Practical work nº 1 - Divider

1 Goal

Generation of a VHDL model of a generic divider for natural numbers.

2 Dividers (chapters 6 and 13 of [DES2006])

2.1 Basic algorithm

Let X and Y be two natural numbers with $Y > 0$. Define Q and R respectively as the *quotient* and the *remainder* of the division of X by Y, with an accuracy of p fractional bits by:

$$2^p.X = Q.Y + R,$$

where Q and R are natural numbers, and $R < Y$. In other words:

$$X = (Q.2^{-p}).Y + (R.2^{-p}), \text{ with } R.2^{-p} < Y.2^{-p},$$

so that the unit in the least significant position (*ulp*) of $Q.2^{-p}$ and $R.2^{-p}$ is equal to 2^{-p}. In the particular case where $p = 0$, that is

$$X = Q.Y + R, \text{ with } R < Y,$$

Q and R are the *quotient* and the *remainder* of the *integer division* of X by Y.

The basic algorithm applies to operands X and Y such that

$$X < Y.$$

In the general case, to ensure that $X < Y$, a previous alignment step is necessary: this will be seen in section 2.2.

The next theorem constitutes the justification of the basic division algorithm:

Theorem 1 – Fundamental equation of division

Given two natural numbers a and b such that $a < b$, then there exists two natural numbers q and r satisfying $2.a = q.b + r$, with $q \in \{0, 1\}$ and $r < b$.

Proof

If $2.a < b$, then $2.a = 0.b + r$ where $r = 2.a < b$. If $2.a \geq b$, then $2.a = 1.b + r$ where $r = 2.a - b < 2.b - b = b$.

The iterative application of the preceding theorem, i.e.

$$\begin{aligned} &2.r(0) = q(1).Y + r(1),\ r(1) < Y, \\ &2.r(1) = q(2).Y + r(2),\ r(2) < Y, \\ &\ldots \\ &2.r(p-1) = q(p).Y + r(p),\ r(p) < Y, \end{aligned} \tag{1}$$

with $r(0) = X$, generates the following relation

$$X.2^p = (q(1).2^{p-1} + q(2).2^{p-2} + \ldots + q(p).2^0).Y + r(p), \tag{2}$$

so that

$Q = q(1).2^{p-1} + q(2).2^{p-2} + ... + q(p).2^0$ and $R = r(p)$.

Assume that a procedure *division_step* has been defined

***procedure** division_step (a, b: **in** natural; q, r: **out** natural);*

that computes q and r such that $2.a = q.b + r$, with $q \in \{0, 1\}$ and $r < b$. Then the following basic division algorithm is a straightforward application of (1) and (2)

Algorithm 1 – Basic division
```
r(0) := X;
for i in 1 .. p loop
   division_step (r(i-1), Y, q(i), r(i));
end loop;
```

It generates the binary representation $q(1)$ $q(2)$ ... $q(p)$ of Q and the remainder $R = r(p)$. The *division_step* procedure is the following (see theorem 1):

Algorithm 2 – Base-2 division step
```
z := 2*a - b;
if z < 0 then q := 0; r := 2*a; else q := 1; r := z; end if;
```

The corresponding divider structure is shown in figure 1.

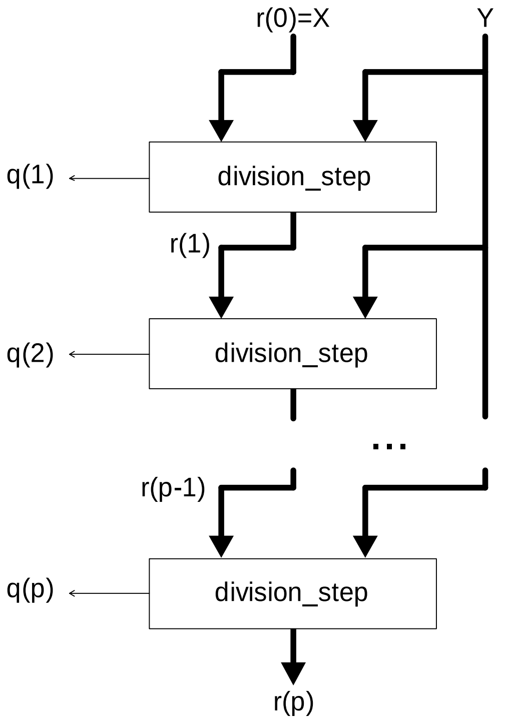

Figure 1 Divider structure

The following VHDL model describes the circuit of figure 1 in the case where both x and y are n-bit naturals:

```
library ieee;
use ieee.std_logic_1164.all;
package my_package is
   constant n: natural := 8;
   constant p: natural := 8;
   type remainders is array (0 to p) of
      std_logic_vector(n-1 downto 0);
   type long_remainders is array (1 to p) of
      std_logic_vector(n downto 0);
end my_package;

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.my_package.all;
entity practi1 is
port (
   x, y: in std_logic_vector(n-1 downto 0);
   quotient: out std_logic_vector(p-1 downto 0);
   remainder: out std_logic_vector(n-1 downto 0)
);
end practi1;

architecture rtl of practi1 is
   signal r: remainders;
   signal z: long_remainders;
begin
   r(0) <= x;
   iteration: for i in 1 to p generate
      z(i) <= (r(i-1)&'0') - ('0'&y);
      with z(i)(n) select quotient(p-i) <= '0' when '1',
         '1' when others;
      with z(i)(n) select r(i) <= (r(i-1)(n-2 downto 0)&'0') when '1',
         z(i)(n-1 downto 0) when others ;
   end generate;
   remainder <= r(p);
end rtl;
```

Observe that if $r(i\text{-}1)$ is smaller than y, and y is smaller than 2^n, then $2.r(i\text{-}1)\text{-}y$ is included between -2^n and -2^n, so that $z(i) = 2.r(i\text{-}1)\text{-}y$ is an $(n+1)$-bit 2's complement number.

2.2 Generic divider

The basic algorithm applies to operands X and Y such that $X < Y$. Assume now that X is an m-bit natural and Y an n-bit natural greater than 0. In order to compute the quotient and the remainder of the division of X by Y, with an accuracy of p fractional bits, first substitute Y by $Y' = 2^m.Y$ and p by $p+m$, and then use algorithm 1 for computing Q and R' such that

$$2^{p+m}.X = Q.Y' + R', \text{ with } R' < Y',$$

so that

$$2^p.X = Q.(Y'/2^m) + R'/2^m = Q.Y + R, \text{ with } R = R'/2^m < Y'/2^m = Y.$$

A first version of the circuit consists of substituting in the VHDL model of section 2.1

n by $new\_n = m+n$
X by $00...0X$, that is, an new_n-bit number equal to X,
Y by $Y00...0$, another new_n -bit number equal to $Y' = 2^m.Y$,
p by $new\_p = p+m$.

The following VHDL model describes a generic divider:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
package my_package is
   constant m: natural := 8;
   constant n: natural := 4;
   constant new_n: natural := m+n;
   constant p: natural := 6;
   constant new_p : natural := m+p;
   constant m_zeroes:
      std_logic_vector(m-1 downto 0) := (others => '0');
   constant n_zeroes:
      std_logic_vector(n-1 downto 0) := (others => '0');
   type remainders is array (0 to new_p) of
      std_logic_vector(new_n-1 downto 0);
   type long_remainders is array (1 to new_p) of
      std_logic_vector(new_n downto 0);
end my_package;

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.my_package.all;
entity practi1b is
port (
   x: in std_logic_vector(m-1 downto 0);
   y: in std_logic_vector(n-1 downto 0);
   quotient: out std_logic_vector(new_p-1 downto 0);
   remainder: out std_logic_vector(n-1 downto 0)
);
end practi1b;

architecture rtl of practi1b is
   signal r: remainders;
   signal z: long_remainders;
   signal new_x, new_y: std_logic_vector(new_n-1 downto 0);
begin
   new_x <= n_zeroes&x;
   new_y <= y&m_zeroes;
   r(0) <= new_x;
   iteration: for i in 1 to new_p generate
      z(i) <= (r(i-1)&'0') - ('0'&new_y);
      with z(i)(new_n) select quotient(new_p-i) <= '0' when '1',
         '1' when others;
      with z(i)(new_n) select r(i) <=
         (r(i-1)(new_n-2 downto 0)&'0') when '1',
         z(i)(new_n-1 downto 0) when others ;
   end generate;
   remainder <= r(new_p)(new_n-1 downto m);
end rtl;
```

3 Previous work

3.1 Read and understand the preceding section (2. Dividers). Be sure to understand both the algorithms and the corresponding VHDL models.

3.2 Most synthesis tools will associate an (*m*+*n*)-bit adder to the computation of

```vhdl
   z(i) <= (r(i-1)&'0') - ('0'&new_y);
```

Nevertheless, as *new_y* = $Y.2^m$, that is, a natural whose *m* less-significant bits are 0's, an *n*-bit adder should be sufficient. Thus, modify the model in such a way that an *n*-bit adder will be inferred instead of an (*m*+*n*)-bit one.

3.3 Generate a command file (.do) for inputting stimuli to the simulation model. For example, assuming that

m = 16, n = 10, p = 0 (integer division),

X = 42631, Y = 712,
X = 12999, Y = 1000,
X = 17, Y = 3,
X = 65000, Y = 511.

Compute the correct value of *Q* and *R* for every case.

4 Practical work

4.1 Simulate the circuit with the previously defined (point 3.3) stimuli.
4.2 Simulate using a test bench wave form
4.3 (optional) write and exhaustive VHDL testbench for the divider
4.4 (optional) rewrite the code in order to explicitly describe the circuit of figure 2.

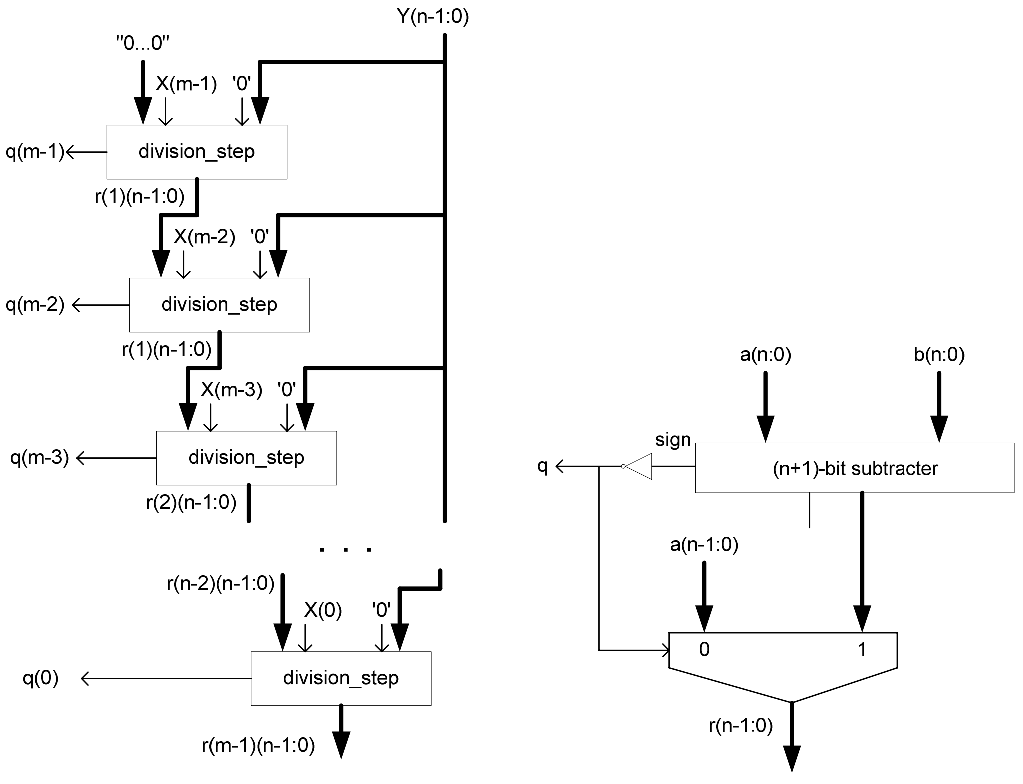

Figure 2. Restoring Divider for natural operands. a. structure. b. basic division step
.

Reference

J.-P.Deschamps, G.Bioul, and G.Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*, Wiley, 2006