A Note on Division

Niklaus Wirth – 20.8.2008 / 1.10.2008

1. Introduction

Early computers had neither multiplication nor division in their instruction sets. These operations required too many latches and gates, and could well be implemented in software using addition, subtraction, and shifting. As computers became more potent, instructions for multiplication and division became expected ingredients.

With the advent of RISCs, the question of whether these two operations had justifiable places in the instruction set, became reanimated. After all, execution of every instruction in a single clock cycle was the fundamental premise in RISC design, and neither multiplication nor division could perform that fast without a disproportionate amount of circuitry. Moreover, it had become apparent that multiplication was relatively rarely needed, and division almost never. Therefore, their speed was a secondary issue. The well-known RISCs of the 1990s, such as MIPS and SPARC had them excluded and replaced with multiply-step and divide-step instructions. 32 of them in sequence provided a full multiplication or division.

We will discuss the question of how to best provide division on a computer lacking such an instruction. This is the case for the ARM, on which our tests were peformed.

2. Integer Division

Integer division can be done by the conventional algorithm using repeated subtraction and shifting by a single digit. First the divisor *y* is shifted left until it is larger than the dividend *x*. Let *q* be the quotient and *r* the remainder. We assume $0 \le y < x$. Note that multiplication by 2 is a left shift by 1 digit, and division by 2 by a right shift. With the precondition $0 < y \le x$, the result satisfies $q*y + r = x$ and $0 \le r < y$.

```
r := x; q := 0; y0 := y;
REPEAT y := 2*y UNTIL y > x;
REPEAT y := y DIV 2; q := 2*q;
   IF r >= y THEN r := r - y; q := q+1 END
UNTIL y = y0;
```

Typically, however, a version of this algorithm is chosen that uses a double length register *rq* holding both remainder and quotient. The number of steps is always equal to the word length *N*. (*Y* is *y* added to the upper half of *rq*).

```
rq := x; i := N;
REPEAT rq := 2*rq; i := i-1;
   IF r >= y THEN rq := rq - Y; q := q+1 END
UNTIL i = 0;
```

This version is shorter and therefore also faster than the first. It is used by the Oberon-ARM compiler producing in-line code. Then I recognized that for small quotients the fixed number of steps is a handicap, and the first version is definitely superior. How much? The following results show when and how much the first version is superior, although its code is 4 instructions longer. The time shown is for 10000 divisions by 3.

dividend	quotient	time in ms
10	3	160
100	33	320
1000	333	438
10000	3333	558
100000	33333	716
1000000	333333	836
10000000	3333333	955
100000000	33333333	1075

```
1000000000      333333333       1234
2147483647      715827882       1274
```

The second algorithm with a fixed number of steps uses 925 ms independent of the value of the quotient.

The code generated by the compiler for the ARM for the first version is the following. with variables x, y, y0, q, r in registers R11 … R7:

```
 3    E1A0700B    MOV   R7 R0 R11
 4    E3A08000    MOV   R8 R0 0
 5    E1A0A009    MOV   R10 R0 R9
 6    E1B0A08A    MOV   R10 R0 R10 LSL  1
 7    E15A000B    CMP   R0 R10 R11
 8    DAFFFFFC    BLE      -2
 9    E1B0A0CA    MOV   R10 R0 R10 ASR  1
10    E1B08088    MOV   R8 R0 R8 LSL  1
11    E157000A    CMP   R0 R7 R10
12    BA000001    BLT      3
13    E057700A    SUB   R7 R7 R10
14    E2988001    ADD   R8 R8 1
15    E15A0009    CMP   R0 R10 R9
16    1AFFFFF7    BNE      -7
```

The code becomes somewhat longer, if also negative dividends are admitted:

```
r := ABS(x); q := 0; y0 := y;
……….
IF x < 0 THEN q := -q;
   IF r # 0 THEN r := y - r; q := q-1 END
END
```

3. Real Division

Real numbers are represented in IEEE floating-point form, i.e. as

$$x = 1.m \times 2^{e+B} \qquad 1.0 \le m < 2.0 \qquad 0 < e < 255$$

m is called the mantissa and *e* the exponent. B = 127 for single and B = 1023 for double precision. A division z = x/y is computed by division of the mantissas and subtraction of the exponents.

```
x = 0.5     3F000000     (sign, 8-bit exponent, 23-bit mantissa)
x = 1.0     3F800000
x = 2.0     40000000
x = 3.0     40400000
```

Real division is therefore typically implemented by using the algorithm of integer arithmetic for the mantissas. In this case, the version using a fixed number of 24 steps is employed. Recip(x) = 1/x.

```
PROCEDURE* Recip(a: REAL): REAL; (*0.5 <= a < 1.0*)
   VAR x, y, q, i: INTEGER;
BEGIN y := SYSTEM.VAL(INTEGER, a) MOD 800000H + 800000H;
   x := 1000000H; i := 24; q := 0;
   REPEAT q := 2*q;
      IF x >= y THEN x := x - y; INC(q) END ;
      x := 2*x; DEC(i)
   UNTIL i = 0;
   RETURN SYSTEM.VAL(REAL, q MOD 800000H + 3F800001H)
END Recip;
```

Note: The assignments involving the type cast (SYSTEM.VAL) yield the mantissa of *a* represented as an integer, and the real result composed of an exponent and the mantissa q.

The alternative for computing y = 1/x is to use an iterative approximation using multiplication. There exist various methods. Best known is the one after Newton and Raphson. For the general case y = f(x) with x = a it is based on computing a root of the inverse function f^{-1} – a.

With $f(x) = f^{-1}(x) = 1/x = x^{-1}$, $f'(x) = -x^{-2}$, we obtain the recurrence relation

$$x_{i+1} = x_i - ((x_i^{-1} - a) / -x_i^{-2}) = x_i + (x_i - a*x_i^2) = x_i*(2 - a*x_i)$$

Hence, every iteration requires two multiplications. Whether this method can compete with the traditional integer method depends on the number of iterations required and on the speed of multiplication. Some modern processors indeed perform multiplication very fast. It appears counterintuitive that multiplication would be (almost) as fast as addition, out of which it is composed. However, this speed is achieved at the cost of a very substantial amount of circuitry, which all works in parallel.

The determination of the appropriate starting value x_0 requires additional effort, and the chances of the Newton method diminish. With an auxiliary variable z = a*x we obtain

```
x := x0; z := a*x;
REPEAT x := x * (2.0 – z); z := a*x UNTIL z # 1.0
```

The following test samples indicate the number *n* of required iterations.

a	x	n
0.5	2.0	7
0.8	1.25	5
0.95	1.0526316	4

Fortunately, a more effective algorithm is available, as represented by the following iteration with $0.5 \le a < 1$. The repetition is based on the invariant $a*x = 1 - z$ and $0 \le z \le 0.5$. A definite advantage is that it is unnecessary to select an initial value for x and that z converges quadratically to zero:

```
x := 1.0; z := 1.0 - a;
REPEAT x := x*(1.0+z); z := z*z UNTIL z = 0.0
```

The following test samples indicate the number *n* of required iterations.

a	x	n
0.5	2.0	5
0.8	1.25	4
0.95	1.0526316	3

The key for an efficient implementation is to use integer arithmetic, even if floating-point addition and multiplication were available. Fractional numbers are represented in fixed-point form. This is particularly appropriate in this case, because x and z vary within the known intervals $0,5 \le a < 1.0$ and $1.0 < x \le 2.0$.

```
PROCEDURE* Recip(a: REAL): REAL; (*0.5 <= a < 1.0*)
   VAR x, z, z1, ph, pl: INTEGER;
BEGIN x := 80000000H; (*1.0*)
   z := SYSTEM.VAL(INTEGER, a) MOD 800000H + 800000H;
   z := LSL(1000000H - z, 8);
   REPEAT
      SYSTEM.MULD(pl, x, z); x := ph + x;
      SYSTEM.MULD(pl, z, z); z := ph;
   UNTIL z < 100H;
   RETURN SYSTEM.VAL(REAL, LSR(LSL(x, 1), 9) + 3F800001H)
END Recip;
```

Note: MULD(p, x, y) denotes ph,pl := x*y.

The binary point is chosen to lie to the left of bit 30 for x, and to the left of bit 31 for z. This choice makes it unnecessary to adjust (shift) the resulting products. Furthermore, the expression x := x*z + x is used instead of x := x*(1.0+z), thereby avoiding the sum 1.0+z reaching or exceeding the value 1.0.

A comparison between the two methods yields the following, somewhat surprising times in ms:

divisions	shift-subtract	iterative
100	10	2
1000	92	20

4000	369	79

The iterative method is in the average about 5 times faster than the shift-subtract method with a fixed number of 24 steps. (For 1000 divisions, the time varies depending on the divisor from 18 to 22).

The code generated by the compiler is:

```
 3  E3A0A102    MOV   R10 R0 80000000H
 4  E1A0548B    MOV   R5 R0 R11 LSL  9
 5  E1B054A5    MOV   R5 R0 R5 LSR  9
 6  E2959502    ADD   R9 R5  800000H
 7  E2795401    RSB   R5 R9 1000000H
 8  E1B09405    MOV   R9 R0 R5 LSL  8
 9  E0876A99    MUL   R7 R6 R10 R9
10  E097A00A    ADD   R10 R7 R10
11  E0876999    MUL   R7 R6 R9 R9
12  E1A09007    MOV   R9 R0 R7
13  E3595C01    CMP   R5 R9 256
14  AAFFFFF9    BGE      -5
15  E1B0508A    MOV   R5 R0 R10 LSL  1
16  E1B054A5    MOV   R5 R0 R5 LSR  9
17  E59F4008    LDR   R4 PC 8
18  E095B004    ADD   R11 R5 R4
```

4. Conclusions

It appears that integer division is best computed by the well-known shift-subtract method. However, this holds only if contrary to conventional habit not a fixed number of steps given by the word length, but only as many as required are performed. The method is therefore fast for the frequent case of small quotients.

For division of REAL numbers, an iterative method using integer multiplication is recommended, if a fast multiply instruction is available. In this case, fixed-point arithmetic is mandatory.