ArPALib: A Big Number Arithmetic Library for Hardware and Software Implementations. A Case Study for the Miller-Rabin Primality Test

Jan Macheta, Agnieszka Dąbrowska-Boruch, Paweł Russek(✉), and Kazimierz Wiatr

AGH University of Science and Technology, Mickiewicza Av. 30, 30-059 Krakow, Poland
{macheta,adabrow,russek,wiatr}@agh.edu.pl

Abstract. In this paper, we present the Arbitrary Precision Arithmetic Library - ArPALib, suitable for algorithms that require integer data representation with an arbitrary bit-width (up to 4096-bit in this study). The unique feature of the library is suitability to be synthesized by HLS (High Level Synthesis) tools, while maintaining full compatibility with C99 standard. To validate the applicability of ArPALib for the FPGA-enhanced SoCs, the Miller-Rabin primality test algorithm is considered as a case study. Also, we provide the performance analysis of our library in the software and hardware applications. The presented results show the speedup of 1.5 of the hardware co-processor over its software counterpart when ApPALib is used.

Keywords: Big numbers · Primality tests · High-Level Synthesis · FPGA

1 Motivation

The big numbers - the integer numbers in computer data representation that comprise of hundreds of bits, are a foundation for security solutions of today's computer systems. Although, some of the modern programming languages allow programmers to choose an arbitrary variable size, the majority of modern C language compilers, supports the maximum integer size of 64-bits only. Therefore, it is necessary to define the custom big data types and create functions for arithmetic operations from scratch. Alternatively, a developer can benefit from one of the ready-to-use big number libraries that are offered either commercially or as open sources [1,2].

The FPGAs have been used for security enhancement algorithms before [3–6]. Meanwhile, the role of the Programmable SoC (PSoC) solutions, that incorporate a CPU subsystem and an FPGA structure in a single chip, is rapidly growing. Soon, such CPU-FPGA hybrid solutions will become ubiquitous, not

DOI: 10.1007/978-3-319-56258-2_28

only for embedded systems but in server solutions as well. The major obstacle in deploying such systems is a cost of hardware development. Designing of hardware is time-consuming, cost-intensive, and requires extra developer skills. However, a shift towards high-level programming can be noticed in design tools today. Thanks to the High-Level Synthesis, C and C++ codes of the algorithms can be translated to their Register Transfer Level (RTL) representations and the process can be controlled by means of inserting pragmas into the source files. The HLS tools significantly speed up the FPGA system development and lead the way to CPU-FPGA systems spreading. Unfortunately, some of the software techniques (*i.e.* dynamic allocation or recursion) are prohibited by HLS. Thus, preparing the hardware synthesizable C code still requires some effort and care.

The main goal of this paper is to introduce the Arbitrary Precision Arithmetic Library (ArPALib) that was developed by the authors. Its main advantage over other available arithmetic libraries is that it can be implemented both in software and HLS synthesized hardware, allowing the developer to swiftly create CPU-FPGA based solutions. It is worth highlighting that the source code of ArPALib is available online in the repository provided by the authors [7].

2 The Big Number Libraries

For a proper background we will first overview the three big number libraries that are available for C programmers. We will refer to the GMP [1] and BigDigits [2] libraries, *ap_cint.h* - built-in library from Xilinx's Vivado HLS.

GMP library is considered to be one of the fastest big number library available today. It covers an arbitrary bit-width for signed and unsigned integers and fixed-point numbers. Its extraordinary performance comes from a variety of implemented algorithms that are selected according to the actual size of the used numbers. Additionally, the GMP's algorithms exploit aggressive optimizations for selected processor architectures (*e.g.* AMD K5/K10, Intel Sandy Bridge, ARM family).

The individual number is represented by the mpz_t structure that comprises the memory pointer to a dynamically-allocated array that stores the value of the number, and its current size. That kind of representation reduces the memory read/write operations, and induces basic pointer arithmetic to perform calculations. Unfortunately, the mentioned coding techniques exclude using GMP from a HLS design flow.

BigDigits library is an open source arithmetic library that conforms to ANSI C standard. The authors of BigDigit implemented mainly paper-and-pencil methods arithmetic algorithms, where arguments must be the same lengths. If the allocated space is bigger than the actual number length, zero-padding operation is performed to ensure the result correctness. BigDigits simplicity made this library a good candidate for HLS, however, some of its algorithms use a recursion, which is not supported by HLS tools.

In our experiments we used Xilinx's Vivado HLS environment, that provides built-in arbitrary precision integer library, included in ***ap_cint.h*** header.

It defines the [u]int <*precision*> type exclusively recognized by the Vivado's C compiler and HLS tool, that makes bit-accurate simulations possible.

The [u]int <*precision*> is implemented in hardware as a bit vector, which means that the width of hardware registers and functional elements fit the width of the number. That provides a one clock cycle for a single operation, but also leads to a limited clock frequency and the fast exhaustion of FPGA resources for large bit-widths (even a single multiplication of two 4096-bit numbers exceeds the capacity of Xilinx Artix-7 family of FPGAs). Therefore, the maximum precision of [u]int is restricted (by default) to 1024. Unfortunately, the Vivado HLS does not provide sequential processing of big numbers and the trade-off between performance and resources cannot be controlled.

3 ArPALib Introduction

To overcome problems mentioned in Sect. 2, we created ArPALib, which is fully synthesizable (by Vivado HLS 2015.4) and C99 compatible library for software and hardware implementations. Our goal was to propose a solution that enables sequential processing of big numbers in blocks of bits of a selected width to reduce. The code of ArPALib is publicly available under the GNU GPLv3 license [7].

The library can be parametrized to redefine the base integer type (named uint_t), which is used as an elementary computational block of the big number, and processed in sequential algorithm steps. The base type allows programmers to force such a bit-width of the co-processor architecture that fits the size of the selected FPGA device. Furthermore, it can be defined as [u]int for the Vivado HLS compiler, thus enabling optimization for speed or resources footprint.

A type for the unsigned integer big number is called uintBig_t in ArPALib. It contains an array of uint_t elements, and the length of the array is defined at compile time. Optionally, the uintBig_t can hold a variable that keeps the current size of the stored big number. Thanks to that, the number of read/write operations is reduced by excluding not used segments from computations, instead of zero padding operation. For example, the number of memory accesses is limited to the size of the smaller argument in the add operation. The bigger the difference of the arguments' length is, the more significant is the speed-up. Our approach requires tracking the arguments' size, so it introduces some extra operations. However, even if the arguments are of similar size, the overhead that ArPALib produces is small (*e.g.* only one comparison operation more for the addition than the algorithm without modification). The library supports dynamic data allocation for software implementations to prevent stack overflow problems.

Algorithms implemented in ArPALib are summarized in Table 1. The library implements all elementary binary operations, comparison and assignment operations. It also provides input/output tools that include conversion of binary strings of different formats to big number values and vice versa in the software version. Unfortunately, all the performed operations are integer-based only, therefore, the Schoenhage-Strassen algorithm or Barett reduction are not available. On the other hand, hardware implementations are modest thanks to that.

Table 1. Algorithms implemented in ArPALib

Addition	• Schoolbook addition with carry algorithm
Subtraction	• Schoolbook subtraction with borrow algorithm
Multiplication	• Karatsuba alg. (for the uint_t type)
	• Schoolbook long multiplication algorithm
Division	• Knuth's D algorithm ref
Exponentiation	• Exponentiation by squaring algorithm
Exponentiation modulo	• Right-to-left binary shift algorithm

4 Tests of ArPALib as Software

We compared the software efficiency of ArPALib (using uint32_t as a base type) to the GMP(v.6.1.1) and BigDigits(v.2.6) for numbers up to 4096-bit long. Tests were conducted on the AMD FX-6100@4 GHz, 32GB RAM DDR3-2400 machine (and compiled by MinGWv5.0RC2 with −O2 flag). To cope with a very short single computation time, and to mitigate the influence of the OS, the measurements were performed in 100,000 repeats. The average processing time for a single addition is given in Fig. 1. The multiplication and division are given in Figs. 2 and 3 respectively.

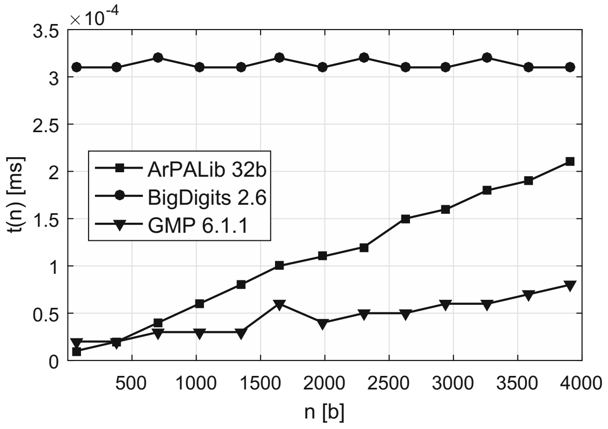

Fig. 1. A comparison of $n \times n$ unsigned integers addition/subtraction

5 A Case Study of ArPALib as Hardware

To present synthesizability and efficiency of ArPALib for FPGA hardware implementation, we will provide metrics of a custom co-processor for the Miller-Rabin

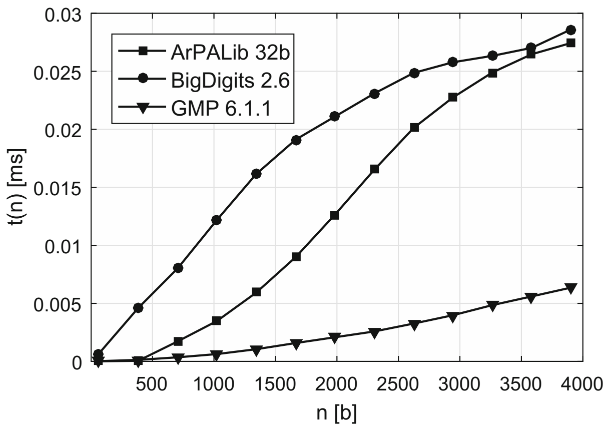

Fig. 2. A comparison of $n \times n$ unsigned integers multiplication

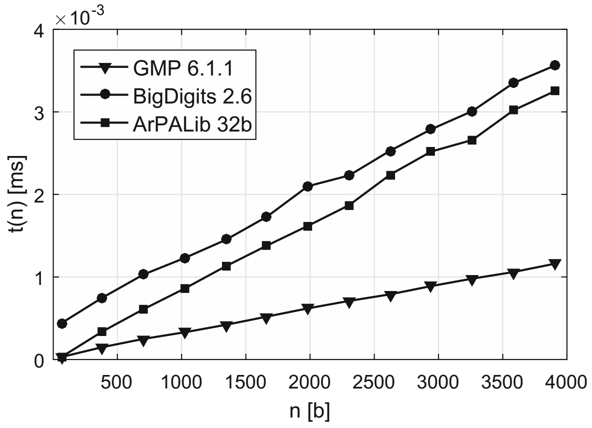

Fig. 3. A comparison of $n \times n$ unsigned integers division

algorithm that was created in the experiment, which is a well-known and widely-used number primality test that is used in security applications.

The implementation and experiments were performed on Xilinx's PSoC of Zynq-7000 family XC7Z020. The chip was a part of the Zedboard platform. Zynq combines the Cortex-A8 CPU of the ARM family and the small programmable logic of the Xilinx's 7 series FPGA. The HLS synthesis and design flow from Vivado 2015.4 development tool were used. In HLS, the architecture of the co-processor is formed according to the algorithm coded in the C programming language that is accompanied by special directives to steer the hardware synthesis (*e.g.* control parallelism or select IO interfaces). The coprocessor communication interface was built around the AXI4-Lite bus. The throughput of AXI-Lite is very modest, but it does not influence the performance of the system for the computational intensiveness of the Rabin-Miller algorithm. The goal was to implement the Rabin-Miller algorithm for the set of ten bases $a \in \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$, which guarantees the deterministic approach for the numbers up to 2^{63}, and the probabilistic test with less uncertainty than $4^{-10} \approx 9.5 \cdot 10^{-7}$ for larger numbers. The code for the implemented coprocessor is presented in Algorithm 1.

Algorithm 1. The algorithm implemented in the Miller-Rabin co-processor

```
Input: $n \in \mathbb{N}/\{1\}$ is the tested number
  procedure ProcPrimalityTest($n$)
      $K \leftarrow \{2,3,5,7,11,13,17,19,23,29\}$        ▷ The constant set of primality witnesses
      if $d \mid 2$ then                                  ▷ Check if $n$ is odd
          return $COMPOSITE$
      end if
      $s \leftarrow 0$
      $d \leftarrow$ BinaryShiftRight($n$)                ▷ $n$ is odd, so $d = (n-1)/2$
      while LSB(d)=0 do                                   ▷ LSB() gets the least significant bit
          $s \leftarrow s + 1$
          $d \leftarrow d/2$
      end while
      for $i \leftarrow 1, 10$ do
          $a \leftarrow K(i)$                             ▷ Get next element of K
          $result \leftarrow IsStrongPseudoprime(a, n, s, d)$
          if $result = COMPOSITE$ then
              return $COMPOSITE$
          end if
      end for
      return $PROBABLY\_PRIME$
  end procedure
```

We created three versions of the 4096-bit coprocessor, differing in the bit-width of the ArPALib base type uint_t (8, 16 and 32 bits). Figure 4 shows the corresponding resource usage of tested versions after FPGA implementation. Expectedly, as the uint_t size doubles, the resources requirement doubles as well. Unfortunately, any attempts to synthesize the coprocessor for the wider uint_t failed due to the FPGA resource shortage.

We also performed speed tests of co-processors. For the set of prime numbers in the range 2^{25} to 2^{4096}, the execution time of the Miller-Rabin coprocessor was measured against ARM Cortex-A9 running the software algorithm for the matching size of uint_t. Results are given in Fig. 5. The 32-bit version of hardware coprocessor runs 50% faster than its software counterpart. The speedup was even higher for the 8 and 16-bit versions, but results cannot be demonstrative as the CPU did not use its native data representation in those cases. It should be mentioned, that when the uint_t size doubles, the speed doubles as well. This scaling comes with the sequential behavior of the implemented big number operators.

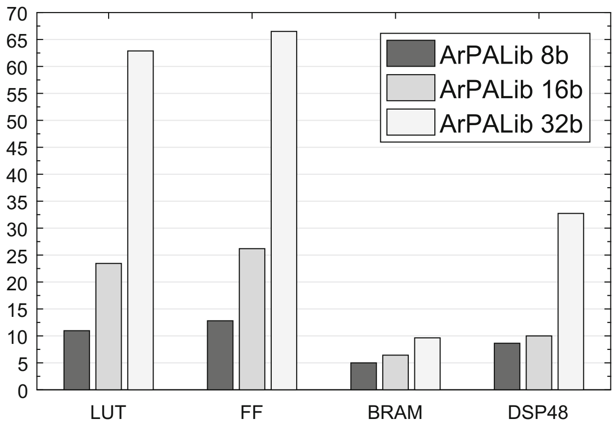

Fig. 4. Utilization of FPGA resources for Miller-Rabin co-processors in Zynq XC7Z020 FPGA. The size of the base type uint_t defined as 8, 16, and 32 bits

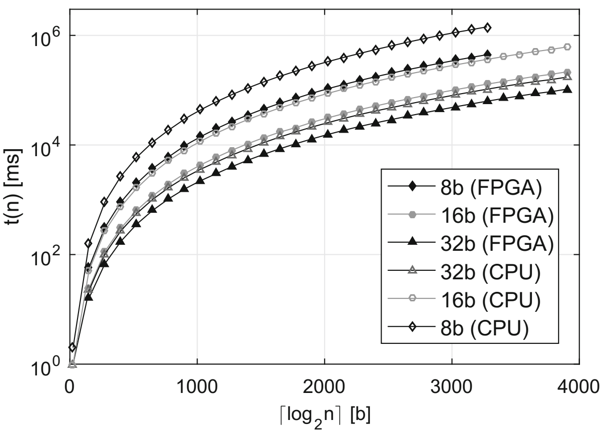

Fig. 5. ArPALib performance of the Miller-Rabin test in the hardware and software for n-bit numbers. The 8, 16, and 32-bits base type was tested and ARM Cortex-A8 (667 MHz) was used for the software version

6 Conclusions

The presented experiment proved that the hardware-software design symmetry come true thanks to HLS tools available today. At present, software routines can be positioned more easily in hardware to gain better performance. Although it requires the cautious coding style of the program, that drawback can be mitigated by the use of hardware and software compatible libraries like ArPALib.

Acknowledgements. This work was performed thanks to the funds for AGH statutory activity 11.11.230.017.

References

1. Granlund, T.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, Hong Kong (2015)
2. Ireland, D.: BigDigits multiple-precision arithmetic source code. http://www.di-mgt.com.au/bigdigits.html. Accessed 29 Sept 2016
3. Gielata, A., Russek, P., Wiatr, K.: AES hardware implementation in FPGA for algorithm acceleration purpose. In: International Conference on Signals and Electronic Systems, pp. 137–140 (2008)
4. Kryjak, T., Gorgon, M.: Pipeline implementation of the 128-bit block cipher CLEFIA in FPGA. In: International Conference on Field Programmable Logic and Applications, FPL 2009, pp. 373–378 (2009)
5. Dąbrowska-Boruch, A., Gancarczyk, G., Wiatr, K.: Implementation of a RANLUX based pseudo-random number generator in FPGA using VHDL and impulse C. Comput. Inf. **32**(6), 1272–1292 (2014)
6. Jamro, E., Russek, P., Dąbrowska-Boruch, A., Wielgosz, M., Wiatr, K.: The implementation of the customized, parallel architecture for a fast word-match program. Int. J. Comput. Syst. Sci. Eng. **26**(4), 285–292 (2011)
7. Macheta, J., et al.: ARPALib repository. https://git.plgrid.pl/projects/ARPALIB/repos/arpalib. Accessed 29 Sept 2016
8. Miller, G.L.: Riemann's hypothesis and tests for primality. J. Comput. Syst. Sci. **13**(3), 300–317 (1976)
9. Pommerening, K.: Cryptology. Part III. Primality Tests: RSA and Pseudoprimes 28 May 2000. Accessed 21 Feb 2016
10. Pomerance, C., Selfridge, J.L., Wagstaff, S.S.: The pseudoprimes to $25 \cdot 10^9$. Math. Comput. **35**(151), 1003–1026 (1980)
11. Walter, C.D.: Right-to-left or left-to-right exponentiation? International Workshop on Constructive Side-Channel Analysis and Secure Design, pp. 40–46 (2010)
12. Conrad, K.: FERMAT'S TEST. http://www.math.uconn.edu/kconrad/blurbs/ugradnumthy/fermattest.pdf
13. Solovay, R., Strassen, V.: A fast Monte-Carlo test for primality. SIAM J. Comput. **6**(1), 84–85 (1977)
14. Bach, E.: Number-theoretic algorithms. Annu. Rev. Comput. Sci. **4**(1), 119–172 (1990)