A Pass-through Systolic Array for Rational Numbers Reduction Extended Abstract

Bogdan Mătăsaru
RISC-Linz, A–4040 Linz, Austria
email: bmatasar@risc.uni-linz.ac.at

Presented at the 2^{nd} International Workshop on
SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTATION
Timişoara, October 4–6, 2000

Abstract. We design a pass-through (on-line) systolic array for reducing rational numbers based on the extended binary algorithm. The numbers are pipelined through the array, the least significant bit first. The length of the array is fixed; longer numbers have to pass several times through the array.

AMS Subject Classification:

Keywords and phrases: systolic, arbitrary precision arithmetic, rational numbers.

1 INTRODUCTION

Cryptography, computer algebra system, computer aided engineering – all have something in common: they heavily rely on arbitrary precision (or exact) arithmetic. Although there were attempts to implement the algorithms in hardware, most of the applications use only software implementations. Considering the impetuous achievements in the field-programmable (FP) devices, we aim to designing a coprocessor for multiple precision arithmetic. The ease of designing the circuits and the possibility to configure them after leaving the factory make the FP devices a good candidate for implementing the coprocessor.

In the currently available solutions (FPGA, ASIC) the host and the coprocessor communicate by streaming the operands and the result. Thus, the coprocessor is fed with the operands and the host retrieves the results in a pipelined manner. The implementation of the algorithms as pass-through systolic arrays appears as a natural solution: not only the 3 phases of a call (operands feeding, the algorithm itself and retrieving the result) can overlap, but also the host can issue a call before the precedent one has finished.

In this paper we design a pass-through systolic array used for reducing rational numbers to the canonical form. The starting point for the design is the pass-through array developed by Jebelean for computing the greatest common divisor of two multiple precision integers ([3]). The key idea of the new design is to compute a pair of cofactors in parallel with the GCD. When the GCD is produced, the two cofactors represent the canonical form of the rational number. The current approach is to compute the GCD with the binary algorithm developed by Brent and Kung ([1]) followed by two exact divisions ([2]) of the operands by the GCD. It requires less operations, but the only way to speed it up is to execute the two exact divisions in parallel. The extension we proposed can update the cofactors in parallel with the execution of the GCD, thus the overall speed is the same as that of computing the GCD. Even so, tests made with simple precision operands (32 bits) show a speedup over the classical sequential algorithm on some machines.

2 THE ALGORITHM

The full description and the correctness proofs can be found in [4]. In the following section I will briefly present the main idea of the algorithm.

The input of the extended binary (plus-minus) GCD algorithm consists of two n-bit operands a and b and four cofactors u, v, t and w. The operations executed on the operands and cofactors are additions, subtractions and shifts and they are decided by inspecting the least significant two bits of each operand.

Let us denote by a_k, respectively b_k, the values of the operands at step k. Moreover, $x[1], x[0]$ represent the least significant bits of an operand x. The extended binary GCD algorithm computes also the sequences of cofactors of u_k, v_k, t_k, and w_k such that at each step k:

$$u_k \cdot a + v_k \cdot b = a_k \cdot 2^k, \quad t_k \cdot a + w_k \cdot b = b_k \cdot 2^k. \tag{1}$$

The algorithm ends when $b_k = 0$. Then, a_k equals the GCD and $t_k \cdot a + w_k \cdot b = 0$, hence $a/b = -w_k/t_k$. For the proofs that $-w_k/t_k$ is the canonical form of the initial fraction and a more detailed description of the algorithm see [4].

The sequence of cofactors is defined recursively starting from the initial values: $u_0 = 1, v_0 = 0, t_0 = 0, w_0 = 1$.

At each iteration one of the following operations will take place:

shift both: $(a_k[0] = 0,\, b_k[0] = 0)$:
$a_{k+1} := a_k/2,\ b_{k+1} := b_k/2$, the cofactors remain unchanged.

interchange and shift: $(a_k[0] = 0,\, b_k[0] = 1)$:
$a_{k+1} = b_k,\ b_{k+1} = a_k/2,$
$u_{k+1} := 2 \cdot t_k,\ v_{k+1} := 2 \cdot w_k,\ t_{k+1} := u_k,\ w_{k+1:} = v_k.$

shift: $(a_k[0] = 1,\, b_k[0] = 0)$:
$a_{k+1} = a_k,\ b_{k+1} = b_k/2,$
$u_{k+1} := 2 \cdot u_k,\ v_{k+1} := 2 \cdot v_k,\ t_{k+1} := t_k,\ w_{k+1} := w_k.$

plus-minus: $(a_k[0] = 1,\, b_k[0] = 1)$, **plus** if $a_k[1] \neq b_k[1]$, otherwise **minus**:
$a_{k+1} = b_k,\ b_{k+1} = (a_k \pm b_k)/2,$
$u_{k+1} := 2 \cdot t_k,\ v_{k+1} := 2 \cdot w_k\ t_{k+1} := u_k \pm t_k,\ w_{k+1} := v_k \pm w_k,$

3 THE SYSTOLIC ARRAY

The cells of the array are auto-configurable. This feature is implemented using the ICF/PCF circuits described in [3] and shown here in the figure 1.

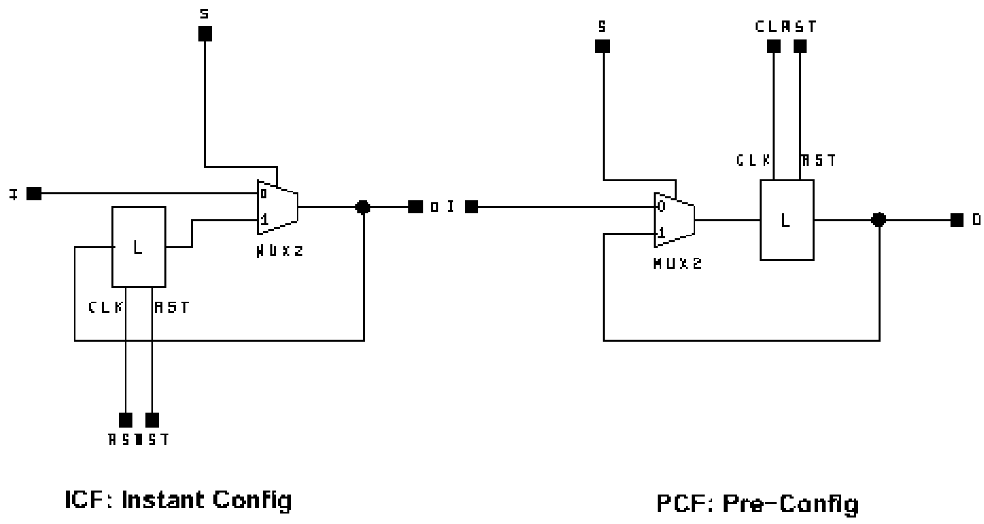

Figure 1: The configuration units

PCF is used when the data used for computation (I) arrives one clock before the data used for configuration (S), whereas ICF is used when they arrive simultaneously.

One can easily verify that the operation **shift both** can occur 0 or more times only in the beginning of the algorithm, followed by at most one **interchange and shift**. Then, there can be only **shift** or **plus-minus** steps.

This leads to a heterogeneous array:

- One `feeder` cell
- Several `processor` cells
- One `retriever` cell

3.1 THE FEEDER

The `feeder` handles the **shift both** and **interchange and shift** steps.

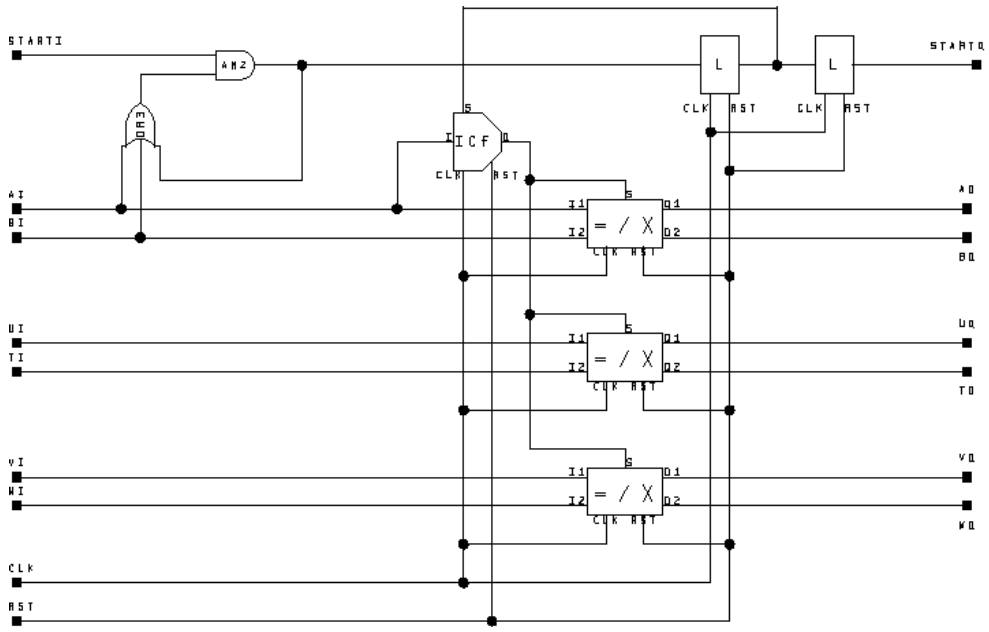

Figure 2: The `feeder`

The figure 2 presents the design of this cell. The operands and the cofactors are passed to the `feeder` one bit per clock cycle, least significant bit first. One more signal, $START$ remains high during the data-transfer.

As long as both signals A and B corresponding to the bits of a and b are low, the output $START$ signal is kept low so that the least significant null bits are ignored. In the same time the cofactors are delayed. Actually one can assume the striping of the least significant null bits can be done by the host, thus leading to a simpler design of the `feeder`.

When the first high arrives on either of A or B, the feeder is eventually configured to execute a **interchange and shift** and the signals are passed to the next cell.

3.2 THE PROCESSOR

After passing through the feeder, the signals pass through a chain of several `processors`. Each `processor` will apply either a **shift**, **plus** or **minus** on the operands and the cofactors.

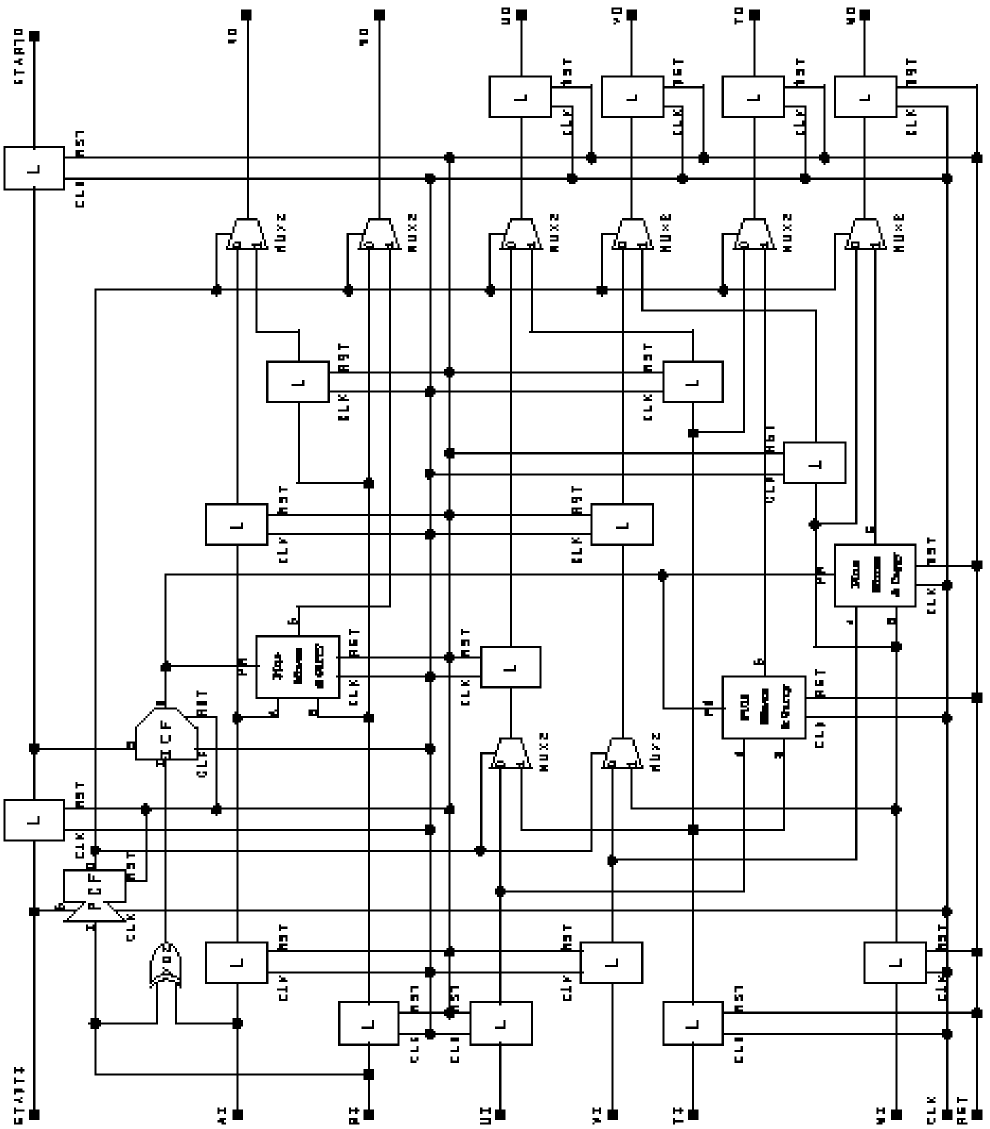

Figure 3: The `processor`

When $START$ becomes high, **PCF** will be configured so that the cell will execute a **shift** if B is low or a **plus-minus** otherwise. The **shift** is executed by delaying A, T, W one clock cycle, and U, V two cycles.

If B is high, on the next clock cycle **ICF** will be configured so that the `processor` will execute a **plus** or a **minus** operation depending whether the signals A and B are different or not.

3.3 THE RETRIEVER

Finally, the `retriever` will synchronize the outputs and adjust or compute the signs of the cofactors. The results are extracted from the `retriever` in the same LSF manner.

In the figure 4 we have a circuit that just delays all the signals but $START$. If on the first low $START$ signal one of the operands or cofactors is high, the value retrieved by the host is a negative number in 2's complement representation, otherwise it is positive.

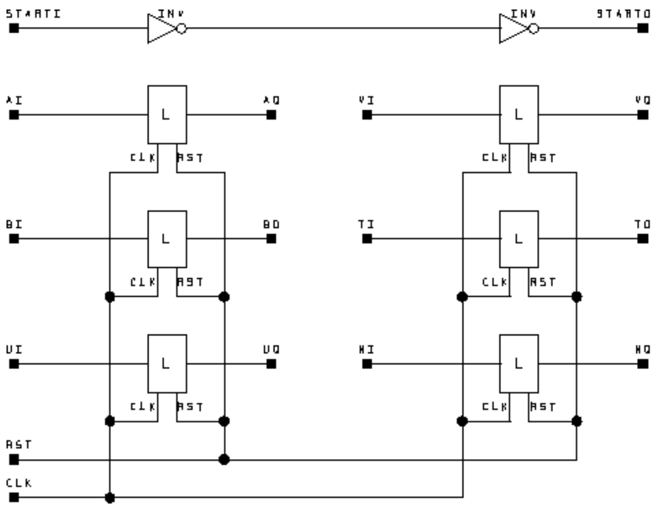

Figure 4: The `retriever`

4 EXPERIMENTS AND CONCLUSIONS

We implemented the array on an Atmel 40K10 chip. With 10 `processors` it can handle 8 bit operands in one pass. It uses 351 logical gates, that is 72.8% of the chip, and it can work at a frequency of 28 MHz. A previous off-line design ([5]), which considers that the operands are at the beginning of the computation already into the array, consumes 83.4% of the chip and works at a frequency of 25 MHz. Moreover, the off-line design was not scalable, because the maximum size of the operands was restricted by the number of cells in the array. In the new version, longer operands require multiple passes through the array.

References

[1] R. P. Brent and H. T. Kung, *A systolic algorithm for integer GCD computation*, Symp. on Computer Arithmetic (K. Hwang, ed.), IEEE Computer Society, 1985.

[2] T. Jebelean, *An algorithm for exact division*, Journal of Symbolic Computation **15** (1993), no. 2, 169–180.

[3] ______, *Auto-Configurable Array for GCD Computation*, Tech. Report 97–12, RISC–Linz, 1997, `http://www.risc.uni-linz.ac.at`.

[4] B. Mătăsaru, *An extension of the binary GCD algorithm for systolic parallelization*, Tech. Report 99–47, RISC–Linz, 1999, `http://www.risc.uni-linz.ac.at`.

[5] B. Mătăsaru and T. Jebelean, , Field-Programmable Logic and Applications, Villach, Austria, August (R. W. Hartenstein and H. Grünbacher, eds.), Springer Verlag LNCS 1896, 2000, pp. 810–813.