Tutorial 8: Fractions

Problem 1: Fractional Computation

Consider the expression \((17/7)a\). What is the most accurate way to implement this using integer arithmetic in C?

```c
#include <stdio.h>

int main (void)
{
    int a = ?;
    int b, c, d, e;
    b = (17/7)*a;
    c = 2.4285714285714285714285714285714*a;
    d = (17*a)/7;
    e = 17*(a/7);
    printf("Number "\"(17/7)a\" as integer = \"%d\n\n", b);
    printf("Number "\"2.43a\" as integer = \"%d\n\n", c);
    printf("Number "\"(17a)/7\" as integer = \"%d\n\n", d);
    printf("Number "\"17(a/7)\" as integer = \"%d\n\n", e);
    return 0;
}
```

Figure 1: C-code for Fractional Computation

If we assume \(a = 89\), then the first alternative \(b = (17/7) \times a = 2 \times 89 = 178\). The second alternative \(c = 2.43 \times a = 2.43 \times 89 = 216\). The third alternative \(d = (17 \times a)/7 = (17 \times 89)/7 = 216\). The last alternative \(e = 17 \times (a/7) = 17 \times (89/7) = 204\). However, note that \(c = 2.43 \times a\) uses floating point hardware or emulation in software and therefore, requires a hardware coprocessor or several function calls. Therefore, it is not really integer arithmetic.

The printouts of the C program in Figure 1 for \(a = 89\) are presented in Figure 2.

Number \"(17/7)a\" as integer = \"178\n\nNumber \"2.43a\" as integer = \"216\n\nNumber \"(17a)/7\" as integer = \"216\n\nNumber \"17(a/7)\" as integer = \"204\n
Figure 2: The Outputs of `printf` Statements

On the other hand if we assume \(a = 442128986\) (a large number), then the first alternative \(b = (17/7) \times a = 884257972\). The second alternative \(c = 2.4285714287 \times a = 2.43 \times 442128986 = 1073741823\). The third alternative produces \(d = (17 \times a)/7 = (17 \times 442128986)/7 = \frac{1073741830}{7} = -153391690\), an overflow! The last alternative produces \(e = 17 \times (a/7) = 17 \times (442128986/7) = 17 \times 63161283 = 1073741811\). However, note that \(c = 2.43 \times a\) uses floating point hardware or emulation in software and therefore, requires a hardware coprocessor or several function calls. Therefore, it is not really integer arithmetic.

The printouts of the C program in Figure 1 for \(a = 442128986\) are presented in Figure 3.
Problem 2: An FIR Filter

Consider the expression for a FIR digital filter in (1):

\[ y[n] = \sum_{i=0}^{N-1} a_i x[n-i] \]  \hspace{1cm} (1)

\( N = 3, \ a_0 = 1.25, \ a_1 = 0.75, \) and \( a_2 = 2.125. \) What is the most accurate way to implement filter this using integer arithmetic in C?
Problem 3: Arithmetic Manipulation

Consider the expression \((3/4b)\). By going through the tutorial question 1 you decide to write the C code segment in Figure 6 to represents this expression in the integer arithmetic.

```
int b, d;
... ...
... 
d = (3*b)/4;
return d;
```

**Figure 6: The C-code for \((3/2b)\)**

You lab partner understands the content of tutorial question 1 better than you and suggests the code segment in Figure 7.

```
int b, d;
... ...
... 
d = b/2 + b/4;
```

**Figure 7: The modified C-code for \((3/4b)\) – Version 1**

You understand the thinking behind your friend’s modification and upstage him by suggesting the code in Figure 8.

```
int b, d;
... ...
... 
d = (b + b/2)/2;
```

**Figure 8: The modified C-code for \((3/4b)\) – Version 2**
How can you explain the modifications in Figure 7 and Figure 8?

**Problem 4: Division by powers of two**

Consider the expression \((1/2a + 3/4b + 9/4c)\). You lab partner going through the tutorial questions 1 and 2 writes the code segment in Figure 9 to represent this expression in integer arithmetic. He uses shift right by 2 (\(>>2\)) for division by 4 (\(1/4\)).

```c
int a, b, c, d;
...
...
d = (2*a + 3*b + 9*c)>>2;
return d;
}
```

*Figure 9: The C-code for \((1/2a + 3/4b + 9/4c)\)*

How do you explain your intelligent modification to your friend?

The in C language (like a calculator) integer division (/) operation truncates towards zero. That means \(5/4 = 1\) and \(-5/4 = -1\). However in C \(5 >> 2 = 1\) and \(-5 >> 2 = -2\), off by 1! Therefore, although the division by powers of two is symmetric around zero as it truncates towards \(0\), shift right operation is asymmetric and truncates towards \(-\infty\).

To rectify this problem we need to identify if the number is negative (bit 31 = 1) and if the bits being shifted out are different from zero. If both conditions are true then 1 is added to the shifted result. Code segment in Figure 10 achieves that.

```c
int a, b, c, d;
...
...
d = (2*a + 3*b + 9*c)>>2;
if((d < 0) && (d & 3))
    d = (d>>2) + 1;
else
    d = d >> 2;
return d;
}
```

*Figure 10: The modified C-code for \((1/2a + 3/4b + 9/4c)\) – Version 1*

**Problem 5: Multiplication by 0.7**

Write a C code to achieve multiplication by (0.7) using a series of shift sub/add.
A = 0.9

Repeat the loop below 32 times:
A = 2 x 0.9
if (A > 1)
insert 1 in the output
A = A – 1
else insert 0 in the output

Figure 11: Algorithm to convert 0.7 to its representation in binary fraction

```c
int main (void)
{
  double  a = 0.7;
  int    i;
  printf("%1.1f = a\n", a);
  for (i=0; i < 32; i++)
  {
    a = 2 * a;
    if (a > 1)
    {
      a = a - 1;
      printf("1\n");
    }
    else printf("0\n");
  }
  printf("\n");
  return 0;
}
```

Figure 12: The C-Code to convert 0.9 to its representation in binary fraction

The printouts of the C program in Figure 12 are presented in Figure 13.

0.7 = 0.10110011001100110011001100110011

Figure 13: The Outputs of `printf` Statements

The C program in Figure 12 implements multiplication by 0.7 using a sequence of add/sub and shift operations.
int mul_0p7 (int mulier)
{
    int a = mulier;
    /* 0.7 = 0.10110011001100110011001100110011
       0.10110011001100110011001100110011 can be written as
       + 0.1 = k0
       + 0.0011 = k1 = 0.11>>2
       + 0.00000011 = k2 = k1>>4
       + 0.000000000011001100 = k3 = (k1 + k2)>>8
       + 0.00000000000000000011001100110011 = k4 = (k1 + k2 + k3)>>16
    */
    a = a - (a >> 2);  // scale a*0.0011 to a*0.11 and rewrite as a*(1-1/4)
    a = a + (a >> 4);        // add 1/16 of the computed value to itself
    a = a + (a >> 8);        // add 1/256 of the computed value to itself
    a = a + (a >> 16);       // add 1/65536 of the computed value to itself
    a = a >>2;               // scale it back by 2 bits
    a = a + (mulier>>1);     // add k0 = 0.5 to it
    return a;
}

int main (void)
{
    int n = 10;
    printf("Number 0.7*n as integer = " "%d"

", mul_0p7(n));
    return 0;
}

Figure 14: The Outputs of printf Statements

There is still one problem with this program! For certain input values the computed results
are lower by value "1" than the values computed by expression (n*0.7).
We see that if the result is correct by one then the expression (7*n - (0.7*n)*10 = 0 …
9). This can be rewritten as (7*n - (0.7*n)*10 – 10 = -10 … -1). However, if the result
is less by 1 then we have (7*n - (0.7*n)*10 – 10 = 0 … 9). The insertion of the C
statement (mulier*7 -a*10 - 10) >0) a = a+1; at the end of the function
mul_0p7(int mulier) will do the check and accordingly correct the result.