1  1 0 0 1  1 0 1 0  1 0 0 1  1
+ 1 1 1 1  1 1 1 1  0 1 1 0  0 1 1 0  1 0 1 0  0 1 1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  0 0 1 0  0 1 1 0  0 1 0 0  0 0 0 0  1 0 1 1  0 1 0 1
```

d. Convert to positive:

```
                                                                  1  1 0 1 1
                            ------------------------------------------------
0 0 1 0  1 1 0 1  0 1 1 1 | 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 0  1 1 0 0  1 0 1 0  1 1 0 1
                                                       0 0 1 0  1 1 0 1  0 1 1 1   |   |
                                            -  ----------------------------------  |   |
                                                          1  1 1 1 1  0 0 1 1  1 |  |
                                                          1  0 1 1 0  1 0 1 1  1 |  |
                                            -  ----------------------------------  |   |
                                                            1 0 0 0  1 0 0 0  0 1 0 |
                                                              1 0 1  1 0 1 0  1 1 1 |
                                            -  --------------------------------------- |
                                                                1 0  1 1 0 1  0 1 1 1
                            -                                   1 0  1 1 0 1  0 1 1 1
                                               -  ------------------------------------
                                                                0 0  0 0 0 0  0 0 0 0
```

Since signs differ, convert quotient to negative:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0101_{two}$

3.29 Start

Set subtract bit to true

1. If subtract bit true: Subtract the Divisor register from the Remainder and place the result in the remainder register.

   else Add the Divisor register to the Remainder and place the result in the remainder register.

   Test Remainder

   >=0

2. a. Shift the Quotient register to the left, setting rightmost bit to 1.

   <0

2. b. Set subtract bit to false.
3. Shift the Divisor register right 1 bit.

   <33rd rep ---> repeat
   Test remainder
   <0

Add Divisor register to remainder and place in Remainder register.

Done

Example:

Perform $n + 1$ iterations for n bits

Remainder 0000 1011

Divisor 0011 0000

Iteration 1:

(subtract)

Rem 1101 1011

Quotient 0

Divisor 0001 1000

Iteration 2:

(add)

Rem 1111 0011

Q 00

Divisor 0000 1100

Iteration 3:

(add)

Rem 1111 1111

Q 000

Divisor 0000 0110

Iteration 4:

(add)

Rem 0000 0101

Q 0001

Divisor 0000 0011

Iteration 5:

(subtract)

Rem 0000 0010

Q 0001 1

Divisor 0000 0001

Since remainder is positive, done.

Q = 0011 and Rem = 0010

3.30

a. –1 391 460 350

b. 2 903 506 946

c. -8.18545×10^{-12}

d. `sw $s0, $t0(16)   sw $r16, $r8(2)`

3.31

a. 613 566 756

b. 613 566 756

c. 6.34413×10^{-17}

d. `addiu, $s2, $a0, 18724  addiu $18, $4, 0x8924`

3.35

$$\begin{array}{r} .285 \times 10^4 \\ +9.84 \times 10^4 \\ \hline 10.125 \times 10^4 \end{array}$$

1.0125×10^4

with guard and round: 1.01×10^5

without: 1.01×10^5

3.36

$$\begin{array}{r} 3.63 \times 10^4 \\ +.687 \times 10^4 \\ \hline 4.317 \times 10^4 \end{array}$$

with guard and round: 4.32×10^4

without: 4.31×10^4

3.37

$$20_{ten} = 10100_{two} = 1.0100_{two} \cdot 2^{-4}$$

$$\begin{aligned} \text{Sign} &= 0, \text{Significand} = .01\overline{0} \\ \text{Single exponent} &= 4 + 127 = 131 \\ \text{Double exponent} &= 4 + 1023 = 1027 \end{aligned}$$

Single precision	0 1000 011 010 0000 0000 0000 0000 0000
Double precision	0 1000 0000 011 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3.38

Single precision	0 1000 0011 010 0100 0000 0000 0000 0000
Double precision	0 1000 0000 011 0100 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3.39

$$0.1_{ten} = 0.0\overline{0011}_{two} = 1.1\overline{0011}_{two} \cdot 2^{-4}$$

$$\begin{aligned} \text{Sign} &= 0, \text{Significand} = .1\overline{0011} \\ \text{Single exponent} &= -4 + 127 = 123 \\ \text{Double exponent} &= -4 + 1023 = 1019 \end{aligned}$$

	1	8	23	
Single precision	0	01111011	10011001100110011001100 10011001100110011001101	trunc round

	1	11	52	
Double precision	0	01111111011	1001100110011001100110011001100110011001100110011001 1001100110011001100110011001100110011001100110011010	trunc round

3.40

Single precision	1 0001 1110 101 0101 0101 0101 0101 0101
Double precision	1 0111 1111 110 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 trunc 1 0111 1111 110 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1011 round

3.41 No, since floating point adds are not associative, doing them simultaneously is not the same as doing them serially.

3.42

a.

Convert $+1.1011 * 2^{14} + -1.11 * 2^{-2}$

```
 1.1011 0000 0000 0000 0000 000
–0.0000 0000 0000 0001 1100 000
-------------------------------
 1.1010 1111 1111 1110 0100 000
```

0100 0110 1101 0111 1111 1111 0010 0000

b. Calculate new exponent:

```
  111 11 1
  100 0110 1
 +011 1110 1
 -----------
 1000 0101 0
 –011 1111 1   minus bias
 1111 1111
 -----------
  100 0101 1   new exponent
```

Multiply significands:

```
                          1.101 1000 0000 0000 0000 0000
                         ×1.110 0000 0000 0000 0000 0000
                         -------------------------------
  1 11 11
       1 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
      11 0110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 +1.10    1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 ----------------------------------------------------------------
 10.11    1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Normalize and round:

exponent 100 0110 0
significand
1.011 1010 0000 0000 0000 0000

Signs differ, so result is negative:

1100 0110 0011 1010 0000 0000 0000 0000

3.43

0 101 1111 1 011 1110 0100 0000 0000 0000
0 101 0001 1 111 1000 0000 0000 0000 0000

a. Determine difference in exponents:

```
 1011 1111
–1010 0011
----------
  001 1100 --> 28
```

Add significands after scaling:

```
 1.011 1110 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
+0.000 0000 0000 0000 0000 0000 0000 1111 1000 0000 0000 0000 0000
-------------------------------------------------------------------
 1.011 1110 0100 0000 0000 0000 0000 1111 1000 0000 0000 0000 0000
```

Round (truncate) and repack:

0 101 1111 1 011 1110 0100 0000 0000 0000
0101 1111 1011 1110 0100 0000 0000 0000

b. Trivially results in zero:

0000 0000 0000 0000 0000 0000 0000 0000

c. We are computing $(x + y) + z$, where $z = -x$ and $y \neq 0$
$(x + y) + -x = y$ intuitively
$(x + y) + -x = 0$ with finite floating-point accuracy

3.44

a. $2^{15} - 1 = 32767$

b.

$2.0_{ten} \times 2^{2^{15}}$

$2^{2^{11}} = 3.23 \times 10^{616}$

$2^{2^{12}} = 1.04 \times 10^{1233}$

$2^{2^{13}} = 1.09 \times 10^{2466}$

$2^{2^{14}} = 1.19 \times 10^{4932}$

$2^{2^{15}} = 1.42 \times 10^{9864}$

so

as small as $2.0_{ten} \times 10^{-9864}$
and almost as large as $2.0_{ten} \times 10^{9864}$

c. 20% more significant digits, and 9556 orders of magnitude more flexibility. (Exponent is 32 times larger.)

3.45 The implied 1 is counted as one of the significand bits. So, 1 sign bit, 16 exponent bits, and 63 fraction bits.

3.46

Load 2×10^{308}
Square it 4×10^{616}
Square it 1.6×10^{1233}
Square it 2.5×10^{2466}
Square it 6.2×10^{4932}
Square it 3.6×10^{9865}

Min 6 instructions to utilize the full exponent range.