16.482 / 16.561: Computer Architecture and Design

Spring 2015

Homework #2 Solution

1. *(15 points) Show how the "refined multiply hardware" (slide #19 from Lec. 2) multiplies the 8-bit values 88 and 45, using an approach similar to the one demonstrated in lecture.*

Solution: I'll use 88 (which, as an 8-bit binary value, is 01011000_2) as the multiplicand and 45 (00101101_2) as the multiplier in this example. Since we're multiplying 8-bit values, we'll have a 16-bit product, and the algorithm will take 8 steps.

Initially, **product/multiplier = 0000000000101011 (underlined bit determines next step)**

Step 1: LSB of product/multiplier = 1 → add multiplicand into left half of register & shift right

```
         0000000000101101
    +    01011000
         0101100000101101
         0010110000010110   ← Product/multiplier after shift
```

Step 2: LSB = 0 → shift right

```
         0001011000001011   ← Product/multiplier after shift
```

Step 3: LSB = 1 → add multiplicand into left half of register & shift right

```
         0001011000001011
    +    01011000
         0110111000001011
         0011011100000101   ← Product/multiplier after shift
```

Step 4: LSB = 1 → add multiplicand into left half of register & shift right

```
         0011011100000101
    +    01011000
         1000111100000101
         0100011110000010   ← Product/multiplier after shift
```

Step 5: LSB = 0 → shift right

```
         0010001111000001   ← Product/multiplier after shift
```

Step 6: LSB = 1 → add multiplicand into left half of register & shift right

```
         0010001111000001
    +    01011000
         0111101111000001
         0011110111100000   ← Product/multiplier after shift
```

Step 7: LSB = 0 → shift right

```
         0001111011110000   ← Product/multiplier after shift
```

Step 8: LSB = 0 → shift right

$\mathbf{0000111101111000_2 = 3960_{10}}$ ← Final result

2. *(10 points) In class, we briefly discussed that the hardware methods presented cannot correctly handle signed multiplication. Describe a method that could correctly handle all multiplication of signed integers without increasing the size of the registers or adders used. (In other words, any other algorithm that adds extra bits while performing the actual multiplication—including Booth's algorithm, for example—is not a valid solution.)*

Solution: Several possible solutions exist, but the easiest would be to handle integer multiplication almost the same way you handle floating point multiplication—multiply the magnitudes and figure out the correct sign when that's done. Multiplier hardware that functions in this manner does need to invert any negative operand in order to get its magnitude before doing the multiplication.

3. *(25 points) Convert each of the following decimal values into single-precision IEEE floating-point format. Show all steps, including how you calculate the fraction and biased exponent stored in the number. (Note: I encourage you to convert each result into hexadecimal, which will help ensure that your assignments are graded and returned relatively quickly!)*

a. -9.625

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.

$9.625 = 1001.101_2 = 1.001101 \times 2^3$

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

Sign = 1 *(negative value)*
Exponent = [actual exponent] + bias = 3 + 127 = 130 = 10000010_2
Fraction = 001101_2 = $001\ 1010\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

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

-9.625 = $1100\ 0001\ 0001\ 1010\ 0000\ 0000\ 0000\ 0000_2$ = **0xC11A0000** *b. 23*

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

$23 = 10111_2 = 1.0111 \times 2^4$

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

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = 4 + 127 = 131 = 10000011_2
Fraction = 0111_2 = $011\ 1000\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

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

$23 = 0100\ 0001\ 1011\ 1000\ 0000\ 0000\ 0000\ 0000_2$ = **0x41B80000**

c. 0.921875

Solution: Although it may be a little difficult to see at first, this value is a sum of powers of 2: 0.921875 = 1/2 + 1/4 + 1/8 + 1/32 + 1/64 = 0.5 + 0.25 + 0.125 + 0.03125 + 0.015625. Therefore:

$0.921875 = 0.111011_2 = 1.11011 \times 2^{-1}$

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

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = -1 + 127 = 126 = 01111110_2
Fraction = 11011_2 = $110\ 1100\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

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

0.921875 = 0011 1111 0110 1100 0000 0000 0000 0000 = **0x3F6C0000** *d. -100.125*

Solution: We again start by converting this value to binary. Note that the fractional part is 1/8:

$$100.125 = 1100100.001_2 = 1.100100001 \times 2^6$$

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 = 100100001_2 = $100\ 1000\ 0100\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

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

-100.125 = 1100 0010 1100 1000 0100 0000 0000 0000 = **0xC2C84000**

e. 2.05 (determine the closest approximation you can)

Solution: While the whole part of this value is a power of 2 (2^1), the fractional part can't be exactly represented. The closest approximation we get with a 23-bit fraction is:

$$2.05 \approx 10.0000110011001100110011_2 = 1.00000110011001100110011_2 \times 2^1$$

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

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = 1 + 127 = 128 = 10000000_2
Fraction = $000\ 0011\ 0011\ 0011\ 0011\ 0011_2$ *(fraction is 23 bits)*

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

2.05 = 0100 0000 0000 0011 0011 0011 0011 0011= **0x40033333**

4. *(25 points) Convert each of the following IEEE single-precision floating-point values into decimal values. Show all steps of your work.*

a. *0x40540000*

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):

0x40540000 = $0100\ 0000\ 0101\ 0100\ 0000\ 0000\ 0000\ 0000_2$

Sign = 0 *(positive value)*
Biased exponent = 10000000_2 = 128
→ Actual exponent = [Biased exponent] – bias = 128 – 127 = 1
Fraction = $101\ 0100\ 0000\ 0000\ 0000\ 0000_2$ = 10101_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:

$1.10101_2 \times 2^1 = 11.0101_2 = 3.3125$

Therefore, the single-precision floating-point value 0x40540000 represents the decimal value **3.3125**.

b. *0xbfb00000*

Solution: 0xbfb00000 = $1011\ 1111\ 1011\ 0000\ 0000\ 0000\ 0000\ 0000_2$

Sign = 1 *(negative value)*
Biased exponent = 01111111_2 = 127
→ Actual exponent = [Biased exponent] – bias = 127 – 127 = 0
Fraction = $011\ 0000\ 0000\ 0000\ 0000\ 0000_2$ = 011_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:

$1.011_2 \times 2^0 = 1.375$

Therefore, the single-precision floating-point value 0xbfb00000 represents the decimal value **-1.375**.

c. 0x3f538000

Solution: 0x3f5380000 = $0011\ 1111\ 0101\ 0011\ 1000\ 0000\ 0000\ 0000_2$

Sign = 0 *(positive value)*
Biased exponent = 01111110_2 = 126
→ Actual exponent = [Biased exponent] – bias = 126 – 127 = -1
Fraction = $101\ 0011\ 1000\ 0000\ 0000\ 0000_2 = 10100111_2$

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:

$1.10100111_2 \times 2^{-1} = 0.110100111_2 = 0.8261719$

Therefore, the single-precision floating-point value 0x3f538000 represents the decimal value **0.8261719**.

d. 0xc2060000

Solution: 0xc2060000 = $1100\ 0010\ 0000\ 0110\ 0000\ 0000\ 0000\ 0000_2$

Sign = 1 *(negative value)*
Biased exponent = 10000100_2 = 132
→ Actual exponent = [Biased exponent] – bias = 132 – 127 = 5
Fraction = $000\ 0110\ 0000\ 0000\ 0000\ 0000_2 = 000011_2$

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:

$1.000011_2 \times 2^5 = 100001.1_2 = 33.5$

Therefore, the single-precision floating-point value 0xc2060000 represents the decimal value **-33.5**.

e. 0xaabbccdd (determine the closest approximation you can)

Solution: 0xaabbccdd = 1010 1010 1011 1011 1100 1100 1101 1101_2

Sign = 1 *(negative value)*
Biased exponent = 01010101_2 = 85
→ Actual exponent = [Biased exponent] – bias = 85 – 127 = -42
Fraction = 011 1011 1100 1100 1101 1101_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:

$$1.011\ 1011\ 1100\ 1100\ 1101\ 1101_2 \times 2^{-42} \approx 3.3360025 \times 10^{-13}$$

Therefore, the single-precision floating-point value 0xaabbccdd approximates the decimal value **$-3.3360025 \times 10^{-13}$**.

5. *(25 points) Compute the result of each floating-point arithmetic arithmetic operation below, in which each of the values is encoded in single-precision IEEE floating-point format. Recall that:*

   - *For floating-point addition, align the binary points, add the significands, then normalize the result.*
   - *For floating-point multiplication, add the exponents (taking care to only account for the bias once), multiply the significands, normalize the result, and then determine the sign.*

All arithmetic should be done in binary, and results should be re-encoded in single-precision IEEE floating-point format.

a. `0x41e00000 + 0x42280000`

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

0x41e00000 = $1.11_2 \times 2^4$ (28_{10})
0x42280000 = $1.0101_2 \times 2^5$ (42_{10})

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $1.11_2 \times 2^4 = 0.111_2 \times 2^5$
- Add significands:
  - $1.0101_2 + 0.111_2 = 10.0011_2$
- Renormalize if necessary
  - $10.0011_2 \times 2^5 = 1.00011 \times 2^6$
- Final result = $\mathbf{1.00011_2 \times 2^6}$ = **0x428c0000 in single-precision format** = 70_{10}

b. `0x40b80000 * 0x40500000`

Solution: Operands are as follows:

0x40b80000 = $1.0111_2 \times 2^2$ (5.75_{10})
0x40500000 = $1.101_2 \times 2^1$ (3.25_{10})

To multiply these numbers:

- Add the exponents to get the final exponent:
  - $2 + 1 = 3$
- Multiply significands:
  - $1.0111_2 * 1.101_2 = 10.0101011_2$
- Renormalize if necessary
  - $10.0101011_2 \times 2^3 = 1.00101011_2 \times 2^4$
- Determine sign
  - Product of two positive values is positive → sign bit = 0
- Final result = **$1.00101011_2 \times 2^4$ = 0x41958000 in single-precision format** = 18.6875_{10}

c. `0xc1280000 + 0xc2308000`

Solution: Operands are as follows:

0xc1280000 = $-1.0101_2 \times 2^3$ (-10.5_{10})
0xc23080000 = $-1.01100001_2 \times 2^5$ (-44.125_{10})

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $-1.0101_2 \times 2^3 = -0.010101_2 \times 2^5$
- Add significands:
  - $(-0.010101)_2 + (-1.01100001)_2 = -1.10110101_2$
- Renormalize if necessary (not necessary in this case)

Final result = **$-1.10110101_2 \times 2^5$ = 0xc25a8000 in single-precision format** = -54.625_{10}

d. `0x41240000 * 0xc1110000`

Solution: Operands are as follows:

0x41240000 = $1.01001_2 \times 2^3$ (10.25_{10})
0xc1110000 = $-1.0010001_2 \times 2^3$ (-9.0625_{10})

To multiply these numbers:

- Add the exponents to get the final exponent:
    - $3 + 3 = 6$
- Multiply significands:
    - $1.01001_2 * 1.0010001_2 = 1.011100111001_2$
- Renormalize if necessary (not necessary in this case)
- Determine sign
    - Product of positive and negative value is negative → sign bit = 0
- Final result = **$1.011100111001_2 \times 2^6$ = 0xc2b9c800 in single-precision format**
  $= -92.890625_{10}$