FPGA IMPLEMENTATION OF THROUGHPUT INCREASING TECHNIQUES OF THE BINARY DIVIDERS

Bojan Jovanovic
Faculty of Electronic Engineering, University of Nis, Serbia

Milun Jevtic
Faculty of Electronic Engineering, University of Nis, Serbia

Abstract
This paper deals with the binary dividers. A few different binary division algorithms are realized along with well known techniques for throughput increasing. Dividers are described in VHDL hardware description language and implemented in Altera and Xilinx FPGA devices. After classification of the binary division algorithms, radix-2 restoring and radix-2 non-restoring algorithms are described with more details. The techniques for speed increasing (parallelism and pipelining) are applied after. All architectures are finally implemented in FPGA device and their comparison was done from the standpoint of speed and size (percentage of FPGA resources).

Keywords: FPGA, VHDL, binary division algorithms

INTRODUCTION

Division is the most complex of the four basic arithmetic operations. Because hardware solutions are correspondingly larger and more complex than the solutions for other operations, it is best to minimize the number of divisions in any algorithm. On the other hand, division is the common part of many algorithms of great practical value (digital signal and image processing [1], Levinson-Durbin algorithm in LPC speech coder, robot control etc.). Therefore, binary dividers must be treated with careful. Division operands can be signed or unsigned numbers, integer or fractionals, with fixed or floating point. Equations (1) and (2) are basic division equations.

$$x = y*qt + rem \tag{1}$$

$$x = y*qt.\ rem \tag{2}$$

By dividing number *x* with number *y* we get quotient *qt* and division remainder *rem*. Equation (1) remainder is integer number less than *y* while equation (2) remainder is less than 1. If dividend *x* is presented with M bits and divisor *y* with N bits, for quotient *qt* will be necessary M bits while remainder *rem* needs N bits. Figure 1 presents basic division algorithms [3].

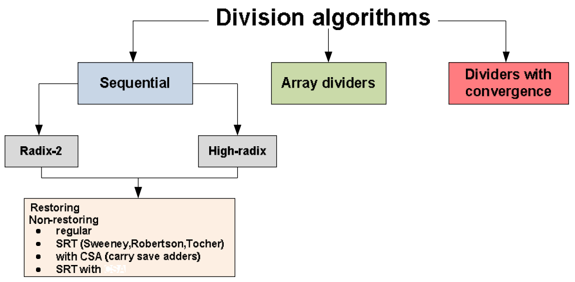

Fig. 1. Basic division algorithms

Radix-2 algorithms generate one quotient bit by division sequence while High-radix algorithms generate more. Here presented dividers use 12 bits wide signed integer operands (in the range [-2048, 2047]) and belong in the group of sequential dividers. Below we consider *radix-2 restoring* and *radix-2 non-restoring* (regular) algorithms.

EXPOSITION

Radix-2 restoring divider

Figure 2 shows simplified block diagram of radix-2 restoring division. Division process can be described as follows:
Put divider *x* in register A, dividend *y* in register B, reset register P. Perform M divide

steps (M is the number of bits for dividend *x* presentation):

- Shift the register pair (P,A) one bit left
- Subtract the content of B from P, put the result back in P
- If the result is negative, set the lsb bit of A to 0, otherwise to 1
- If the result is negative, restore the old value of P by adding the contents of B back in P

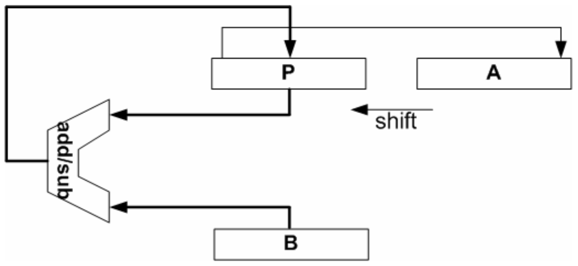

Fig. 2. Radix-2 restoring division

After M steps, the register A will contain the quotient while the content of the register P will be equel to remainder. For implementation of this division algorithm some additional logic for controlling divison process as well as the clock signal for synchronization would be necessary. Detailed logic circuit of sequential radix-2 restoring divider is shown in Figure 3.

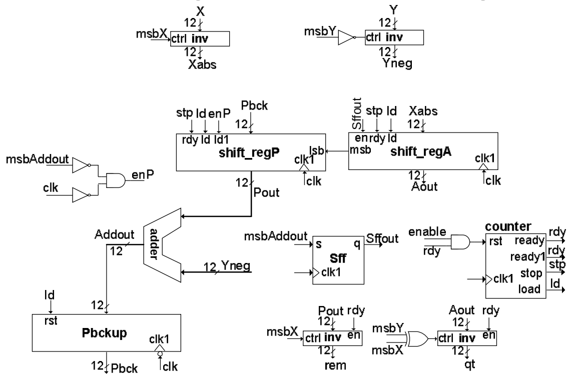

Fig. 3. Detailed logic circuit of radix-2 restoring divider

At the division beginning we get absolute value of dividend *x*, negative value of divisor *y* and reset the counter and Pbckup register. Then we store Xabs in register A and reset register P. On the rising edge of the clock signal clk1 shifting in registers A and P is performed. After shifting, the value of divisor *y* is subtracted from the content of register P (this is done by adding the negative value of divisor – yneg to the content of register P). On the clk1 falling edge the result of addition is stored in register Pbckup. If the result of addition is greater than zero signal enP is activated and the content of register Pbckup is stored into register P. On the RS flip-flop s input we have the *msb* bit from the adder output so that the output of this flip-flop (Sffout) controls register's A *lsb* bit. The counter makes sure that shifting is performed M times (M is the number of bits needed for dividend *x* presentation). At the end of division process quotient and remainder will always be positive (that's because we got the absolute value of dividend *x* as an input in the division process). The real values of the quotient and remainder are on the outputs of inverters with control (at the bottom of the logic circuit from Figure 3). Since the sing of remainder directly depends on dividend sign (if dividend is positive remainder is also positive and vice versa), we get divider *msb* bit on control input of the inverter which determines remainder. Quotient will be negative if dividend and divisor are of opposite sign (one positive another negative – xor operation).

Radix-2 non-restoring division

Radix-2 non-restoring division of dividend *x* (M bits) with divisor *y* (N bits) can be described with the following program sequence [4]:

```
$D := |Y|;\ R_M := X;$
for $j := M - 1$ downto $0$ do
    if $R_{j+1} = 0$ then do
        $Q := [q_{M-1}q_{M-2} \cdots q_{j+1}0 \cdots 0];\ Rem := 0;$
        go to label;
    endif;
    if $R_{j+1} < 0$ then
        $q_j := -1$ else
        $q_j := 1$
    endif;
$R_j := R_{j+1} - q_j \cdot 2^j \cdot D;$
endfor;
$Q := [q_{M-1}q_{M-2} \cdots q0];$
if $X > 0$ and $R_0 < 0$ then
  $Rem := R_0 + D;\ Q := Q - 1;$
    elseif $X < 0$ and $R_0 > 0$ then
    $Rem := R_0 - D;\ Q := Q + 1;$
    else $Rem := R_0;$
endif;
label: if $Y < 0$ then $Qt := -Q$ else $Qt := Q$ endif;
```

With every run through *for* loop one bit of quotient is generated (starting from the *msb* bit). There are maximally M runs through *for* loop. In the case when remainder is equal to zero *for* loop runs less than M times. In the j^th^ loop run the value of R_j is calculated while quotient bit value q_j depends on R_{j+1} value calculated in the previous loop run. Figure 4 shows the logic circuit of radix-2 non-restoring divider.

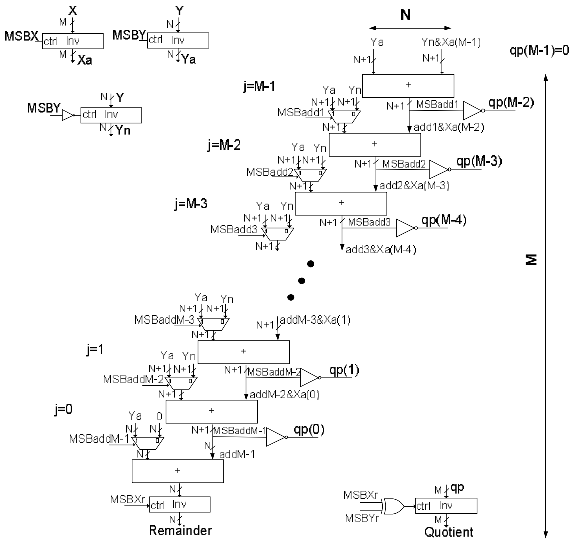

Fig. 4. Detailed logic circuit of radix-2 non-restoring divider

Firstly, absolute values of divider and divisor are determined. With such values division process is started. As a consequence, quotient and remainder will always be the positive numbers. To get quotient and remainder real values two inverters with control input are used (at the bottom of the logic circuit). Radix-2 non-restoring logic circuit comprise of adders and multiplexers array. The number of adders and multiplexers is determined by the number of dividend's bits while its size (the number of input bits) is determined by the number of divisor's bits. The *msb* bit on each adder output determines one quotient bit (first adder output determines quotient *msb* bit) as well as multiplexer select input. Output of the last adder represents division remainder.

Parallelism and pipelining

The throughput of some logic design is determined by its critical path. The critical path is the path between two points in the design which are interconnected. We call it critical because it has the longest signal propagation time, longer than any other design path. If t_{cp} represents signal propagation time of the critical path, maximal design throughput is determined with the following equation:

$$f_{max} \approx 1/t_{cp} \tag{3}$$

Parallelism and *pipelining* are well known techniques for the design throughput increasing [5]. Figure 5 shows some reference design and its parallel and pipelined versions.

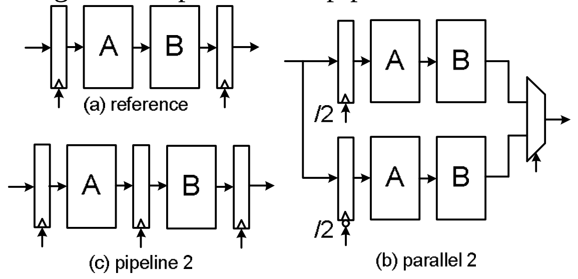

Fig. 5. Micro-architecture example: a) reference design, b)parallel design, c) pipeline design

Parallelism is actually simple design reproduction. The number of design reproduction depends on parallelism stage. Each design reproduction process the part of input data which leads to throughput increasing or design relief (when the throughput is not increased). On the Figure 5(b) half of the input data are processed with one design reproduction. The other half of the input data are processed with other design reproduction. After data processing, the results are alternatively passed to the output using one multiplexer. *Parallelism* does not affect the reduction of the design critical path. Design throughput increasing is the consequence of the fact that design functions run parallel and independently on two (or more) different design reproductions. This is described with equation (4)

$$f_{max} \approx k/t_{cp} \tag{4}$$

wher constant k represents parallelism stage.
Pipelining is the technique which uses pipeline registers on the suitable points in the design to accept the wave of input data which flows to the design output. When data are stored in pipeline registers it is now possible to accept new wave of design input data. This is illustrated on Figure 6.

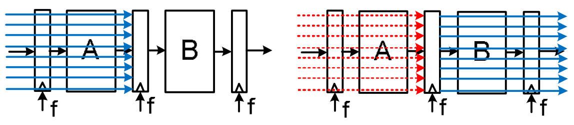

Fig. 6. The illustration of stage 2 pipelining

Pipeline technique can not be applied on every logic design. This technique increases the throughput by reducing the length of the critical path in the design. Pipelining drawback is increased design latency.

Parallelism technique (stage 2) is applied on both previously described division algorithms while pipelining technique (stage 2 and 4) is applied only on radix-2 non-restoring division algorithm. Figure 7 shows the implementation of the stage 2 pipeline technique. Critical paths before and after pipelining implementation are also presented.

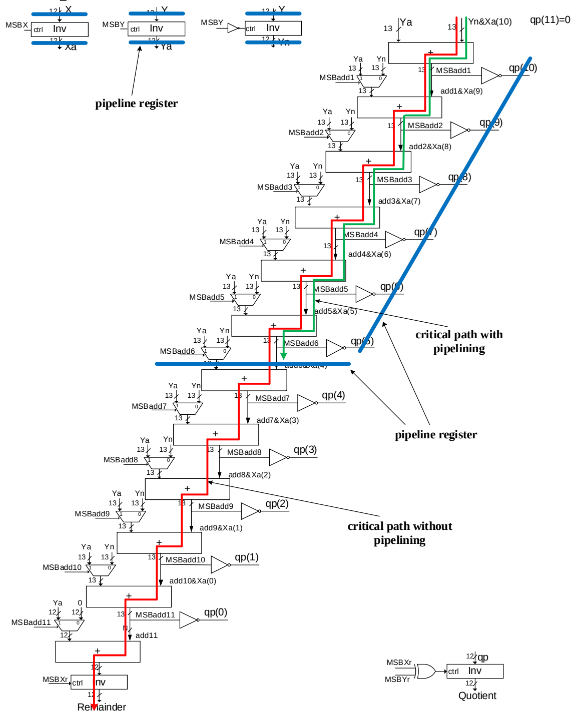
pipeline register
critical path with pipelining
pipeline register
critical path without pipelining
Remainder
Quotient

Fig. 7. Stage 2 pipeline implementation

From the figure 7 can be seen that critical path after pipelining stage 2 implementation is twice shorter. Blue lines on figure 7 represent pipeline registers. Stage 4 pipelining technique is similarly implemented. It consists of more pipelining registers and, as a consequence, the critical path is even shorter.

FPGA implementation

Both previously described division algorithms, along with all throughput increasing techniques are described in VHDL hardware description language. All described designs are implemented in Altera EP2C35F672C6 FPGA device from CycloneII family and in Xilinx XC3S500E FPGA device from Spartan3E family. Tables I and II show the implementation results.

Table I. Altera CycloneII implementation results

Altera Cyclone II	**Total Logic Elements**	**f_{max} [MHz]**
Radix-2 non-restoring		
Reference design	**532**/33216 (2%)	11.89
Parallel 2	**1075**/33216 (3%)	19.53
Pipeline 2	**528**/33216 (2%)	24.1
Pipeline 4	**538**/33216 (2%)	46.05
Radix-2 restoring		
Reference design	**204**/33216 (<1%)	2.3
Parallel 2	**428**/33216 (1%)	4.76
QuartII IP core divider	**250**/33216 (<1%)	29.16

Tables also contain the implementation results of divider IP (Intellectual Property) cores which are specially adapted (from the standpoint of throughput and size) to the FPGA device they implement in. It is naturally to expect that these IP dividers have the best throughput/size ratio.

Table II. Xilinx Spartan3E implementation results

Xilinx Spartan 3E	**N^0 of Occupied Slices**	**f_{max} [MHz]**
Radix-2 non-restoring		
Reference design	**302**/4656 (6%)	18.81
Parallel 2	**521**/4656 (11%)	27.36
Pipeline 2	**425**/4656 (9%)	31.86
Pipeline 4	**447**/4656 (9%)	53.77
Radix-2 restoring		
Reference design	**101**/4656 (2%)	6.27
Parallel 2	**229**/4656 (1%)	12.63
ISE11.1 IP core divider	**291**/4656 (6%)	37.28

Comparing to the reference design (either restoring or non-restoring) the implementation of stage 2 parallelism leads to twice (roughly) increase in the design throughput. The design size is also twice increased so throughput/size ratio is not significantly changed. Implementing pipelining, however, at the cost of slight increase in size we get significant throughput increasing (depending on pipeline stage). Therefore, can be concluded that the pipelining is an effective means to increase the divider throughput, more efficient than parallelism.

CONCLUSION

This paper presents two binary division algorithms. Throughput increasing techniques are also described. *Pipelining* technique has proven to be more efficient because at the cost of slight increase in design size (relative to the reference design) significantly increases its throughput. *Parallelism* technique equally increase both, design size and its throughput. The ratio throughput/size is not significantly changed (relative to the reference design), so parallelism technique is considered less efficient. In the future studies divider power consumption will be considered. It is necessary to examine how throughput increasing techniques impact divider power consumption. With the trend of power consumption reduction it is important to consider the tradeoffs between divider throughput and its power consumption. This is particularly important in ASIC design – custom design based on a library of basic digital circuits of selected technology.

REFERENCE

[1] L. O’Goroman, M. Sammon, and M. Seul, “Practical algorithms for image analysis,” Cambridge University Press, New York, 2008.

[2] J.D. Gordy, R.A. Goubran, “A combined LPC based speech coder and filtered-X LMS algorithm for acoustic echo cancellation,” IEEE Conf. on Acoustic, Speech and Signal Processing, 17-21. May 2004., vol. 4., pp. 125-128.

[3] B. Pahrami, “Computer Arithmetic: Algorithms and Hardware Design,” Oxford University Press, Oxford 1999.

[4] N. Takagi, S. Kadowaki, and K. Takagi, “A Hardware Algorithm for Integer division,” 17th IEEE Symposium on Computer Arithmetic, 27-29. June 2005., pp. 140-146

[5] S.S. Jadhov, “Advanced Computer Arithmetic and Computing,” Technical Publications Pune, 2009.