16.482 / 16.561: Computer Architecture and Design

Examples of Booth's Algorithm and Floating-Point Arithmetic

Booth's Algorithm examples:

a. 4 x 5

Initially, **product/multiplier = 0 00000101 0**
Multiplicand = 0 0100, -Multiplicand = 1 1100

Step 1: Lowest two bits of product/multiplier = 10 → add –Mcand into left half, then shift right
→ **product/multiplier = 1 11100010 1**
Step 2: Lowest two bits = 01 → add Mcand into left half, then shift right
→ **product/multiplier = 0 00010001 0**
Step 3: Lowest two bits = 10 → add –Mcand into left half, then shift right
→ **product/multiplier = 1 11101000 1**
Step 4: Lowest two bits = 01 → add Mcand into left half, then shift right
→ **<u>product</u>/multiplier = 0 <u>00010100</u> 0**

b. (-2) x 7

Initially, **product/multiplier = 0 00000111 0**
Multiplicand = 1 1110, -Multiplicand = 0 0010

Step 1: Lowest two bits of product/multiplier = 10 → add –Mcand into left half, then shift right
→ **product/multiplier = 0 00010011 1**
Step 2: Lowest two bits = 11 → shift right
→ **product/multiplier = 0 00001001 1**
Step 3: Lowest two bits = 11 → shift right
→ **product/multiplier = 0 00000100 1**
Step 4: Lowest two bits = 01 → add Mcand into left half, then shift right
→ **<u>product</u>/multiplier = 1 <u>11110010</u> 0**

c. (-8) x (-3)

Initially, **product/multiplier = 0 00001101 0**
Multiplicand = 1 1000, -Multiplicand = 0 1000

Step 1: Lowest two bits of product/multiplier = 10 → add –Mcand into left half, then shift right
→ **product/multiplier = 0 01000110 1**
Step 2: Lowest two bits = 01 → add Mcand into left half, then shift right
→ **product/multiplier = 1 11100011 0**
Step 3: Lowest two bits = 10 → add –Mcand into left half, then shift right
→ **product/multiplier = 0 00110001 1**
Step 4: Lowest two bits = 11 → shift right
→ **product/multiplier = 0 00011000 1**

Decimal → IEEE floating-point conversion:

a. 4.125

Solution: We first need to convert this value to binary and then normalize it:

$4.125 = 100.001_2 = 1.00001 \times 2^2$

We can directly determine each of the fields in our single-precision floating-point value:

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = 2 + 127 = 129 = 10000001_2
Fraction = 00001_2 = $000\ 0100\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

$4.25 = 0100\ 0000\ 1000\ 0100\ 0000\ 0000\ 0000\ 0000_2$ = **0x40840000**

b. -75

Solution: With a negative value, when doing our conversion to binary, we work strictly with the magnitude—floating-point values don't use 2's complement form, so the sign and magnitude are stored separately.

$75 = 1001011_2 = 1.001011 \times 2^6$

Now, determine each of the fields in our single-precision floating-point value:

Sign = 1 *(negative value)*
Exponent = [actual exponent] + bias = 6 + 127 = 133 = 10000101_2
Fraction = 001011_2 = $001\ 0110\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

$-75 = 1100\ 0010\ 1001\ 0110\ 0000\ 0000\ 0000\ 0000_2$ = **0xC2960000**

c. 0.34375

Solution: Although it may be a little difficult to see at first, this value is a sum of powers of 2: 0.34375 = 1/4 + 1/16 + 1/32 = 0.25 + 0.0625 + 0.03125. Therefore:

$$0.34375 = 0.01011_2 = 1.011 \times 2^{-2}$$

Now, determine each of the fields in our single-precision floating-point value:

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = -2 + 127 = 125 = 01111101_2
Fraction = 011_2 = $011\ 0000\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

0.65625 = 0011 1110 1 011 0000 0000 0000 0000 0000 = **0x3EB00000**

d. -141.75

Solution: We again start by converting this value to binary. Note that the fractional part is 1/2 + 1/4 = 0.5 + 0.25:

$$141.75 = 10001101.11_2 = 1.000110111 \times 2^{7}$$

Now, determine each of the fields in our single-precision floating-point value:

Sign = 1 *(negative value)*
Exponent = [actual exponent] + bias = 7 + 127 = 134 = 10000110_2
Fraction = 000110111_2 = $000\ 1101\ 1100\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

-141.75 = 1100 0011 0000 1101 1100 0000 0000 0000 = **0xC30DC000**

e. 16.561 (determine the closest approximation you can)

Solution: While the whole part of this value is a power of 2 ($16 = 2^4$), the fractional part can't be exactly represented. The closest approximation we get with a 23-bit fraction is:

$$16.561 \approx 10000.100011111001110111 0_2 = 1.00001000111110011101110 \times 2^4$$

Now, determine each of the fields in our single-precision floating-point value:

Sign = 0 *(negative value)*
Exponent = [actual exponent] + bias = 4 + 127 = 131 = 10000011_2
Fraction = $000\ 0100\ 0111\ 1100\ 1110\ 1110_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

16.561 = 1100 0001 1000 0100 0111 1100 1110 1110 = **0xC1847CEE**

IEEE floating-point → decimal conversion

a. 0x43020000

Solution: In all cases, we break the value given into the three fields of a single-precision floating-point value: sign (1 bit), biased exponent (8 bits), and fraction (23 bits):

0x43020000 = $0100\ 0011\ 0000\ 0010\ 0000\ 0000\ 0000\ 0000_2$

Sign = 0 *(positive value)*
Biased exponent = 10000110_2 = 134
→ Actual exponent = [Biased exponent] – bias = 134 – 127 = 7
Fraction = $000\ 0010\ 0000\ 0000\ 0000\ 0000_2$ = 000001_2

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal:

$1.000001_2 \times 2^7 = 10000010_2 = 130$

Therefore, the single-precision floating-point value 0x43020000 represents the decimal value **130**.

b. 0xc0f80000

0xc0f80000 = $1100\ 0000\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2$

Sign = 1 *(negative value)*
Biased exponent = 10000001_2 = 129
→ Actual exponent = [Biased exponent] – bias = 129 – 127 = 2
Fraction = $111\ 1000\ 0000\ 0000\ 0000\ 0000_2$ = 1111_2

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal:

$1.1111_2 \times 2^2 = 111.11_2 = 7.75$

Therefore, the single-precision floating-point value 0xc0440000 represents the decimal value **-7.75**.

c. 0x3eaaaaab

0x3eaaaaab = $0011\ 1110\ 1010\ 1010\ 1010\ 1010\ 1010\ 1011_2$

Sign = 0 *(positive value)*
Biased exponent = 01111101_2 = 125
→ Actual exponent = [Biased exponent] – bias = 125 – 127 = -2
Fraction = $010\ 1010\ 1010\ 1010\ 1010\ 1011_2$

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal. Note that, with so many bits in the fraction, I'm simply looking for an approximation:

$$1.01010101010101010101011_2 \times 2^{-2} = 0101010101010101010101011_2 \approx 0.33333334$$

Therefore, the single-precision floating-point value 0x3eaaaaab approximates the decimal value **0.33333334**.

d. 0xc17e0000

0xc17e0000 = $1100\ 0001\ 0111\ 1110\ 0000\ 0000\ 0000\ 0000_2$

Sign = 1 *(negative value)*
Biased exponent = 10000010_2 = 130
→ Actual exponent = [Biased exponent] – bias = 130 – 127 = 3
Fraction = $111\ 1110\ 0000\ 0000\ 0000\ 0000_2 = 111111_2$

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal:

$$1.111111_2 \times 2^3 = 1111.111_2 = 15.875$$

Therefore, the single-precision floating-point value 0xc17e0000 represents the decimal value **-15.875**.

e. 0xdeadbeef

0xdeadbeef = $1101\ 1110\ 1010\ 1101\ 1011\ 1110\ 1110\ 1111_2$

Sign = 1 *(negative value)*
Biased exponent = 10111101_2 = 189
→ Actual exponent = [Biased exponent] – bias = 189 – 127 = 62
Fraction = $010\ 1101\ 1011\ 1110\ 1110\ 1111_2$

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal. As in part c, I'm simply looking for an approximation:

$1.01011011011111011101111_2 \times 2^{62} \approx 6.2598534 \times 2^{18}$

Therefore, the single-precision floating-point value 0xdeadbeef approximates the decimal value **-6.2598534×2^{18}**.

Floating-point arithmetic:

a. `0x41900000 + 0x3fe00000`

Solution: Operands are as follows (I'm assuming you can handle the conversions without seeing all the steps):

0x41900000 = $1.001_2 \times 2^4$ (18_{10})
0x3fe00000 = $1.11_2 \times 2^0$ (1.75_{10})

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $1.11_2 \times 2^0 = 0.000111_2 \times 2^4$
- Add significands:
  - $1.001_2 + 0.000111_2 = 1.001111_2$
- Renormalize if necessary (not necessary in this case)
- Final result = $\mathbf{1.001111_2 \times 2^4}$ = **0x419e0000 in single-precision format** = 19.75_{10}

b. `0x40e00000 * 0x3e800000`

Solution: Operands are as follows:

0x40e00000 = $1.11_2 \times 2^2$ (7_{10})
0x3e800000 = $1.0_2 \times 2^{-2}$ (0.25_{10})

To multiply these numbers:

- Add the exponents to get the final exponent:
  - $2 + (-2) = 0$
- Multiply significands:
  - $1.11_2 * 1.0_2 = 1.11_2$
- Renormalize if necessary (not necessary in this case)
- Determine sign
  - Product of two positive values is positive → sign bit = 0
- Final result = $\mathbf{1.11_2 \times 2^0}$ = **0x3fe00000 in single-precision format** = 1.75_{10}

c. `0x40b00000 + 0xc0100000`

Solution: Operands are as follows:

0x40b00000 = $1.011_2 \times 2^2$ (5.5_{10})
0xc0100000 = $-1.001_2 \times 2^1$ (-2.25_{10})

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $-1.001_2 \times 2^1 = -0.1001_2 \times 2^2$
- Add significands:
  - $1.011_2 + (-0.1001)_2 = 0.1101_2$
- Renormalize if necessary:
  - $0.1101 \times 2^2 = 1.101 \times 2^1$
- Final result = $\mathbf{1.101_2 \times 2^1}$ = **0x40500000 in single-precision format** = 3.25_{10}

d. `0xc0800000 * 0x3f200000`

Solution: Operands are as follows:

0xc0800000 = $-1.0_2 \times 2^2$ (-4_{10})
0x3f200000 = $1.01_2 \times 2^{-1}$ (0.625_{10})

To multiply these numbers:

- Add the exponents to get the final exponent:
  - $2 + (-1) = 1$
- Multiply significands:
  - $1.0_2 * 1.01_2 = 1.01_2$
- Renormalize if necessary (not necessary in this case)
- Determine sign
  - Product of positive and negative value is negative → sign bit = 1
- Final result = $\mathbf{-1.01_2 \times 2^1}$ = **0xc0200000 in single-precision format** = -2.5_{10}