Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA

Jiří Buček, Pavel Kubalík, Róbert Lórencz, and Tomáš Zahradnický
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic
Email: { bucekj | xkubalik | lorencz | zahradt }@fit.cvut.cz

***Abstract*— The residual processor is a dedicated hardware for solving sets of linear congruences. It is a part of the modular system for solving sets of linear equations without rounding errors using Residue Number System. We present a new FPGA implementation of the residual processor, focusing mainly on the memory unit that forms a bottleneck of the calculation, and therefore determines the effectivity of the system. FPGA has been chosen, as it allows us to optimally implement the designed architecture depending on the size of the problem. The proposed memory architecture of the modular system is implemented using the internal FPGA block RAM. Our goal is to determine the maximum matrix dimension fitting directly into the FPGA, and achieved speed as a function of the dimension. Experimental results are obtained for the Xilinx Virtex 6 family.**

I. Introduction

Residue Number System (RNS), despite being known for a long time, is becoming a hardware attractive arithmetic today, not only because it permits us to represent long integers as independent combinations of small integers based on the Chinese Remainder Theorem, but also because it requires a simple arithmetic unit. These properties offer natural parallelism, lead to simpler hardware, and reduce chip size when compared to a traditional floating point unit implemented in hardware.

RNS is used in areas of digital image processing [1][2], digital signal processing [3][4][5], and in public-key [6] and elliptic curve [7][8] cryptography. RNS is also used to simulate multiple precision arithmetic and for error-free solution of linear systems [9][10]. Error-free solution of linear systems is often needed in case of large, dense and ill-conditioned systems, where rounding errors can lead to long run times due to stability problems, or even hinder the solution completely.

Performing error-free solution of linear systems on regular CPUs has large time (and area) complexity. The CPU architecture is usually not optimized for the algorithms and operations needed (parallelism with respect to multiple modules, modular arithmetic operations etc).

Papers [11] and [12] design a hardware RNS linear equation solver — Modular System (MS) — whose implementation was very difficult at that time. With current technologies, it is possible to implement the system, and especially FPGA technologies offer a straightforward implementation with reconfiguration possibilities based on the cardinality of the problem and optimize for time and area.

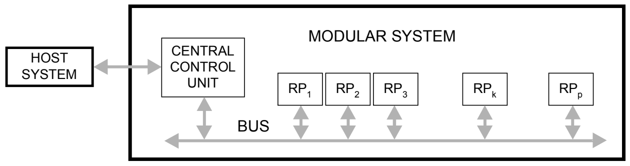

Fig. 1. Architecture of the Modular System [11]

However, for an efficient FPGA implementation, several parts of the system must be redesigned to use resources found in modern FPGAs. This is true especially for the memory architecture, which in [12] used asynchronous logic and custom memory elements. We present a new memory architecture using standard RAM blocks found in most recent FPGAs. We also redesigned the addressing and pivoting logic to support efficient implementation of the elimination algorithm used.

After a brief introduction of the architecture of the MS for solution of sets of linear equations (SLE)s $\mathbf{Ax} = \mathbf{b}$, the paper focuses on the memory architecture of the residual processors (RP)s inside the MS. Next, there follow FPGA implementation results for various problem sizes, their analyses, and evaluations. Finally, the paper is concluded with the properties of the FPGA residual processor implementation.

II. Architecture of the Modular System

Paper [12] describes the method, the algorithm, and the corresponding parallel hardware architecture of the MS (Fig. 1).

It should be noted that evaluation in each modulus is performed independently of the others and that the addition is carry-free, subtraction borrow-free across the individual moduli, and therefore the computation can occur safely in parallel. Once the computation is done, the result is transformed back into the rational number set either with the Chinese Remainder Theorem or the Mixed Radix Conversion.

The error-free solution of an SLE with operations performed in residue arithmetic is implemented in this special MS. The MS typically has a parallel SIMD architecture, and consists of a control unit and several processing units – (RP)s denoted as RP$_1$, RP$_2$, …, RP$_p$ interconnected with a BUS (see Fig. 1).

III. Residual Processor Architecture

The architecture of the residual processor RP$_k$ is depicted in Fig. 2 consisting of Memory, Arithmetic Units AU$_2$, AU$_3$, …, AU$_{n+2}$ and the Control unit.

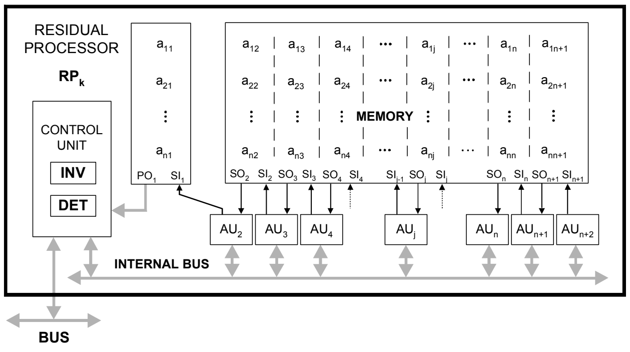

Fig. 2. Architecture of the Residual Processor [11]

The memory contains residues of matrix $\mathbf{A}$ and vector $\mathbf{x}$ elements. The storage of values of a row of the matrix from AU registers is performed bitwise via Serial Inputs SI_1, SI_2, $\ldots$, SI_{n+1}. Loading of values of rows of Memory to AUs is done via Serial Outputs SO_2, SO_3, $\ldots$, SO_{n+1}. The bits of element values of the first matrix column are read by the Control unit via the parallel bus PO_1. All AUs and the Control unit are interconnected via Internal Data Buses $\mathrm{IDB}_{\mathrm{in}}$ and $\mathrm{IDB}_{\mathrm{out}}$. The above RP_k architecture can solve systems of linear congruences (SLC)s $\mathbf{A}\mathbf{x}_k \equiv \mathbf{b} \pmod{m_k}$. RPs together with the Control unit of the MS also support all conversion operations from integer to RNS and vice versa. The INV and DET units compute the modular multiplicative inverse and the determinant of $\mathbf{A}$, respectively.

All SLCs in MS are solved with the Gauss-Jordan elimination with Rutishauser modification [12] (GJR), which is especially suitable for hardware implementation. The elimination process in RNS is specific in a way that it has to perform a so called "nonzero residue pivoting" that was introduced in [11]. Pivoting and massive data access constitute a bottleneck, and therefore the memory architecture design is critical and is dealt with in the next section.

The Rutishauser modification of Gauss-Jordan elimination implies that the column data is shifted by one column to the left during each elimination step. The shift is accomplished by the AUs and the memory interconnection design in Fig. 2. In addition, the first column of the SLC matrix contains values of the elements intensively used during the elimination process and for this reason the output from the first column needs to be parallel (these values are used in the INV and DET units). The values in the first column determine the first multiplication operand in the entire row being processed, both in pivot elimination and row reduction. The other columns a_{i2} to a_{in+1} inclusive are used as the second multiplication operand, and also for addition operations. Assuming serial-parallel (shift-add) multiplication, we need to read individual bits of these values, thus requiring serial access only.

A. New Memory Architecture and Pivoting

The elimination process requires nonzero residue pivoting. The pivot column is always the first column of the matrix, and all nonzero values are equally acceptable as pivots. Search for a pivot is done sequentially; however, this search can be easily

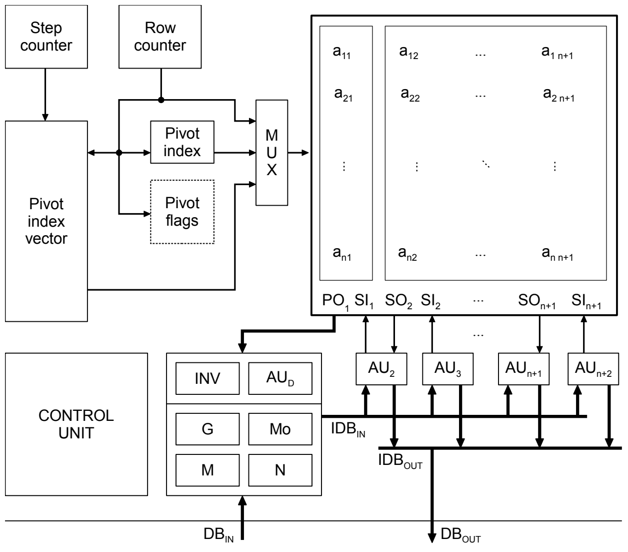

Fig. 3. New architecture of the Residual Processor including new memory architecture

performed concurrently with write operations to the memory. The search is performed while the matrix is loaded or updated during computation. In most cases, the pivot is passed to the inversion unit (INV) long before the inverse is needed. In order for a value to be accepted as a pivot, i) it must be nonzero, ii) the row has not contained a pivot yet, and iii) no pivot has yet been found for this elimination step.

Once the pivot is found, its row index must be stored in a pivot index vector at the address of the current elimination step. The pivot row must be flagged in order to skip it during the pivot search performed in subsequent elimination steps. If no pivot was found, the matrix is singular in this modulus.

The elimination is performed by rows. The architecture must support addressing of the pivot row first; then sequentially reduce memory matrix rows, with an exception of the pivot row which must be skipped. The first value in each row must be read in parallel. This value is either the pivot, which is inverted, or a value from a different row, which is negated.

The remaining values in each row are read bit-serial (but all values concurrently) from the MSb first. This ensures the correct order for left-shift modular multiplication and addition, and follows from the design depicted at Fig. 3.

Upon completion of the elimination process, the solution vector appears in the first column. The order of its elements corresponds to the pivot row indices and may need to be reordered. The result is therefore read out in correct order by addressing through the pivot index vector.

Algorithm 1. Elimination algorithm including the pivoting instructions from Table I. Parameters: n is matrix dimension, e is word length.

```
1. $k = 0$, assuming pivot is found during loading of matrix
2. **while** $k < n$ **begin**
3.     PSI
4.     GETD
```

TABLE I

PIVOTING INSTRUCTIONS USED IN ALGORITHM 1.

Instruction	Description
PSI	Select pivot row
GETD	Read data row from memory
PUT	Write data to memory and test for pivot
PTS	Test and skip pivot row during reduction
PCRR	Reset row counter

```
 5.    multiplication of the pivot row with pivot$^{-1}$
       ($5e$ clocks)
 6.    PUT
 7.    PCRR
 8.    **repeat** $n-1$ **times begin**
 9.      PTS
10.      GETD
11.      reduction of other matrix rows using adjusted
         pivoting row ($5e$ clocks)
12.      PUT
13.    **end repeat**
14. **end while**
```

B. Arithmetic Units, Modular Inverse and Controller

The Arithmetic Units AU_2, AU_3, $\ldots$, AU_{n+2} and AU_{D} design closely follows the original design in [12]. The modular inversion unit INV was originally designed as a look-up table. However, for larger moduli, this table grows too large, and therefore we use a new inversion unit computing modular inverse with the left-shift modular inverse algorithm [13].

The controller contains a finite state machine using a memory-based transition and output functions. This allows flexibility with regard to modification and future extensions.

C. FPGA Implementation

The memory architecture as a critical part of the RP can be divided into two parts: the pivoting unit and common memories. The pivoting unit is always implemented in FPGA. Memory can be implemented internally using block RAM components, or externally e.g. by a DDR SDRAM. The main implementation differences are in their parameters such as memory capacity, throughput, and latency. On one hand, the internal implementation with FPGA memory has small capacity and low latency, while on the other hand the external memory provides large capacity but also a high latency.

We design an architecture with the internal memory. We can estimate the size of the largest matrix with respect to maximum size of the block RAM given by the FPGA chip type. Nonetheless, the maximum frequency of the implemented design cannot be easily estimated or calculated. Our tested memory architecture consists of two parts: i) the internal memory, and ii) a pivoting control logic to support addressing during the calculation in the RP. The design of our memory architecture is shown in Fig. 3.

The memory matrix consists of the first column a_{i1} and the remaining columns a_{i2} to $a_{i,n+1}$. All columns share a common address. During pivot search, the address is taken from the Row counter and if the pivot is found in the current row, the address is written into the Pivot index vector at the address of current elimination step, and the Pivot flag for the address of the current row is set. At the same time, the pivot address is stored in the Pivot index register for comparison during the next elimination step.

IV. EXPERIMENTAL RESULTS

All experiments were conducted on the residual processor architecture consisting of: data memory, Pivot index, Pivot flags, counters, arithmetic units, inversion unit, and control units. The tools used for simulation and implementation were selected with respect to the hardware programming language. The design was written in VHDL. The maximum matrix dimension n and the word length e and are configurable at synthesis time using generics. The actual matrix dimension, value of the modulus and matrix data are set at runtime.

The design was simulated, synthesized and implemented (mapped, placed and routed). The experiments were performed on one residual processor (single modulus), including the input data modulo reduction and matrix elimination. (Transformation into the rational numbers was not performed).

To verify the design, we added test units increasing testability and observability of the simulated design to verify the calculation. The test data were generated using Wolfram Mathematica and converted with a Python script. The simulation of the residual processor architecture ran within Mentor ModelSim. The simulation results and Mathematica results were compared by another script. The simulation was done for matrices up to $n = 100$, greater matrices were not simulated due to a high simulation time.

The implementation process started after simulation, when correctness matrix calculation was verified. We tested several different matrix dimensions, and always set the word length to 24 bits. The block RAM modules were inferred from a functional description by the synthesis tool. Each column (a_{i1} to $a_{i,n+1}$) is implemented as a block RAM module. The memory is not always used effectively, depending on the number of rows. Each memory module has the full capacity given by the width of its address bus, that is a power of two. Therefore the memory utilization increases by a step when the number of address bits changes. The number of arithmetic units was automatically generated by the selected matrix dimension.

The Xilinx FPGA platform was selected for all tests. We used Xilinx ISE to synthesize and implement our design to the FPGA. We selected the FPGA with the highest block RAM memory capacity in the Virtex 6 family, that is, the xc6vsx475t-2-ff1156. In order to get a good estimate of the best achievable timing, we set the "High effort" with "Continue on Impossible" options for the implementation part. Constraining the timing would achieve even better timing. We gathered the minimum period, logic and memory utilization from the implementation (post place and route) report files. The results of our experiments are shown in Table II.

TABLE II
IMPLEMENTATION RESULTS FOR THE FPGA RESIDUAL PROCESSOR ARCHITECTURE (FPGA IS XILINX XC6VSX475T).

	Area utilization				Time		
n	slices	%slices	BRAM	%BRAM	T_P[ns]	f_{clk}[MHz]	T_{elim} [ms]
100	4223	6%	101	10%	6	166	7.2
300	12194	16%	301	28%	7	143	74.8
500	19277	26%	501	47%	9	111	266.6
700	29394	40%	701	66%	11.4	88	658.2
900	38372	52%	901	85%	12.7	78	1216.6
1000	42368	57%	1001	94%	13	77	1537.1

The n column denotes the matrix dimension. The "slices" and "BRAM" columns are the number of used slices and used block RAMs also with the percentage of occupied FPGA resources (xc6vsx475t). The "T_P" and "f_{clk}" columns are minimum clock periods and maximum operation frequencies. The "T_{elim}" column shows the time in milliseconds to solve a set of linear congruences for one modulus depending on minimum clock period for the selected matrix dimension.

The elimination time (T_{elim}) assumes the data are already loaded and stored in memory and elimination process is in run. The load part takes only a small part of all time needed for solution of a set of linear congruences. For example, for matrix dimension $n = 100$, the load process takes only 12.7%, while for $n = 1000$ it takes only 11.7%.

The results show that our residual processor architecture allows for a maximum matrix size of approx. 1000 rows by 1001 columns with a word size of 24 bits in the chosen FPGA type. Even with the maximum tested matrix dimension of 1000, which uses more than 90% of the available block RAM, only approx. 60% of all available slices in FPGA are used.

The clock period increases with the increasing matrix dimension. Static time analysis shows that the main parts of the delay in the circuit are in addressing, control and inner data bus signals. The fanout of signals significantly increase when size of matrix increases. For comparison, on a CPU, solving a SLC of dimension 100, 500 and 1000 takes approximately 3 ms, 424 ms, and 3.37 s, respectively (Intel T9400 CPU at 2.53GHz, cache size 6144 KB, C language compiled with GCC). This shows that for $n = 1000$, our design is approx. 2 times faster. In case of future ASIC implementation of our design, we can expect even greater speedup (which is difficult to predict, but can reach 100 or more).

Further work will focus on the design of an external memory interface, which will be influenced by the limited FPGA input/output pins. The acquired results will be used to design the whole system for solving SLEs.

V. CONCLUSION

We have designed a Residual Processor (RP) architecture which solves a set of linear congruences. RP was designed with a focus on its effective implementation in FPGA for various problem sizes with a special attention to the memory architecture. The memory design is critical to the RP because of massive data access and pivoting. RPs are portions of a Modular System for solving sets of linear equations in RNS.

All important parts of the RP architecture, such as data memory, Pivot index, Pivot flags, counters, arithmetic units, inversion unit and control unit were implemented and tested in a Xilinx Virtex 6 FPGA with the largest RAM size. The results show that our RP architecture allows for a maximum matrix size of approx. 1000 rows by 1001 columns with a word size of 24 bits in the chosen FPGA type. The maximum tested matrix dimension of 1000 uses more than 90% of the available block RAM and approximately 60% of all available slices in the FPGA while being approx. 2 times faster than a CPU software implementation.

Future work will focus on a new RP architecture with external memory and limited numbers of AUs. Also bitwise communication between internal memory and AUs will be studied. Next, we will implement the RP in ASIC and evaluate its performance.

ACKNOWLEDGMENT

This research was supported by the Czech Science Foundation project no. P103/12/2377.

REFERENCES

[1] D. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, oct. 2010, pp. 775–780.

[2] T. Toivonen and J. Heikkila, "Video filtering with fermat number theoretic transforms using residue number system," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 16, no. 1, pp. 92–101, jan. 2006.

[3] G. Cardarilli, A. Nannarelli, and M. Re, "Residue number system for low-power DSP applications," in *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, nov. 2007, pp. 1412–1416.

[4] R. Chaves and L. Sousa, "RDSP: a RISC DSP based on residue number system," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, sept. 2003, pp. 128 – 135.

[5] A. Mirshekari and M. Mosleh, "Hardware implementation of a fast FIR filter with residue number system," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, may 2010, pp. 312 –315.

[6] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *Computers, IEEE Transactions on*, vol. 53, no. 6, pp. 769 –774, june 2004.

[7] D. Schinianakis, A. Kakarountas, and T. Stouraitis, "A new approach to elliptic curve cryptography: an RNS architecture," in *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, may 2006, pp. 1241–1245.

[8] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Proceedings of the 10th international workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 62–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3-5

[9] R. T. Gregory and E. V. Krishnamurthy, *Methods and Application of Error-free Computation*. Springer Verlag, 1984.

[10] D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics*, Addison-Wesley Series in Mathematics ed. Addison-Wesley Publishing Company, Inc., 1973, vol. 2.

[11] R. Lórencz and M. Morháč, "A modular system for solving linear equations exactly, i. architecture and numerical algorithms," *Computers and Artificial Intelligence*, vol. 11, no. 4, pp. 351–361, 1992.

[12] M. Morháč and R. Lórencz, "A modular system for solving linear equations exactly, ii. hardware realization," *Computers and Artificial Intelligence*, vol. 11, no. 5, pp. 497–507, 1992.

[13] R. Lórencz, "New algorithm for classical modular inverse," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. London, UK, UK: Springer-Verlag, 2003, pp. 57–70.