How To Divide By 3

Introduction

Doing arithmetic on an FPGA is different from using a CPU. There is no single Arithmetic Logic Unit that receives operands from a fixed bank of registers and to which results are returned. There are various ways of mapping an algorithm onto the logic cells of an FPGA, but the general model is that separate hardware is generated for each expression to be evaluated. In Handel-C, the circuit to implement a function can be reused in different parts of a design. But consider code like this:

```
a = b + c;
c = d + e;
```

In a CPU, a single ALU would be used to perform both additions, but on the FPGA, two separate adders will be generated to perform them.
There are two tradeoffs that make the FPGA approach attractive compared to the general-purpose ALU approach used in a CPU:

1. The FPGA circuits are tailored to the calculations involved. If the variables in the example above are just 3 bits wide, then the adders will each consist of just 3 full adders each. No circuits are wasted providing a capability to add 32-bit operands when they are't needed, for example.
2. There are significant speed advantages possible using the FPGA approach. For example, if an FPGA can evaluate a complex expression like `a + b *c + d` in one clock cycle (using two adders and a multiplier), whereas the same calculation would require a sequence of three instructions (assuming the operands are in registers) to evaluate using a single ALU.

One goal of an FPGA design, then, is to eliminate as much unnecessary calculation as possible, thereby reducing the complexity of the circuitry the compiler generates. The complexity of an adder or subtracter grows linearly with the widths of the operands, but the complexity of a parallel multiplier grows with the square of the widths of the operands. Division is much more complex than multiplication. For example, to multiply a 20-bit number by the constant value 3 requires 19 logic elements on a Quartus Cyclone FPGA, but dividing a 20-bit number by 3 requires 141 logic elements. ("Logic elements" remains an undefined term at this point.) The question, then, is whether division by a small constant can be implemented more efficiently. The algorithm developed below divides a 20-bit number by three using 40 logic elements in the Quartus/Cyclone environment.

> Note that the technique here is probably not worth pursuing if your design involves just a small number of divide operations; as long as the design fits in the FPGA and meets any timing constraints that might exist, there is no need to go any further.

A complication involving division is that the result usually has a fractional component, which may need to be rounded. For the present discussion, we will assume you want to compute the average of three 20-bit signed integers to produce a signed quotient that has been rounded to the nearest integer value, also being represented in 20 bits.

Fixed-Point Numbers

Think of an integer as a number with a binary point at the right end. The bits in the integer have weights of … 2^2, 2^1, 2^0. But you can think of that same integer as having a fractional component without affecting the circuitry needed to do addition or subtraction. For example, you could position the binary point three positions over from the right end, giving bit weights of … 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}.

> Adding the integers 15 and -1 to produce 14 ($001111_2 + 111111_2 = 001110_2$ is identical to adding the fixed-point numbers 1.875 and -0.125 to produce +1.75 ($001.111_2 + 111.111_2 = 001.110_2$: the only difference is in how we humans interpret the location of the binary point.

In this example, the smallest fraction that can be represented is 0.125 (000.001_2), so we say this fixed point notation has a precision of 0.125 or, equivalently, of 3 fraction bits.

As long as all operands are represented using the same number of fraction bits (have the same precision), their binary points will line up, and integer arithmetic will work just fine for doing calculations. However, if the binary points don't automatically line up and/or you are working with a set of operands that sometimes need more fraction bits and sometimes need more integer bits to represent their values you would need to work with floating-point numbers. Floating-point numbers also let you work with a larger range of values for a given number of bits by using scale factors and approximations instead of exact representations of extremely large and extremely small values. The problem with floating-point numbers is that it takes a lot of circuitry do arithmetic using them. Fortunately, fixed-point notation is a totally satisfactory way to work with fractions for many FPGA-based applications, including the problem at hand: taking the average of three numbers.

Multiply Instead of Dividing

We already saw that multiplication requires less circuitry than division. One way to simplify things is to multiply by the reciprocal of the divisor instead of actually dividing. When the divisor is a constant, its reciprocal can be pre-computed. Of course the reciprocal of an integer will be a fraction. For example, to divide by 3, we can multiply by 1/3 ($0.010101\ldots_2$). Note that this particular value is a repeating fraction in both decimal (0.3333…) and binary. Two issues have to be dealt with: *precision* and *rounding*.

Assume we want to take the average of 3 integers and that the result will also be an integer. This will require four bits of precision to differentiate among the possible remainders (0, 1, or 2) reliably. So our multiplier will be 0.0101_2, (0.3125_{10}.) Six bits of precision would give a decimal value of 0.328125 for the multiplier, which should not improve the accuracy of the rounded integer result.

To round to the nearest integer, we can add 0.1000_2 (0.5_{10}) to the product before dropping the fraction bits to get the rounded integer quotient.

In this case, two bits of precision is the same as dividing by 4, because the first t bits of the multiplier are 0.01_2. The fraction would be equal to the rightmost t bits of the dividend. Three bits of precision would make the multiplier 0.01 which is numerically equivalent to the 2-bit multiplier. So you need to go to f bits to increase the actual precision.

Add Instead of Multiplying

With four bits of precision, multiplying by 1/3 is the same as multiplying by ¼ (0.0100_2), multiplying by 1/16 (0.0001_2), and adding the two products together. Here is an example of dividing the decimal number 8 by 3 and rounding the result:

```
  001000.0000   8.0

  000010.0000   2.0 = 8.0 / 4
+ 000000.1000   0.5 = 8.0 / 16
= 000010.1000   2.5 = 8 * 0.3125
+ 000000.1000   0.5 (rounding factor)
= 000011.0000   3.0 (rounded quotient)
  000011        3   (integer answer)
```

In summary, to compute the average of three 20-bit numbers:

- Sign-extend each of the three numbers to 22 bits and sum them. The extra two bits guarantee that the full sum will not overflow.
- Place an imaginary binary point four places in from the right of the 22-bit sum. Without actually doing anything to the bits, this is the same as multiplying the sum by 1/16. To be used later, this copy of the sum must be sign-extended to 24 bits.
- Sign-extend the sum to 24 bits and shift it left two bit positions. Again assuming an imaginary binary point four places in from the right, this value is the sum multiplied by ¼.
- Add the above two sums to produce a 24-bit sum with 4 fraction bits.
- Add the rounding factor and drop the fraction bits.

All of the above can be done in one clock cycle, but here is a Handel-C code snippet that shows the algorithm being done in 5 separate clocks:

```
sum               = adjs(x,22) + adjs(y,22) + adjs(z,22);
one_sixteenth     = adjs(sum, 24);
one_quarter       = (adjs(sum, 24) << 2);
avg_with_fraction = one_quarter + one_sixteenth;
average           = (avg_with_fraction + 0b01000) \\ 4;
```