**Computer Arithmetic 2016–2017** | **Homework 6 Solutions**

# 1 Double rounding

I will try here to provide additional explanations since most students face problems while solving this question. I do not expect the students to write all this.

1. In regular binary floating point, the RZ mode is equivalent to a truncation. If we truncate once at a wider precision then again at a narrower precision this will not be different from a single truncation directly at the narrower precision. Hence, double rounding is not a problem for RZ in regular binary.

   For the RNA, we may get an example as 1.10010 and we want to round to two bits only but round first to three bits then to two bits. The first rounding leads to 1.101 which is then rounded to 1.11. A single rounding leads to 1.10. Hence, double rounding may lead to an error for RNA.

   Similarly, for RNE the same example but with 1.10011 shows that double rounding may lead to an error in RNE.

   On the other hand, for RP and RM, double rounding will not lead to a problem. To prove it for RP, we can enumerate the different cases when we round at $t'$ bits then later to $t$ bits.

   $$x.\overbrace{\underbrace{xxxx\ldots xx}_{t \text{ bits}}xxx\ldots xx}^{t' \text{ bits}}\underbrace{xxxx\ldots xxxx}_{\text{discarded part}}$$

   (a) For a negative result, the RP for regular binary is equivalent to a truncation. Hence, a truncation at $t'$ to remove the discarded part followed by another at $t$ gives the same result as a direct truncation at $t$.

   (b) For a positive result:

   - If the discarded part beyond the $t'$ bits has no bits of value one, i.e. all the bits are zero and the sticky bit is zero. Hence, the rounding to $t'$ does not change the value. A second rounding to $t$ yields the correct result.
   - If there are any bits equal to one in the discarded part, the sticky is one and the value is incremented when rounded to $t'$.
     i. If the part between $t$ and $t'$ is all ones, the incrementation leads to a carry propagation all the way to the $t$ part and makes all the bits beyond $t$ equal to zero. A second rounding at $t$ does not change the value. A direct rounding to $t$ yields the

same result.

| | |
|---|---|
| Before rounding to $t'$ | $x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}111\ldots11}^{t' \text{ bits}}\underbrace{xxx1x\ldots xxxx}_{\text{discarded part}}$ |
| After rounding to $t'$ | $x.\overbrace{xxxxx\ldots xx000\ldots00}^{t' \text{ bits}}$ |
| | $\underbrace{\qquad\quad+1}_{t \text{ bits}}$ |
| After the second rounding to $t$ | $x.xxxxx\ldots xx$ |
| | $\underbrace{\qquad\quad+1}_{t \text{ bits}}$ |
| Before a direct rounding to $t$ | $x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}111\ldots11}^{t' \text{ bits}}\underbrace{xxx1x\ldots xxxx}_{\text{discarded part}}$ |
| After the direct rounding to $t$ | $x.xxxxx\ldots xx$ |
| | $\underbrace{\qquad\quad+1}_{t \text{ bits}}$ |

ii. If the part between $t$ and $t'$ is not all ones, then the rounding at $t'$ will just increment this part. The following rounding to $t$ takes that incrementation in effect in its sticky bit. That means that any bits that are discarded after the $t'$ are taken into account later in the rounding to $t$ and hence the result is correct.

These various cases show that a second rounding to $t$ yields the correct result always for RP. Since RM(x)=-RP(-x), we are sure that double rounding does not lead to a problem in RM either in the regular binary representations.

To conclude, double rounding does not lead to any problems in RZ, RP, or RM. Double rounding may lead to mistakes in the two RN (RNA and RNE). It is important to stress here that we are deriving all these results for the regular binary representations. If the numbers are represented using signed digits or in any other way, the results may be different.

2. In the addition or subtraction of floating point numbers, we align the significands by shifting the significand of the smaller operand to the right. The shift amount is equal to the difference of the exponents of the two operands. Let us analyze the possibility of rounding the result of an addition or subtraction to $t' \geq 2t+2$. Due to the alignment shift, the result may have many bits to the right of the rounding position $t$.

   (a) If the number of those bits is smaller than $t$ the result is represented accurately within $t'$ and double rounding will not cause a problem.

$$x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}\underbrace{xxx\ldots xx}_{<t \text{ bits}}000\ldots000}^{t' \text{ bits}}$$

   (b) If the number of bits is equal to $t$,

$$x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}\underbrace{xxx\ldots xx}_{t \text{ bits}}00}^{t' \text{ bits}}$$

   the use of $t'$ is still sufficient to represent an accurate result and to preform a later rounding.

(c) If the number of bits is equal to $t+1$,

$$x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}\underbrace{xxxx\ldots xx}_{t+1 \text{ bits}}}^{t' \text{ bits}}0$$

this means that the smaller operand has been shifted completely out of the range of bits for the larger operand and that the $G$ position for the rounding is equal to the integer bit of the smaller operand (either the hidden one or zero if it was a subnormal number). The use of $t'$ is still sufficient.

(d) If the number of bits is equal to $t+2$,

$$x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}\underbrace{0xxxx\ldots xx}_{t+2 \text{ bits}}}^{t' \text{ bits}}$$

this means that the $G$ is zero and the $R$ position holds the value of the integer bit. The use of $t'$ is still sufficient.

(e) If the number of bits is larger than $t+2$,

$$x.\overbrace{\underbrace{xxxxx\ldots xx}_{t \text{ bits}}\underbrace{00xxx\ldots xx}_{t+2 \text{ bits}}}^{t' \text{ bits}}\underbrace{xxxxx\ldots xxxx}_{\text{discarded part}}$$

this means that the $G$ and $R$ positions are both zeros. In this case, rounding in any of the IEEE modes is dependent on the sticky bit (and the sign bit for RP and RM). However, the effect of a rounding at $t'$ will take into effect any bits in the discarded part similar to what we have seen earlier. Even in the case that the rounding at $t'$ causes a carry propagation, that carry will stop at the $R$ position and will not affect the $G$ position. Hence, a second rounding to $t$ gives the same result as a direct rounding.

In conclusion, $t' \geq 2t+2$ is a sufficient condition to guarantee that double rounding does not lead to a problem for addition and subtraction.

3. With the inclusion of a hidden one, each of the two operands has $t+1$ bits. The multiplication of two $t+1$ bit numbers produces a number with at most $2t+2$ bits. Hence, the use of $t' \geq 2t+2$ guarrantees that we can represent the result accurately without any loss. A later rounding to $t$ will thus produce the correctly rounded result.

4. Double precision has $t' = 52$ while single precision has $t = 23$. Since $52 \geq 2 \times 23 + 2$ then, according to the proofs just presented, double rounding is not a problem. In real processors that support both precisions, the designers actually use one FPU for the double precision and round the result to the single precision when needed. The proofs in this question are the reason they can do it.

---

# 2 Division in IEEE 754-2008

1. Let us assume that $c = a/b$ and that $c$ has $p+1$ bits in its significand. This means that $c = \sum_{i=\ell}^{p+\ell} c_{-i}2^{-i}$ and the least significant bit of $c$ is of the value $c_{-p-\ell}$ at the weight of $2^{-p-\ell}$ with $c_{-p-\ell} = 1$ while $c_{-\ell} = 1$ in order to fill $p+1$ bits.

   Because the operation is division, we know that $b \neq 0$. Hence, $b$ has one or more bits of value '1'. Let us assume the most significant non-zero bit of $b$ is at position $-j$ with weight $2^{-j}$ and the least significant non-zero bit is at position $-k$ with weight $2^{-k}$. If $b$ is a normalized number then $-j = 0$ while if $b$ is subnormal then $-j < 0$. If $b$ has only one non-zero bit then $j = k$, otherwise

$-j > -k$. The value of $b$ spans the positions from $-j$ to $-k$ and the number of significant bits in $b$ is thus $-j + k + 1$.

The least significant bit of the product $b \times c$ is of weight $2^{-k} \times 2^{-p-\ell} = 2^{-k-p-\ell}$ while the most significant bit is of weight $2^{-j} \times 2^{-\ell} = 2^{-j-\ell}$. Hence, the number of bits in the product $b \times c$, with the given assumption on the number of bits in $c$, is $(-j+k+p+1)$. However, we know that $a$ has at most $p$ bits only which proves that our assumption that $c$ has $p+1$ bits is impossible.

The tie case occurs when there is exactly only one bit equal to one followed by all bits equal to zero as in

$$\boxed{\longleftarrow \ p \text{ digits} \longrightarrow} \ 1000\ldots$$

which is a number represented in $p+1$ bits. We have just proven that an exact result of $p+1$ bits can never occur, hence the tie case can never occur.

2. By definition, the different round to nearest directions differ only in the tie case. Since the tie case never occurs for the binary division operation then all the rounding to nearest directions are equivalent. Moreover, for positive results, the RTZ is equivalent to RMI since both tend to reduce the magnitude of a positive number and the RAZ is equivalent to RPI since both tend to increase the magnitude of a positive number. For negative results, RTZ is equivalent to RPI while RAZ is equivalent to RMI.

3. A simple example is enough to prove this statement. If $a = 200\cdots01 \times 10^8$ ($p$ digits) and $b = 00\cdots02 \times 10^3$ (also $p$ digits) then $a/b = 100\cdots005 \times 10^5$ in $p+1$ digits and it gives the exact tie case.

---

# 3 Square root in IEEE 754-2008

1. Obviously, the invalid flag may be raised if the input is sNaN while the inexact flag may be raised if a rounding is needed as in the case of $\sqrt{2}$. This operation is not division so the divide by zero flag is not raised. Furthermore, the exponent of the result in the square root operation is half the exponent of the input. Since the input is representable within the range of the decimal64 format then the result is definitely within the range. Hence, the overflow and underflow flags are never raised. In conclusion, the statement is true.

2. Let us assume that $c^2 = d$ and that $c$ has $p+1$ digits in its significand. This means that $c = \sum_{i=0}^{p} c_{-i}10^{-i}$ and the least significant digit of $c$ is of the value $c_{-p}$ at the weight of $10^{-p}$ with $0 < c_{-p} < 10$. For $c^2$, the least significant digit will thus be of value $c_{-p}^2 10^{-2p}$. Since the radix of the system $10 = 5 \times 2$ does not contain any square terms in its factors then $c_{-p}^2$ can never be a multiple of 10 meaning that the least significant digit of $c^2$ is not zero. Similarly, the most significant digit of $c^2$ ($c_0^2 10^0$) is not zero. Hence, $c^2$ should have at least $2p+1$ digits in the significand. However, we know that $d$ has at most $p$ digits only so $c$ cannot have $p+1$ digits.

   The tie case occurs when there is exactly only one digit equal to five followed by all digits equal to zero as in

$$\boxed{\longleftarrow \ p \text{ digits} \longrightarrow} \ 5000\ldots$$

   which is a number represented in $p+1$ digits. We have just proven that an exact result of $p+1$ digits can never occur, hence the tie case can never occur.

3. By definition, the different round to nearest directions differ only in the tie case. Since the tie case never occurs for the square root operation then all the rounding to nearest directions are equivalent. Moreover, since the result is always positive, the RTZ is equivalent to RMI since both tend to reduce the magnitude of a positive number and the RAZ is equivalent to RPI since both tend to increase the magnitude of a positive number.

---

# 4 Integrated Rounding

1. The table is

| $L$ | $G$ | $S$ | $C_r$ |
|---|---|---|---|
| 0 | 0 | 0 | $x$ |
| 0 | 0 | 1 | $x$ |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | $x$ |
| 1 | 0 | 1 | $x$ |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

2. The table in the case of one bit right shift is

| Without right shift | | | | | After 1-bit right shift | | | |
|---|---|---|---|---|---|---|---|---|
| $N'$ | $L'$ | $G'$ | $(R'+S')$ | $C_r$(pre) | $L$ | $G$ | $S$ | $C_r$(post) |
| 0 | 0 | 0 | 0 | $x$ | 0 | 0 | 0 | $x$ |
| 0 | 0 | 0 | 1 | $x$ | 0 | 0 | 1 | $x$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | $x$ |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | $x$ |
| 0 | 1 | 0 | 0 | $x$ | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | $x$ | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | $x$ | 1 | 0 | 0 | $x$ |
| 1 | 0 | 0 | 1 | $x$ | 1 | 0 | 1 | $x$ |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | $x$ |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | $x$ |
| 1 | 1 | 0 | 0 | $x$ | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | $x$ | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

3. We choose all the don't care cases to be equal to 1 except for the rows when $L'G'(R'+S') = 010$ and $N'L'G'(R'+S') = 0100$. In these rows, both $C_r$(pre) and $C_r$(post) are set to 0. Hence the equation is

$$\begin{aligned} C_r(\text{pre}) &= \overline{\overline{L'}G'\overline{(R'+S')} + \overline{N'}L'\overline{G'}\,\overline{(R'+S')}} \\ &= (L' + \overline{G'} + R' + S')(N' + \overline{L'} + G' + R' + S') \\ &= N'L' + L'G' + N'\overline{G'} + \overline{L'}\,\overline{G'} + R' + S' \\ &= N'(L' + \overline{G'}) + L' \oplus \overline{G'} + R' + S'. \end{aligned}$$

4. The table in the case of one bit left shift is

| Without leftt shift | | | | | After 1-bit left shift | | | |
|---|---|---|---|---|---|---|---|---|
| $L'$ | $G'$ | $R'$ | $S'$ | $C_r$(pre) | $L$ | $G$ | $S$ | $C_r$(post) |
| 0 | 0 | 0 | 0 | $x$ | 0 | 0 | 0 | $x$ |
| 0 | 0 | 0 | 1 | $x$ | 0 | 0 | 1 | $x$ |
| 0 | 0 | 1 | 0 | $x$ | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | $x$ | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | $x$ |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | $x$ |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | $x$ | 0 | 0 | 0 | $x$ |
| 1 | 0 | 0 | 1 | $x$ | 0 | 0 | 1 | $x$ |
| 1 | 0 | 1 | 0 | $x$ | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | $x$ | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | $x$ |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | $x$ |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

We choose all the don't care cases to be equal to 1 except for the rows when $L'G'R'S' = 0010$, 0100, 1010, or 1100. In these rows, both $C_r$(pre) and $C_r$(post) are set to 0. Hence the equation is

$$C_r(\text{pre}) \quad = \quad S' + G'R'.$$

5. The combination gives

$$\begin{aligned} C_r(\text{pre}) &= \overline{\text{sub}}(N'L' + L'G' + R' + S') + \text{sub}(S' + G'R') \\ &= S' + \overline{\text{sub}}(N'L' + L'G' + R') + \text{sub}G'R' \end{aligned}$$

---

# 5 Hypotenus function

1. The value of $a^2 = 9 \times 10^{320}$ overflows beyond the maximum representable value in binary64 but remains representable in decimal64. Similarly for $b^2 = 16 \times 10^{320}$. Hence, in binary64 the result is $+\infty$ while in decimal64 the result is $5 \times 10^{160}$.

2. This modified code correctly handles the overflow cases and also avoids dividing by zero. The result for decimal64 is $5 \times 10^{160}$ as before. The result for binary64 is close to the correct result but suffers from the rounding errors in representing both $a$ and $b$ in binary64 as well as the other calculated values. In fact, $a$ is represented by a slightly smaller number starting with 299999999999999986349 ... while $b$ is represented by a slightly larger number starting with 400000000000000002611 .... The final result starts with 500000000000000018872946624114074 ...

3. The previous code already handles the case of both inputs being zero through the check on the value of `x`. The following code attempts to handle the other mentioned cases

```
if(isNaN(a) and isInfinite(b)) c=+inf;
else if(isInfinite(a) and isNaN(b)) c=+inf;
else { x = maxabs(a,b);
n = minabs(a,b);
c = (x == 0.0) ?  0.0 :  x*sqrt(1 + (n/x)^2);
}
```

4. For both inputs being $qNaN$, the result is $qNaN$ without signaling any exceptions. While for one or both inputs being $sNaN$, the result is $qNaN$ and the invalid operation exception is

signaled. The following code handles the above NaN cases as well as the case of one $qNaN$ with another finite input. Notice the order of checking the different cases.

```
if(isSignaling(a) or isSignaling(b)) {c=qNaN; signal invalid;}
else if(isNaN(a) and isNaN(b)) c=qNaN;
else if(isNaN(a) and isInfinite(b)) c=+inf;
else if(isInfinite(a) and isNaN(b)) c=+inf;
else if(isNaN(a) and isFinite(b)) c=qNaN;
else if(isFinite(a) and isNaN(b)) c=qNaN;
else { x = maxabs(a,b);
n = minabs(a,b);
c = (x == 0.0) ?  0.0 :  x*sqrt(1 + (n/x)^2);
}
```

---