

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?

The C code in Figure 1 provides several alternatives.

```
#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 $b = (17/7) * a = 2 * 89 = 178$. The second alternative $c = 2.43 * a = 2.43 * 89 = 216$. The third alternative $d = (17 * a)/7 = (17 * 89)/7 = 1513/7 = 216$. The last alternative $e = 17 * (a/7) = 17 * (89/7) = 17 * 12 = 204$. However, note that $c = 2.43 * a$ 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\"

Number \"2.43a\" as integer = \"216\"

Number \"(17a)/7\" as integer = \"216\"

Number \"17(a/7)\" as integer = \"204\"
```

Figure 2: The Outputs of printf Statements

On the other hand if we assume $a = 442128986$ (a large number), then the first alternative $b = (17/7) * a = 884257972$. The second alternative $c = 2.4285714287 * a = 2.43 * 442128986 = 1073741823$. The third alternative produces $d = (17 * a)/7 = (17 * 442128986)/7 = -1073741830/7 = -153391690$, an overflow! The last alternative produces $e = 17 * (a/7) = 17 * (442128986/7) = 17 * 63161283 = 1073741811$. However, note that $c = 2.43 * a$ 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.

```

Number "(17/7)a" as integer = "884257972"
Number "2.43a" as integer = "1074373435"
Number "(17a)/7" as integer = "-153391690"
Number "17(a/7)" as integer = "1073741811"

```

Figure 3: The Outputs of printf Statements

Problem 2: An FIR Filter

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

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

$N = 3$, $a_0 = 1.25$, $a_1 = 0.75$, and $a_2 = 2.125$. What is the most accurate way to implement filter this using integer arithmetic in C?

The whole system architecture for the FIR filter is shown in Figure 4. The A/D converts an analogue input signal to an 8-bit digital signal. After the filtering by the FIR filter the digital signal is converted back into an analogue output using the D/A converter.

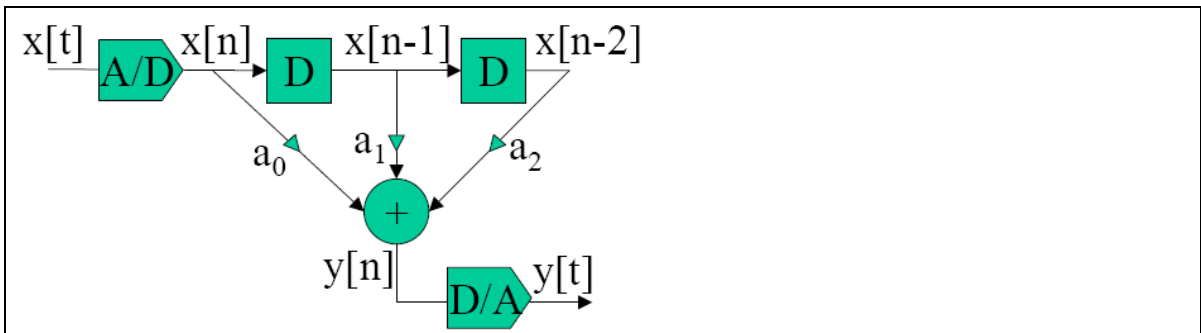


Figure 4: The System Structure for a FIR Filter.

The C code in Figure 5 provides a direct implementation of the digital filter in (1) and in Figure 4. However, the C program presented in Figure 5 has a major problem. The declaration of the array `a[]` as `(int a[] = {1.25, 0.75, 2.125})`, effectively stores the elements of array `a[]` as `(int a[] = {1, 0, 2})`. Assuming that three consecutive samples of the signal `x[i]` are the maximum value of 127 (or -127), then this gives rise to a maximum error of $((523 - 381)/523 = 27\%)$. This problem can be rectify by scaling `a[]` by 8 as `(int a[] = {10, 6, 10})`. And changing the statement `(write_output(y);)` to `(int write_output(y/8);)`.

However, there is still a problem, the function `int write_output(out_val)` takes a `char` type as its parameter. But the maximum value of 523 does not fit in 8 bits, This causes an overflow. To rectify the problem we need a scaling by 4. This will change the call to the function to `(write_output(y/32);)`.

```

extern char read_input(void);
extern void write_output(char out_val);
#define N 3

int main (void)
{
    int x [] = {0, 0, 0};
    int i, y;
    int a [] = {1.25, 0.75, 2.125};

    while (1)
    {
        y = 0;
        x[2] = read_input();
        for (i=0; i < N; i++)
        {
            y = y + a[i]*x[N-1-i];
        }
        write_output(y);
        for (i=1; i < N; i++)
        { x[i-1] = x[i];
        }
    }
    return 0;
}

```

Figure 5: The C-Code for the FIR Filter

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 represent 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?

In the integer multiplication, we need to guard against the overflow. The product of two integer numbers should fit in 32 bits. If we know that $(3*b)$ can cause overflow then we write $(3/4 = 1/2 + 1/4)$. This modification employed in Figure 7 will ensure that there is no overflow. However, division operations $(/2)$ and $(/4)$ cause loss of accuracy. Replacing $3/4$ by $((1 + 1/2)/2)$ produces the maximum accuracy in the integer arithmetic. Another solution that was presented in the (A/10) operation in the lecture notes is to write $3/4$ as $(1 - 1/4)$. While $((1 + 1/2)/2)$ truncates towards zero, $(1 - 1/4)$ truncates towards $(+\infty)$.

Problem 4: Division by powers of two

Consider the expression $(1/2a + 3/4b + 9/4c)$. Your 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 ($1/4$).

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

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

Being a diligent and intelligent student (my mother always believed I was!), you recall, the discussion in the class (if you were the rare species who turned up for the class!) about the replacement of division by powers of two with the shift right operations. You decide to modify your friend's code to the code in Figure 10.

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

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.

Problem 5: Multiplication by 0.7

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

First, we need to write (0.7) as a binary fraction in 32 bits. The algorithm to do this is given in Figure 11, and its C Program is shown in Figure 12.

```

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

```

int main (void)
{
double a = 0.7;

int i;
printf("%1.1f = a");
for (i=0; i < 32; i++)
{a= 2*a;
  if (a>1)
  {
    a = a-1;
    printf("1");
  }
  else printf("0");
}
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
    + 0.000000000000000000011001100110011 = 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\"\\n\\n", 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 $(7*n - (0.7*n)*10 = 0 \dots 9)$. This can be rewritten as $(7*n - (0.7*n)*10 - 10 = -10 \dots -1)$. However, if the result is less by 1 then we have $(7*n - (0.7*n)*10 - 10 = 0 \dots 9)$. 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.