University of Cincinnati

Date: 10/18/2013

I, Akhil Kalathungal M.S., hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
An Arbitrary Precision Integer Arithmetic Library for FPGAs

Student's name:  Akhil Kalathungal M.S.

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.

Committee member: Raj Bhatnagar, Ph.D.

Committee member: George Purdy, Ph.D.
An Arbitrary Precision Integer Arithmetic Library for FPGAs

A thesis submitted to the
Division of Research and Advanced Studies
Of the
University of Cincinnati

In partial fulfillment of the
Requirements for the degree of

MASTER OF SCIENCE

In the Department of
Electrical and Computer Engineering
Of the College of Engineering and Applied Science
October 18, 2013

By

Akhil Kalathungal
B.Tech. (Applied Electronics and Instrumentation Engineering), Kerala University,
May 2007

Thesis Advisor and Committee Chair: Dr. Carla Purdy
Abstract

In computer science, arbitrary precision integer arithmetic (also called multiple precision arithmetic, bignum arithmetic) indicates that calculations are performed on numbers in which digits of precision are limited only by the available memory of the host system. A common application is public-key cryptography whose algorithms commonly employ arithmetic with integers having hundreds of digits. The intent of this thesis work is to support application programmers using FPGAs with an arbitrary precision (integer) arithmetic logic so that a user can implement cryptographic algorithms like RSA, AES etc. An application programmer who uses an FPGA based embedded processor can use the underlying hardware modules which are developed for doing the arbitrary precision operations. This thesis covers the back end of the arbitrary precision library which includes the library wrapper and the underlying arithmetic and logic unit. Algorithms have been implemented for the four basic operations: addition, subtraction, multiplication and division for multiple precision numbers. The hardware description language used to implement the library is Verilog. Test benches have been developed to test the functionality of the library and all the corner cases for the inputs have been covered with respect to verification of the output values. The library communicates with memory modules at input and output. The memory modules are on chip RAM modules; the input RAM module is used to store the two multiple precision input operands and the output RAM module is used to store the output result. The basic data chunk size of the library is selected to be 32 bits. A comparative study has been done for three different library sizes of 32 bits, 64 bits, and 96 bits for the parameters area, power and time. Also a user manual for the arbitrary precision library has been developed. Since the hardware is implemented in Verilog, this design can be used for an FPGA or even a full custom ASIC that the library code can be synthesized onto.
Acknowledgements

It has been a real long journey for me to get this thesis work accomplished and now it is time for me to jot down a note of acknowledgement to everyone who had been a part of this work. First of all, I would like to extend my sincere gratitude towards Dr. Carla Purdy, who has been my mentor and who supported me with lots of ideas regarding the design and the implementation of this thesis. Without Dr. Purdy’s help, it would have been impossible for me to complete this work within this span of time. She has always given me valid pointers, directions and ideas so that I could overcome the obstacles that I faced in my way towards this research. I am indebted to Dr Carla Purdy for the reply emails even during her vacations and email discussions in the late night hours, hence by giving me the inspiration to get this work done. I would also like to thank my committee members, Dr George Purdy and Dr Raj Bhatnagar, who were gladly willing to review my thesis work in spite of their busy work schedules and who agreed to be a part of my thesis defence committee. Special thanks to Julie Muenchen and the graduate school on their work towards my master’s graduation. Also at this moment, I would like to thank my dearest parents for their wonderful and incessant support, kindness and love since this world dawned on to me. Last but not the least, I express my sincere thanks to my sister, brother-in-law, nephew and all my dear friends who were a part of my graduate student life, for all their constant guidance, support and love. Life would have been a real tough one without you people.

Thank you all.
Contents

1 Introduction ................................................................................................................ .......................... 1
2 Background .................................................................................................................. ......................... 6
  2.1 Why limit the infinite precision library to integer operations ................................................. 6
  2.2 Software based implementations for arbitrary precision arithmetic ........................................... 7
  2.3 Application of arbitrary precision arithmetic.............................................................................. 10
  2.4 Hardware solutions for arbitrary precision arithmetic and public key cryptography ............ 11
  2.5 Requirement for this thesis ........................................................................................................ 13
3 Arbitrary precision integer library design ........................................................................................... 14
  3.1 Different tools used and devices used in the project ................................................................. 15
  3.2 More about the architecture and the ALU design ................................................................. 15
  3.3 ALU main connectivity diagram ............................................................................................. ..... 16
  3.4 Algorithms used ........................................................................................................... ............... 17
    3.4.1 ADDITION & SUBTRACTION ................................................................................................ 17
      3.4.1.1 Addition algorithm .................................................................................................... ... 19
      3.4.1.2 Subtraction algorithm ................................................................................................. ... 20
      3.4.1.3 Addition & subtraction block diagram ............................................................................ 21
    3.4.2 MULTIPLICATION .......................................................................................................... ....... 22
      3.4.2.1 Multiplication algorithm .............................................................................................. ... 22
      3.4.2.2 Multiplication block diagram .......................................................................................... 27
    3.4.3 DIVISION ................................................................................................................ .............. 28
      3.4.3.1 Division algorithm .................................................................................................... ....... 28
      3.4.3.2 Division block diagram ................................................................................................ .... 32
  3.5 Number of clock cycles required for the arithmetic operations................................................. 33
4 Hardware diagrams, experimental results, test bench and the annotated waveforms ..................... 34
  4.1 Hardware diagrams............................................................................................................... 34
    4.1.1 Arbitrary precision library: main hardware ........................................................................ 34
    4.1.2 Altera unsigned multiplier accumulator ............................................................................. 37
    4.1.3 Output dualportRAM ....................................................................................................... 37
4.3.4.3 Library size 96 bits

5 Conclusion and future work

Bibliography

APPENDIX
List of Figures

Figure 3.1: ALU main connectivity diagram ................................................................................................ 16
Figure 3.2: Addition algorithm .................................................................................................................... 19
Figure 3.3: Subtraction algorithm ............................................................................................................... 20
Figure 3.4: Addition and subtraction block diagram .................................................................................. 21
Figure 3.5: A 4 bit ripple carry adder circuit ............................................................................................... 21
Figure 3.6: Algorithm for 32 * 32 bit multiplication ................................................................................... 23
Figure 3.7: Basic multiplication algorithm from [7] .................................................................................... 24
Figure 3.8: Arbitrary precision multiplication algorithm ............................................................................ 25
Figure 3.9: Multiplication block diagram .................................................................................................... 27
Figure 3.10: Basic division algorithm .......................................................................................................... 29
Figure 3.11: Arbitrary precision division algorithm .................................................................................... 31
Figure 3.12: Division block diagram ............................................................................................................ 32
Figure 4.1: Arbitrary precision library front end hardware 1 ..................................................................... 35
Figure 4.2: Arbitrary precision library front end hardware 2 ..................................................................... 36
Figure 4.3: Altera unsigned multiplier accumulator ................................................................................... 37
Figure 4.4: Output dualportRAM ................................................................................................................ 37
Figure 4.5: Input dualportRAM ................................................................................................................... 38
Figure 4.6: Arbitrary precision data divider hardware 1 ............................................................................ 40
Figure 4.7: Arbitrary precision data divider hardware 2 ............................................................................ 41
Figure 4.8: Unsigned arbitrary precision accumulator hardware 1 ............................................................ 42
Figure 4.9: Unsigned arbitrary precision accumulator hardware 2 ............................................................ 43
Figure 4.10: Example flow summary report for hand coded RAM ............................................................. 45
Figure 4.11: Common area elements for hand coded RAM ....................................................................... 45
Figure 4.12: Example flow summary report for Altera LPM RAM .............................................................. 46
Figure 4.13: Common area elements for Altera LPM RAM ........................................................................ 47
Figure 4.14: Total logic elements for FPGA: Cyclone II ............................................................................. 48
Figure 4.15: Total combinational functions for FPGA: Cyclone II ............................................................... 48
Figure 4.16: Dedicated logic registers for FPGA: Cyclone II ..................................................................... 49
Figure 4.17: Combinational ALUTs for FPGA: Arria II GX ....................................................................... 49
Figure 4.18: Dedicated logic registers for FPGA: Arria II GX ..................................................................... 49
Figure 4.19: Combinational ALUTs for FPGA: Cyclone V ....................................................................... 50
Figure 4.20: Dedicated logic registers for FPGA: Cyclone V ..................................................................... 50
Figure 4.21: Example PowerPlay Power Analyzer tool report for hand coded RAM ................................. 51
Figure 4.22: Example PowerPlay Power Analyzer tool report for Altera LPM RAM ................................. 52
Figure 4.23: Total thermal power dissipation for FPGA: Cyclone II ............................................................ 54
Figure 4.24: Core dynamic thermal power dissipation for FPGA: Cyclone II ............................................. 54
Figure 4.25: Core static thermal power dissipation for FPGA: Cyclone II ................................................... 55
Figure 4.26: I/O thermal power dissipation for FPGA: Cyclone II ............................................................... 55
Figure 4.69: Subtraction 64 bit; no OVERFLOW (positive – positive = positive) ......................................... 87
Figure 4.70: Subtraction 64 bit; no OVERFLOW (positive – positive = negative) ........................................ 88
Figure 4.71: Subtraction 64 bit; with OVERFLOW ..................................................................................... 90
Figure 4.72: Subtraction 64 bit special case 1; no OVERFLOW ................................................................. 92
Figure 4.73: Subtraction 64 bit special case 2; no OVERFLOW ................................................................... 93
Figure 4.74: Subtraction 96 bit; no OVERFLOW (positive – positive = positive) ......................................... 95
Figure 4.75: Subtraction 96 bit; no OVERFLOW (positive – positive = negative) ........................................ 97
Figure 4.76: Subtraction 96 bit; with OVERFLOW .................................................................................... 99
Figure 4.77: Subtraction 96 bit special case 1; no OVERFLOW ................................................................. 101
Figure 4.78: Subtraction 96 bit special case 2; no OVERFLOW ................................................................. 102
Figure 4.79: Multiplication 32 bit .............................................................................................................. 104
Figure 4.80: Multiplication 64 bit input operands .................................................................................... 106
Figure 4.81: Multiplication 64 bit product blocks ..................................................................................... 107
Figure 4.82: Multiplication 96 bit input operands .................................................................................... 109
Figure 4.83: Multiplication 96 bit first three product blocks ..................................................................... 110
Figure 4.84: Multiplication 96 bit, last three product blocks ..................................................................... 110
Figure 4.85: Division 32 bit, quotient and remainder ............................................................................... 112
Figure 4.86: Division 64 bit, input operands storage ................................................................................ 113
Figure 4.87: Division 64 bit, quotient blocks and remainder blocks ......................................................... 114
Figure 4.88: Division 96 bit, input operands storage ................................................................................ 116
Figure 4.89: Division 96 bit, quotient blocks ............................................................................................. 116
Figure 4.90: Division 96 bit, remainder blocks ......................................................................................... 117

List of Tables

Table 2.1: Algorithms and thresholds for GNU MP Library .......................................................................... 9
Table 3.1: Number of clock cycles taken .................................................................................................... 33
Table 4.1: Summary table for experiments ................................................................................................ 44
Table 4.2: Area comparison for hand coded RAM ...................................................................................... 46
Table 4.3: Area comparison for Altera LPM RAM ....................................................................................... 47
Table 4.4: Power comparison for hand coded RAM ................................................................................... 52
Table 4.5: Power comparison for Altera LPM RAM .................................................................................... 53
Table 4.6: Timing comparison for hand coded RAM .................................................................................. 61
Table 4.7: Timing comparison for Altera LPM RAM .................................................................................... 64
Table 4.8: Table based on timing parameters ............................................................................................ 70
Chapter 1

1 Introduction

Arbitrary precision arithmetic is used in applications where precise results with very large numbers are required. In [1], K.R Ghazi et al. describe the way in which arbitrary precision increases the accuracy and reproducibility of computations. It can be used in public-key cryptography such as Ron Rivest, Adi Shamir and Leonard Adleman’s algorithm (RSA) or the Advanced Encryption Standard (AES), whose algorithms commonly employ arithmetic with integers having hundreds of digits. It is applicable in situations where artificial limits and overflows would be inappropriate. If there is an arbitrary precision library, then that can be used to compute fundamental mathematical constants such as $\pi$ to millions of digits and to analyze the properties of the digit strings or more generally to investigate the precise behavior of functions such as the Riemann zeta function where certain questions are difficult to explore via analytical methods. Another example is in rendering fractal images with an extremely high magnification, such as those found in the Mandelbrot set.

An inherent limitation of fixed-precision arithmetic is overflow, and arbitrary-precision arithmetic can be used to avoid it. A simple example for an overflow is a 5-digit odometer's display which changes from 99999 to 00000, when the numbers on display grow too large to represent. Similarly, a fixed-precision integer may exhibit wraparound if numbers grow too large to represent at the fixed level of precision. In some processors the overflow issue is dealt by saturation, that means that if a result is not representable, then it is replaced with the nearest representable value. (For example with 16-bit unsigned saturation, adding any positive amount to 65535 would yield 65535.) In some processors, an exception is generated if an arithmetic result exceeds the available precision. So a motivation for this
work is, wherever necessary, the exception can be caught and recovered from—for instance, the
operation could be restarted in software using the arbitrary-precision arithmetic library.

There is support for arbitrary precision arithmetic in some high level programming languages such as
Lisp, Python, Perl, Haskell, Ruby etc. Those languages use, or have an option to use, arbitrary-precision
numbers for all integer arithmetic. In high level languages like C and C++, there are multiple precision
libraries available which can be used. GNU MP [2] (GMP) is a free library arbitrary precision arithmetic,
operating on signed integers, rational numbers, and floating point numbers. Also The MPFR library [3] is
a C library for multiple-precision floating-point computations with correct rounding. The use of the
arbitrary precision arithmetic reduces performance; however it eliminates the possibility of incorrect
results (or exceptions) due to simple overflow. Also the arithmetic results will be the same on all
machines, regardless of any particular machine’s word size. The exclusive use of arbitrary-precision
numbers in a programming language also simplifies the language, because a number is a number and
there is no need for multiple types to represent different levels of precision.

There is no hardware implementation for such an arbitrary precision library expect for some proprietary
encryption chips from Broadcom [4] and IBM [5]. Those VLSI chips are used for encryption algorithms in
the Secure Sockets Layer (SSL) which uses proprietary hardware architecture for encryptions. However
that architecture will be specific to the encryptions algorithms which have to be supported using those
chips. So an idea of arbitrary precision integer library arithmetic is a new concept for the hardware
chips.

This thesis deals with a hardware implementation in Verilog for an arbitrary precision integer arithmetic
library. The end application of the library is a code which can be synthesized onto any FPGA. Till now,
the encryption algorithms on any FPGA s use fixed precision arithmetic which may not be secure enough for a good encryption. For example, an RSA key size of 512 can be factorized by decryption hardware or algorithm [6]. But if we have an arbitrary precision synthesizable library for the FPGAs, the encryption schemes can be made more secure. Also when people use Verilog, an ALU can be easily designed by assigning the ALU data bus width size to be equal to the input operand size. However for an arbitrary precision logic, the input operands can be as long as 1024 or 2048 bits or even more. Hence if we design an ALU which can do the basic addition, subtraction, multiplication and division for that big data bus size, that specific design may not be synthesizable onto an FPGA since the FPGA may not have that many input/output pins to get the operand bits from. So the approach used in this thesis is to store the input operands in an on chip RAM and to clock the operands into the ALU, 32 bits per clock cycle.

The data chunk size has been fixed to 32 bits for several reasons. Since this thesis is a hardware ALU design which communicates with the memory modules on the input and the output, an efficient approach is to reduce the number of input and output pins. Otherwise the design may not be synthesizable onto a small chip which has limited inputs and outputs. Hence in this case, a 32 bit data chunk size is preferred to a 64 bit data chunk size. Also for an Altera FPGA device, an application programmer who writes the upper layer software uses the NIOS II embedded-processor architecture which is a 32-bit one. The library which is developed as a part of this thesis can be used by an application programmer in NIOS II and in that case a 32-bit data chunk size is required for a proper communication with the library.

This design approach can also be used in full-custom VLSI design methodology. Thus our implementation in FPGA is like a design ‘prototype’. In full-custom design, the layouts of each individual transistor and the individual connections between them have to be specified. Even if that approach maximizes the
performance of the chip, it is extremely labor-intensive to implement. The algorithms discussed in this thesis are translated into software using Verilog which in turn are synthesized into hardware modules. Hence the design is equivalent to creating standard cell libraries. So this project can be treated as an open source for any integrated circuit such as an ASIC or an FPGA.

Also the algorithms which are used for implementing the mathematical operations are primitive ones (like the pen and paper method) since this work is a proof of concept that such algorithms can be implemented in hardware. A lot of future work needs to be done to develop more sophisticated algorithms and that is an extension to this thesis.

The goals of this work include:

- Implement the basic algorithms for addition, subtraction, multiplication and division using Verilog HDL
- Implement the architecture of the entire backend of the arbitrary precision library (which includes the memory modules and the ALU) and provide a user manual
- Use the Altera FPGA CAD Tool Quartus II to make sure the Verilog library is synthesizable onto the hardware
- Extract the hardware library models from Quartus II and simulate using the ModelSim simulator. Verify for the functionality using test benches. Verify the functionality for different Library widths.
- Do a comparison study between the different FPGA devices on the Altera board (e.g. Cyclone series, Arria series) with respect to area, power and the timing
- Do a comparison study between the memory modules which had been hand coded in Verilog and the Altera Library Of Parameterized Modules (LPM) with respect to area, power and the timing
- Discuss further improvements in the future for an efficient library implementation
The thesis is organized as follows:

**Chapter 1** is the Introduction which lists the motivation for the work, the goals of this thesis and the thesis organization.

**Chapter 2** reviews the background on arbitrary precision arithmetic libraries, the software implementation and the chips in industry which supports encryption for arbitrary precision.

**Chapter 3** presents the approach taken in this paper, the entire system architecture and the different algorithms and the hardware configuration diagrams which had been used for implementation.

**Chapter 4** describes how to set up the experiments to verify the proposed method and presents the results. This section discusses the test benches and the way the test bench is written so that it covers all the corner cases for the inputs.

**Chapter 5** presents conclusions and future research directions.

A user data manual for an application programmer on FPGA is provided in an Appendix.
Chapter 2

2 Background

In this chapter, we discuss the background knowledge related to our thesis project, which involves the software implementation for the arbitrary precision arithmetic libraries.

2.1 Why limit the infinite precision library to integer operations

The goal of this paper is to implement infinite precision integer arithmetic algorithms as a library in the hardware and to test the library on a field programmable gate array (FPGA). At this moment, we consider only integer operations, not arbitrary precision floating point operations. This is because the floating point operations internally use integer arithmetic logic on the significand and the exponent, and the operations can use the integer ALUs. Also floating point needs some extra hardware for normalization. A floating point number is often represented approximately as a fixed number of significant digits (the mantissa) and scaled using an exponent. The typical number that can be represented exactly is of the form:

\[ \text{Significant digits} \times \text{base}^{\text{exponent}} \]

Addition and subtraction: A simple method to add/subtract floating-point numbers is to first represent them with the same exponent, with some adjustments done for the round-off errors. Multiplication and division: To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized. Similarly, division is accomplished by subtracting the divisor’s exponent from the dividend’s exponent, and dividing the dividend’s significand by the divisor’s significand. More study is done in [7], where, Figures 3.15 and Figures 3.17 explain the algorithms for addition and multiplication.
2.2 Software based implementations for arbitrary precision arithmetic

Arbitrary precision arithmetic has lots of software based implementations, either as a part of a high level programming language which has a built in library or as a separate library which can be incorporated into a high level language. In the latter case, the library can be used with the help of the user interface functions provided. In both, the arbitrary precision data type is stored in the memory using linked lists or arrays. The various algorithms operate on the linked lists or the array data contents. For example, the GNU MP Library [2] and the MPFR library [3] have their own data containers for handling arbitrary precision and the algorithms.

Java has its built-in library biginteger class for doing infinite precision math, ‘Class BigInteger’. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations. All the summary and detail of the class namely the Field, the Constructor, Methods etc. are given in [8].

Perl has its own built-in library for doing infinite precision math, ‘bignum’ [9]. It is just a thin wrapper around various modules of the Math::BigInt family in Perl. The following modules are currently used by bignum: 1) Math::BigInt::Lite (for speed, and only if it is loadable) 2) Math::BigInt 3) Math::BigFloat. The numbers are stored as objects, and their internals might change at any time, especially between math operations. The objects also might belong to different classes, like Math::BigInt, or Math::BigFloat.
In Python the built-in \texttt{int} (3.x) / \texttt{long} (2.x) integer type is of arbitrary precision. Long integers have unlimited precision\cite{10}. The \texttt{Decimal} class in the standard library module \texttt{Decimal} has user definable precision and limited mathematical operations (exponentiation, square root, etc. but no trigonometric functions). The \texttt{Fraction} class in the module \texttt{Fractions} implements rational numbers. Third-party packages like "mpmath" and "bigfloat" are available, where more extensive arbitrary precision floating point arithmetic can be done.

In C and C++, there is no built in support for doing infinite precision. However there are external libraries that provide data types and subroutines to store numbers with the required precision and to perform computations. GNU Multiple Precision Arithmetic Library (GMP) is a free library, operating on signed integers, rational numbers and floating point numbers. There are no practical limits to the precision except the ones implied by the available memory in the machine GMP runs on (operand dimension limit is $2^{32}-1$ bits on 32-bit machines and $2^{37}$ bits on 64-bit machines). GMP developers claim is faster than any other bignum library for all operand sizes by incorporating some important factors:

- Using full words as the basic arithmetic type.
- Using different algorithms for different operand sizes; algorithms that are faster for very big numbers are usually slower for small numbers.
- Highly optimized assembly language code for the most important inner loops, specialized for different processors

The GNU MP library is well documented in\cite{2}. The main target platforms are Unix-type systems, such as GNU/Linux, Solaris, HP-UX, Mac OS X/Darwin, BSD, AIX, etc. It also is known to work on Windows in both 32-bit and 64-bit mode. In the library a limb means the part of a multi-precision number that fits in a single machine word, and normally a limb are 32 or 64 bits. Multiplication is done for the input operands
based on the size of the input operands. The package has a threshold for the number of digits in the
input operand and based on the threshold the corresponding algorithm is invoked for the multiplication.
So the base case N * M limb multiplication is a straight forward rectangular set of cross-products, the
same as long multiplication done by hand. Based on the different operand threshold, the library has
solutions based on algorithms like Karatsuba multiplication (described in [11] section 4.3.3 part A),
Toom-3 multiplication (described in [11] section 4.3.3 part A) Toom-4, Toom-6.5, Toom-8.5, FFT etc. The
table given next lists the different multiplications which are selected by the GNU MP library based on
the input operand threshold.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basecase</td>
<td>(none)</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>MUL_TOOM22_THRESHOLD</td>
</tr>
<tr>
<td>Toom-3</td>
<td>MUL_TOOM33_THRESHOLD</td>
</tr>
<tr>
<td>Toom-4</td>
<td>MUL_TOOM44_THRESHOLD</td>
</tr>
<tr>
<td>Toom-6.5</td>
<td>MUL_TOOM6H_THRESHOLD</td>
</tr>
<tr>
<td>Toom-8.5</td>
<td>MUL_TOOM8H_THRESHOLD</td>
</tr>
<tr>
<td>FFT</td>
<td>MUL_FFT_THRESHOLD</td>
</tr>
</tbody>
</table>

A single limb division (N * 1 division), is implemented using repeated 2×1 divisions from high to low,
either with a hardware divide instruction or a multiplication by inverse as presented in [12], whichever is
best on a given CPU. Base case division (N×M division) is like long division done by hand [11], but in a
different base, 2^{mp\_bits\_per\_limb}, where mp\_bits\_per\_limb is the number of bits per limb (32 on a 32 bit
machine or 64 on a 64 bit machine). With Q=N−M the number of quotient limbs, this is an O(QM)
algorithm and will run at a speed similar to a base case Q×M multiplication, differing in fact only in the
extra multiply and divide for each of the Q quotient limbs. For example if N is 5 and M is 3, then the
quotient will have Q = 5 − 3 = 2 limbs (on a 32 bit or a 64 bit CPU), and the algorithm will be of the order
of $O(5 \times 3)$. For divisors larger than a specific threshold \texttt{DC\_DIV\_QR\_THRESHOLD}, division is done by using a recursive divide and conquer algorithm based on [13]. \texttt{DC\_DIV\_QR\_THRESHOLD} is based on the divisor size $N$, so it will be somewhere above twice \texttt{MUL\_TOOM22\_THRESHOLD}, but how much above depends on the CPU. For bigger operands, other methods like Block-Wise Barrett Division, Exact division [14], Exact Remainder, Small Quotient Division etc. are used.

2.3 Application of arbitrary precision arithmetic

Application of such arbitrary precision integer arithmetic is mainly for encryption algorithms like RSA, AES etc. RSA keys are typically 1024 to 2048 bits long. Some experts believe that 1024-bit keys may become breakable in the near future [15] (though this is disputed since the device The Weizmann Institute Relation Locator, TWIRL has not been built yet [16]); a few say even 4096-bit keys could be broken in the foreseeable future. Therefore, it is generally presumed that RSA is secure if $n$ (the number of bits in the key) is sufficiently large. Keys of 512 bits were shown to be practically breakable as early as 1999 when RSA-155 (155 decimal digits, 512 bits) was factored by using several hundred computers [6] and are now factored in a few weeks using common hardware [17]. Exploits using 512-bit code-signing certificates that may have been factored were reported in 2011. A theoretical hardware device named TWIRL (The Weizmann Institute Relation Locator) and described by Shamir and Tromer in 2003 called into question the security of 1024 bit keys. The work is discussed in [15]. It is currently recommended that $n$ be at least 2048 bits long.

AES has a fixed block size of 128 bits, and a key size of 128, 192 or 256 bits. It has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. By 2006, the best known attacks were on 7 rounds for 128-bit keys, 8 rounds for 192-bit keys, and 9 rounds for 256-bit keys. In 2002, a theoretical attack, termed the "XSL attack" [18], was announced by Nicolas Courtois and Josef Pieprzyk, purporting to show a weakness in the AES algorithm due to its simple description. On July 1, 2009, a
related-key attack was reported on the 192-bit and 256-bit versions of AES, discovered by Alex Biryukov and Dmitry Khovratovich, [19] which exploits AES's somewhat simple key schedule and has a complexity of $2^{119}$. In December 2009 it was improved to $2^{99.5}$. The first key-recovery attacks on full AES were due to Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger, and were published in 2011 [20]. The attack is based on bicliques and is faster than brute force by a factor of about four. It requires $2^{126.1}$ operations to recover an AES-128 key. For AES-192 and AES-256, $2^{189.7}$ and $2^{254.4}$ operations are needed, respectively.

2.4 Hardware solutions for arbitrary precision arithmetic and public key cryptography

At present, there are no hardware solutions for the public key cryptography encryptions. However there are special purpose custom made chips. These chips are intended to work with the SSL (Secure Sockets Layer) Protocol which in turn uses a public key cryptography scheme like an RSA and hence they contain the intelligent hardware for doing some of the arbitrary precision arithmetic. The need for such hardware accelerators in SSL Protocol had been discussed in [21]. There are a couple of solutions available. The first one is from Broadcom, the CryptoNetX™ SSL4000 ACCELERATOR ADAPTER; the need for which is described in [22]. The chip is described in [4]. From the data manual, the solution is assumed to be implemented in the hardware, we could not explore much since the CryptoNetX architecture is a Broadcom proprietary one. A single SSL4000 allows ecommerce devices to achieve sustained throughput of 4000 connections per second, nearly a 2000% increase in performance over SW-based ecommerce devices (733 MHz Pentium® III, Windows IIS).

IBM mainframe computers (family name System z) offload the cryptography service functions to dedicated processors. System z servers are used by IBM customers for business-critical installations in medium and large organizations which need very high availability, where scheduled and unscheduled
downtime costs are high, and at traditional "mainframe shops" such as banks and insurance companies which already have mainframe applications at the center of their business processes. The cryptographic functions support started with the family of servers named System z9. The System z9 architecture (The IBM System z9 Enterprise Class) is as explained in [23] which describe the cryptography used in [23] Chapter 7. The architecture explains three ways for supporting cryptography:

A) Central Processor (CP) assists for cryptographic functions (CPACF) (processor-based encryption). CPACF has been enhanced to include support of the Advanced Encryption Standard (AES) for 128-bit keys; Secure Hash Algorithm-256 (SHA-2); and Pseudo Random Number Generation (PRNG). This standard provides a set of symmetric cryptographic functions that focus on the encryption/decryption function of SSL, Virtual Private Network (VPN) etc.

B) Crypto Express2 (CEX2C) feature is designed to satisfy high-end server security requirements. This executes functions that were previously offered by PCICA (PCI Cryptographic Accelerator) and PCIXCC (PCI Cryptographic Coprocessor) features, performing hardware acceleration for SSL transactions and clear key RSA acceleration. This feature uses a single cryptographic coprocessor. It has secure (encrypted) keys for AES-128 and 2048-bit key RSA management capability.

C) User-Defined Extensions: For unique customer applications, Crypto Express2 will support the loading of customized cryptographic functions on z9 EC. Under a special contract with IBM, as a Crypto Express2 customer, the user gains the flexibility to define and load custom cryptographic functions yourself.

Later on IBM moved to the next version of mainframes, the IBM System z10 which has a quad core processing engine. The architecture has also been modified from z9, and for cryptography, the System
z10 adds hardware-based 192-bit and 256-bit AES in addition to the 128-bit AES support already available on the z9. The IBM 4765 cryptographic coprocessor hardware and firmware architecture has been discussed in [5]. This coprocessor has been introduced for the z10 architecture, and there are new CPACF features included as well.

2.5 Requirement for this thesis

As far as we know arbitrary precision integer arithmetic has no Hardware Description Language (like Verilog or VHDL) solution for an ASIC or an FPGA. That became the starting point of this work and hence this paper is a proof of concept for the hardware algorithms. In this thesis, Verilog is the HDL which is used to implement all the algorithms (we can use VHDL as well, since both the languages are accepted as HDL standards).
Chapter 3

3 Arbitrary precision integer library design

This thesis is a hardware design project and the implementation has been done on the Altera DE1 FPGA board with Cyclone II FPGA device. Only the backend of the library, i.e. the library wrapper and the underlying Arithmetic and Logic Unit (ALU) have been designed and implemented. The paper limits discussions to only multiple precision integer arithmetic at this moment. The fixed point (or the floating point) arithmetic is a future extension to the library. The ALU supports the four basic operations, addition, subtraction, multiplication and division, and algorithms have been designed to implement these four operations with multiple precision. There are two design parameters for the library out of which one can be user configured: One is the library width (this is user configured, which is the number of bits of operation that has to be supported by the library) and second is the data width. The data width is hard coded to 32 bits (that is, 32 bits of data chunks for operands A and B every clock) since this project is implemented on the Altera DE1 FPGA board. An application example usage for this ALU library is in an RSA or AES encryption application algorithm implementation on the FPGA hardware. Normally these algorithms will be written in C language using the NIOS II embedded processor with the Micro C/OS II [24] operating system (or in general, any operating system which is supported by the FPGA) which has a data width size of 32 bits. With that school of thought, the data width has been fixed at 32 bits and the library is implemented for a data bus size of 32 bits. However if a user want to use a library data bit width other than 32, the library can be modified to use that data width. For that reason, even the data width has been parameterized in the ALU design. Another design aspect is with the memory modules which have been used to store the input operands and to store the output results. To store the arbitrary precision library input operands and the library output results, on chip RAM modules have been used. Two separate designs have been covered in this perspective, one in which the Altera specific
on chip LPM (Library of Parameterized Modules) memory RAM has been used and the other in which a memory module (hand coded in Verilog) has been synthesized onto a hardware memory block. A comparative study has been done between these two implementations for four different op codes (addition, subtraction, multiplication and division) for the following parameters: area, timing and power. Hence in short, a user can select data blocks of 1 - N in number (where each block is of size 32 bits) for a library of size 1 - N and the maximum number of data blocks that can be selected is only limited by the size of the FPGA device the library is synthesized onto.

The tables in the results section show the comparison between different implementations and help a user to select the LPM RAM or their own memory module based on his requirements. Also the test benches have been developed to test the four different operations with different library sizes. An effort has been made to cover all the corner cases since exhaustive testing for big size library widths is impossible to do. The waveforms and results have been collated in the results section. The library has been verified for its functionality using the test bench test output waveforms.

3.1 Different tools used and devices used in the project

FPGA CAD Tool Used: Quartus II 12.0 Web Edition (32-bit), From Altera

Hardware Simulator Used: ModelSim-Altera 10.0d (Quartus II 12.0) Starter Edition

Hardware Used: Altera DE1 FPGA board

3.2 More about the architecture and the ALU design

On chip memory RAM has been used to store the input operands and the output results as shown in Figure 3.1. For encryption algorithms, the input operands and the keys used for encryption can be stored in on chip RAMs. It might be possible to implement a read locking mechanism from the input RAM if an application code tries to read those data directly. Hence, this method prevents an attacker
from reading the secure input operands and the key values using an application code. Please note that
the same architecture can be applied to ASIC logic as well. The architecture presents a general method
to store the keys and operands; however it can be modified. For example, a storage mechanism for the
keys and the operands can be provided through system hardware registers. Or we can make use of
external memories on the board like SDRAM, flash memory etc.

The Arithmetic and Logic Unit (ALU):
We have implemented a synchronous, clocked ALU with control signals and the data path which
supports four different arithmetic operations: ADD, SUB, MULT and DIV. The arbitrary precision library
has its own wrapper which is connected to the internal real ALU.

3.3 ALU main connectivity diagram
The ALU is connected with Input RAM and Output RAM with data buses as shown in Figure 3.1.

![Figure 3.1: ALU main connectivity diagram](image-url)
The control signals are given to the library wrapper from an application program. The control signals include the clock signal and the op code required for the library. The two operands for the operation, operand A and operand B, are stored in the on chip system input RAM. The application program has to do this before the library starts its operation. After the input RAM has been set with specific operands, the application has to give the wrapper the required op code (as a control signal) to start the corresponding arithmetic operation. The wrapper reads the input RAM (which has operand A and operand B data chunks) block wise on each clock, does the operations and writes the results as blocks to the output RAM in the very next clock cycle. The library reads the written results from the output RAM and gives it to the application on the output RAM write cycle. For the detailed library description diagram with individual signal descriptions, please see the data manual (Appendix). For the detailed operation sequence of each and every op code, please see section 6.1.3 of the Appendix.

3.4 Algorithms used

The algorithms used for addition, subtraction, multiplication and division are discussed below.

3.4.1 ADDITION & SUBTRACTION

The addition operation has been implemented in the internal ALU using a basic Ripple Carry Adder method which has been implemented structurally. The ADD algorithm uses a ripple carry adder [25]approach between the data chunks, which means the carry signal from the first data chunk addition is propagated to the next data chunk addition and so on. For an unsigned addition, the final carry bit can be read using an output pin ‘carry_signal’ and the overflow pin ‘add_sub_overflow’ can be ignored. The final carry bit can be read back from the output RAM also for an unsigned addition. For signed addition, the overflow pin ‘add_sub_overflow’ is set if there is any overflow and the output pin ‘carry_signal’ can be ignored. The subtraction operation uses the ripple adders, and the subtraction is done with the help of the library wrapper doing the two’s complement conversion for the numbers before they are fed to
the internal Ripple Adders. It is assumed that the processor uses the two’s complement method. An overflow pin ‘add_sub_overflow’ is provided to indicate overflows for addition and subtraction operations. More details on the library usage, the data interpretation, and the algorithms for multiplication, division etc. are described in detail in the library data manual as provided in the Appendix. Please see Appendix for the library usage instructions for each and every op code.

Addition and Subtraction use the same algorithm with one basic difference. For Subtraction, before feeding the input to the Structural Ripple Carry Adders, the operand B is complemented and added with 1 to get its equivalent two’s complement. Then the remaining operation is continued the same as that of an addition operation. An overflow occurs when the result from an operation cannot be represented with the available hardware. When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, suppose the words size is 32-bit and the addition operation is -10 + 4 = -6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands. Hence in addition overflow happens only when: 1) Adding two positive numbers gives a negative result and 2) Adding two negative numbers gives a positive result. The arbitrary precision library has its logic which catches the above two cases and set the overflow pin ‘add_sub_overflow’.

For subtraction it is just the opposite principle, when the signs of the operands are the same, overflow cannot occur. To see this, remember \( x - y = x + (-y) \) because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up adding operands of different signs. And from the deduction for addition, we know that overflow cannot occur in this case either. The overflow rule for subtraction states that, if two two's complement numbers are
subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend. The arbitrary precision library has its logic which catches the above overflow cases for two’s complement subtraction and sets the overflow pin ‘add_sub_overflow’.

### 3.4.1.1 Addition algorithm

Figure 3.2 show the steps in the addition algorithm.

```
First clock:
Start
> first data chunks for A and B

\{(Internal Carry Prev, 32 bit SUM) = Operand A + Operand B; \}
> '+' means Structural Ripple Carry Adders

Write 32 bit SUM to output RAM

End

All other clocks:
Start

\{(Internal Carry, 32 bit SUM) = Operand A + Operand B + Internal Carry Prev; \}
> '+' means Structural Ripple Carry Adders

> This value is generated and propagated
> between clocks for new data chunks
> Propagated as Internal Carry Prev.

Internal Carry Prev = Internal Carry;
Write 32 bit SUM to output RAM
If (clock == Final Clock)
Start
Check for overflow and set the overflow output pin
Write Internal Carry to the output RAM, since this is the final carry bit
End
End

Figure 3.2: Addition algorithm
```
3.4.1.2 Subtraction algorithm

In addition to the Addition algorithm, here the 2’s complement conversion is implemented for the operand B data chunks. Please note that only the first data chunk needs to be complemented and added with 1 with since it is the LSB data block. All the other data chunks are just complemented and wait for the carry bit which is rippled from the previous data chunk.

Figure 3.3 show the steps in the subtraction algorithm.

Figure 3.3: Subtraction algorithm
3.4.1.3 Addition & subtraction block diagram

Figure 3.4: Addition and subtraction block diagram

Ripple Carry Adders have been used for adding the data chunks of size 32 bits. Figure 3.5 shows a 4 bit Ripple Carry Adder circuit.

Figure 3.5: A 4 bit ripple carry adder circuit

Each full adder has a Cin (carry in) input, which is the Cout (carry out) of the previous adder. This kind of adder is called a ripple-carry adder, since each carry bit "ripples" to the next full adder. The first (and
only the first) full adder may be replaced by a half adder for a ripple carry adder. The layout of a ripple-carry adder is simple, which allows for fast design time; however, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit ripple-carry adder, there are 32 full adders, so the critical path (worst case) delay is 3 (from input to carry in first adder) + 31 * 2 (for carry propagation in later adders) = 65 gate delays. The implementation hence has an effect on the upper frequency that can be attained by the ALU library.

We could have designed the internal adders with faster methods such as the ‘Carry-look ahead adders’ [26]. However from an optimization perspective, the size of the synthesized hardware on the target board is also a constraint. The Carry-look ahead adders make the internal ALU design more complex and use more area. So in this thesis work, operating frequency of the ALU library has been compromised with respect to the size of the hardware that is synthesized. Carry-look ahead adders can be implemented in the future if someone would like to make the arbitrary precision library faster. In that scenario, the basic architecture of the arbitrary precision library remains the same, only the internal structural full adder design will change.

3.4.2 MULTIPLICATION

3.4.2.1 Multiplication algorithm

The algorithm which has been used for multiplication is the basic longhand pen and paper method; hence it takes lots of clock cycles to run. To avoid this latency for a normal 32 bit * 32 bit multiplication, a 32 bit multiplication is implemented using the Altera Multiplier Accumulator module so that the basic data width multiplication (32 * 32 bit) does not take many clock cycles. The Altera website [27] gives us a Verilog HDL example for an unsigned multiplier-accumulator. Synthesis tools are able to detect
multiplier-accumulator designs in the HDL code and automatically infer the ‘altmult_accum’ mega function to provide optimal results (In this scenario, Quartus II CAD Tool does the job). For more details on the Integer Arithmetic Mega functions User guide from Altera, please see the document [28]. So we have two separate algorithms: one for the basic 32 * 32 bit and other one for any other number of data chunks. The algorithm is implemented for unsigned numbers; the implementation for signed number multiplication is a future extension to the library. The algorithm for 32 * 32 bits is given in Figure 3.6.

![Algorithm for 32 * 32 bit multiplication](image)

Figure 3.6: Algorithm for 32 * 32 bit multiplication

The algorithm for any other arbitrary bit width multiplication (say 64 * 64, 96 * 96) is a modification to the basic multiplication algorithm given in [7] Figure 3.5, which is given next and shown in Figure 3.7.
The above multiplication algorithm is discussed for a word size of 32-bits. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

This previous given algorithm is extended for an arbitrary precision library width with the help of an output RAM which holds all the intermediate products and an accumulator which adds all the intermediate products accordingly. The arbitrary precision multiplication algorithm is as explained in the Figure 3.8.
Figure 3.8: Arbitrary precision multiplication algorithm
The accumulator module for the previous multiplication logic has also been designed and hand coded in Verilog. The accumulator is not directly connected to the library wrapper; it is connected to the output RAM to read the intermediate products into its own hardware registers. This is made to ensure more security in the system. From the library perspective, since the output RAM only stores the intermediate products, the final product is generated by the accumulator and the application code has a provision to read the final product from the accumulator directly. A pin named ‘acc_out_for_test’ is provided with the intention that the application code can read that pin after a specified number of clock cycles (as specified in the Data Manual provided in the Appendix; also specified in the Results section) to get the final product. Since the multiplication operation is done only on the unsigned integers, we do not have to account for the overflow. The accumulator is internally designed such that it selects a hardware register size based on the library data width and hence it has enough space to store the sum of the intermediate products. Say the operands A and B are 64 bits then the accumulator will have 128 bits to store the final product. In general, for multiplication an n bit operand multiplied by another n bit operand will generate a product of 2n bits and the accumulator design is such that it can hold the entire 2n bits of the final product.
3.4.2.2 Multiplication block diagram

Figure 3.9: Multiplication block diagram

Figure 3.9 shows the multiplication block diagram and the associated hardware which has been used for arbitrary precision multiplication. Please note that the above diagram is the generic one for arbitrary precision multiplication and not the specific 32 * 32 bit multiplication. Hence the Accumulator which has been given in Figure 3.4 is not the Altera specific multiply and Accumulator. The accumulator is a basic accumulator (with no multiplication feature) and it is hand coded in Verilog. For more details on the working of the arbitrary precision Multiplication please see the Data Manual section in the Appendix. A future extension to this algorithm is to use faster multiplication methods such as Karatsuba [11], Toom-3 or FFT [11] which needs intelligent hardware modules. However those faster methods are going to take more space on the chips and that is a trade-off.
3.4.3 DIVISION

3.4.3.1 Division algorithm

Division, the reciprocal operation of multiplication is a less frequent arithmetic operation used in the processors and it is more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0. The basic long division algorithm is explained using division hardware Figure 3.9 [7] and a division algorithm Figure 3.10 [7]. It is assumed that both the dividend and the divisor are positive and hence the quotient and the remainder are non-negative. The arbitrary precision library also assumes that the dividend and the divisor are unsigned numbers and hence there is no chance for any overflow due to the division arithmetic. The arbitrary precision divider library module has been implemented with logic such that it selects its own hardware register sizes based on the library data width. Hence to do the operations on the arbitrary precision width data, the library does its own internal arithmetic.

The algorithm for any other arbitrary bit width division (say 64 / 64, 96 / 96) is a modification to the basic division algorithm given in Figure 3.10, which is a variant of the longhand division algorithm as discussed in [7]. The algorithm below is for 32 / 32 bit.
The above division algorithm is discussed for a word size of 32-bits. The operation can be explained with the help of an example:

11 divided by 3: 11 (1011) is dividend and 3 (0011) is divider.

```
  1011-
-0011
  -------- 0 Difference is negative: copy dividend and put 0 in quotient.
```

```
  1011-
-0011
  -------- 00 Difference is negative: copy dividend and put 0 in quotient.
```
1011-
-0011

"""" 001 Difference is positive: use difference and put 1 in quotient.

0101-
-0011

"""" 0011 Difference is positive: use difference and put 1 in quotient.

10

So the Quotient is, 3 (0011); the remainder is 2 (10).

This previous given algorithm is extended for an arbitrary precision library width with the help of an arbitrary precision data divider which can hold all the data chunks in its hardware registers whose size is based on the library width. In general, both the dividend and the divisor inputs are made the same size (n bits) and both the quotient and remainder registers are made equal to the size of the dividend (n bits) so that they can hold the outputs after division. The algorithm for arbitrary precision division is as given in Figure 3.11.
Start
First Loop:
Start
Keep on reading the values from input RAM to the dividend register and the divider register for all the data blocks
End
Main Loop:
First Time:
Start:
Move the values from the dividend register to the dividend copy based on the library width and do the bit adjustments as required
Move the values from the divider register to the divider copy based on the library width and do the bit adjustments as required
Bit = library width
Quotient = 0
End
Else:
Start
diff = dividend copy - divider copy;
If diff < 0
    Keep dividend copy the same
    Move quotient one bit left and put 0 in quotient bit 0
Else
    dividend copy = diff
    Move quotient one bit left and put 1 in quotient bit 0
End if
Shift the divider copy right by 1 bit
Go to Main Loop till Bit becomes 0
Bit = Bit - 1
End
Keep Reading the Values from the Arbitrary Precision Divider

Figure 3.11: Arbitrary precision division algorithm
The arbitrary precision divider module for the previous division logic has also been designed and hand coded in Verilog. The arbitrary precision divider is not directly connected to the library wrapper; it is connected to the output RAM to read the dividend and the divider blocks into its own hardware registers. This is made to ensure more security in the system. The final quotient and remainder generated by the arbitrary precision divider can be read by the application code directly using a pin named ‘div_out_for_test’. It has to be read after a specified number of clock cycles (as specified in the Data Manual provided in the Appendix) to get the final quotient and remainder. Also the value from the pin ‘div_out_for_test’ is written to the output RAM after the final quotient and remainder has been generated.

### 3.4.3.2 Division block diagram

![Division block diagram](image-url)
Figure 3.12 shows the division block diagram and the associated hardware which had been used for arbitrary precision division. Addition hardware arbitrary precision division has been implemented in Verilog and the outputs Quotient and Remainder are stored as output variables from the algorithm. In the arbitrary precision data divider module, the size of quotient and remainder registers are made equal to the size of dividend input to avoid any overflow. For more details on the working of the arbitrary precision Division please see the Data Manual section in Appendix.

### 3.5 Number of clock cycles required for the arithmetic operations

Table 3.1: Number of clock cycles taken

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>CLOCKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>(2 \times n + 3)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>(2 \times n + 1)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>(2 \times n \times \text{(library width)})</td>
</tr>
<tr>
<td>Division</td>
<td>(2 \times n + (\text{library width}) + 2)</td>
</tr>
</tbody>
</table>

Here \(n = \frac{\text{library width}}{\text{data width}}\)

All the clock counters are started from the place where the respective op code is selected by the application code to trigger the library. Addition takes two more clock cycles than subtraction since for an unsigned addition, the 'Carry Bit' also has to be read from the Output RAM.
Chapter 4

4  Hardware diagrams, experimental results, test bench and the annotated waveforms

This is the results chapter and it is divided into three sections. The first section shows and explains the hardware diagrams extracted for the arbitrary precision library when the library has been synthesized onto an FPGA.

4.1  Hardware diagrams

Hardware Diagrams for the arbitrary precision integer library can be exported from the RTL viewer (Netlist Viewer) tool in the Quartus II tool (Tools -> Netlist Viewers -> RTL Viewer). This section shows the various hardware diagrams generated for the arbitrary precision library. The arbitrary precision library is compiled and synthesized onto a Cyclone II FPGA device which uses the hand coded RAM module. The hardware diagrams of the library synthesized onto any other FPGA device will look the same, even the ones which use Altera LPM RAM modules. Screenshots are taken for a 96 bit precision arbitrary precision integer library.

4.1.1  Arbitrary precision library: main hardware

Figure 4.1 and Figure 4.2 are the front end hardware diagrams of the arbitrary precision library. The front end hardware diagram had to be split into two figures since it is not possible to make a screenshot fit into a single page. Figure 4.1 is annotated with the Input RAM hardware, Output RAM hardware, 32 Bit Structural Ripple Adder hardware, Unsigned Accumulator hardware and the Altera unsigned Multiplier Accumulator hardware. Figure 4.2 is annotated with the arbitrary precision data divider hardware. Please note that the hardware diagram will look the same even if the implementation uses the Altera LPM RAM and on any FPGA device other than Cyclone II. The hardware diagrams for each and every module are given in detail in this section which includes RAMs, multiplier, divider, adder etc.
Figure 4.1: Arbitrary precision library front end hardware 1
Figure 4.2: Arbitrary precision library front end hardware 2
4.1.2 Altera unsigned multiplier accumulator

Given below is the hardware diagram for the Altera Unsigned Multiplier Accumulator Module. This module is only used for 32 * 32 bit multiplication. Please note that the input buses for data A and data B are 32 bits wide and the output data bus is 64 bits wide.

![Figure 4.3: Altera unsigned multiplier accumulator](image)

4.1.3 Output dualportRAM

Given below is the hardware diagram for the Output DualportRAM Module. Please note that the address bus is 10 bits wide and both the input and output data buses are 32 bits wide.

![Figure 4.4: Output dualportRAM](image)
4.1.4 Input dualportRAM

Given below is the hardware diagram for the Input DualportRAM Module. Please note that both the address buses for A and B is 10 bits wide and both the input and output data buses for A and B are 32 bits wide.

Figure 4.5: Input dualportRAM
### 4.1.5 Arbitrary precision data divider

Figure 4.6 and 4.7 are the hardware diagrams for the arbitrary precision data divider. The diagram has to be split into two since it is not possible to make a screenshot fit into a single page. This hardware keeps growing as the size of the library increases. The figures which are kept in this report are for the library of size 96 bits. Please note that the quotient output and the remainder output have the same data width as the dividend register. Also the remainder output from the module is a wire taken from an internal register, dividend copy register.

### 4.1.6 Unsigned arbitrary precision accumulator

Figure 4.8 and 4.9 are the hardware diagrams for the unsigned arbitrary precision accumulator. This hardware keeps growing as the size of the library is kept increasing. The figures which are kept in this report are for the library of size 96 bits. The module has its own internal clock counter register in addition to the global clock counter register. Please note that ‘adder_out’ is the main data output from this module and this data will be the processed input values, shifted accordingly and added afterwards. We can see a dividing logic which essentially does all the shifting.
Figure 4.6: Arbitrary precision data divider hardware 1
Figure 4.7: Arbitrary precision data divider hardware 2
Figure 4.8: Unsigned arbitrary precision accumulator hardware 1
Figure 4.9: Unsigned arbitrary precision accumulator hardware 2
4.2 Experimental results

In this section, the compilation results from the Quartus II CAD tool have been collected into different tables. The parameters considered for the experimental results are area, power and time. The arbitrary precision library has been put on three different FPGA devices: Cyclone II, Arria II GX and Cyclone V. The results are collected for three different library widths (32 bit, 64 bit and 96 bit). Also the results are taken for two different design approaches: one for the RAM module which is hand coded in Verilog and other for Altera LPM RAM module.

Table 4.1: Summary table for experiments

<table>
<thead>
<tr>
<th>Devices Used</th>
<th>Cyclone II</th>
<th>Arria II GX</th>
<th>Cyclone V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Area</td>
<td>Power</td>
<td>Time</td>
</tr>
<tr>
<td>Library Sizes</td>
<td>32 Bit</td>
<td>64 Bit</td>
<td>96 Bit</td>
</tr>
<tr>
<td>Implementation</td>
<td>Hand Coded RAM</td>
<td>Altera LPM RAM</td>
<td></td>
</tr>
</tbody>
</table>

4.2.1 Area

This section compares the area occupied by the arbitrary precision library on different FPGA devices. The first section is for the RAM memory module which has been hand coded. The second section is for the Altera LPM RAM module which has been taken from the Altera website. The third section is the comparison study done after plotting graphs.

4.2.1.1 Hand coded RAM

Here is an example Flow Summary report from the Quartus II CAD Tool which is for a 32 bit library width.
Figure 4.10: Example flow summary report for hand coded RAM

The reports for 64 bit and 96 bits have the same fields and in this thesis we have collected only the parameters which are valid for comparison. One point to note is the 8 Embedded Multiplier 9-bit elements which are selected for 32 * 32 bit multiplication. This is specific to the library width of 32 bits since the 32 bit multiplication internally uses the Altera unsigned multiplier accumulator module. Other elements which are common for any library sizes are:

![Insert table here]

Figure 4.11: Common area elements for hand coded RAM

Given next is the table which compares the area occupied by the arbitrary precision library on the chip for the three FPGA families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):
<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>FPGA</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLONE II</td>
<td>Total Logic Elements</td>
<td>845 / 18,752 (5 %)</td>
<td>2,106 / 18,752 (11 %)</td>
<td>4,599 / 18,752 (25 %)</td>
</tr>
<tr>
<td></td>
<td>Total Combinational Functions</td>
<td>845 / 18,752 (5 %)</td>
<td>1,968 / 18,752 (10 %)</td>
<td>4,381 / 18,752 (23 %)</td>
</tr>
<tr>
<td></td>
<td>Dedicated Logic Registers</td>
<td>306 / 18,752 (2 %)</td>
<td>875 / 18,752 (5 %)</td>
<td>1,198 / 18,752 (6 %)</td>
</tr>
<tr>
<td></td>
<td>Total pins: 309/315 (98 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cyclone II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRIA II GX</td>
<td>Combinational ALUTs</td>
<td>474 / 36,100 (1 %)</td>
<td>1,114 / 36,100 (3 %)</td>
<td>3,149 / 36,100 (9 %)</td>
</tr>
<tr>
<td></td>
<td>Dedicated Logic Registers</td>
<td>306 / 36,100 (1 %)</td>
<td>875 / 36,100 (2 %)</td>
<td>1,199 / 36,100 (3 %)</td>
</tr>
<tr>
<td></td>
<td>Total pins: 309/404 (76 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cyclone V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Combinational ALUTs</td>
<td>519 / 112,960 (1 %)</td>
<td>1,143 / 112,960 (1 %)</td>
<td>3,130 / 112,960 (3 %)</td>
</tr>
<tr>
<td></td>
<td>Dedicated Logic Registers</td>
<td>306 / 225,920 (1 %)</td>
<td>875 / 225,920 (1 %)</td>
<td>1,198 / 225,920 (1 %)</td>
</tr>
<tr>
<td></td>
<td>Total pins: 309/378 (82 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.2.1.2 Altera LPM RAM

Here is an example compilation report from the Quartus II CAD Tool which is for a 32 bit library width.

![Flow Summary](image)

Figure 4.12: Example flow summary report for Altera LPM RAM
The reports for 64 bit and 96 bits have the same fields and in this thesis we have collected only the parameters which are valid for comparison. One point to note is the 8 Embedded Multiplier 9-bit elements which are selected for 32 * 32 bit multiplication. This is specific to the library width of 32 bits since the 32 bit multiplication internally uses the Altera unsigned multiplier accumulator module. Other elements which are common for any library sizes are:

![Common area elements for Altera LPM RAM](image)

Figure 4.13: Common area elements for Altera LPM RAM

Given next is the table which compares the area occupied in the chip for the three families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):

Table 4.3: Area comparison for Altera LPM RAM

<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLONE II</td>
<td>Total Logic Elements</td>
<td>845 / 18,752 (5%)</td>
<td>2,067 / 18,752 (11%)</td>
</tr>
<tr>
<td>Total pins: 309/315 (98%)</td>
<td>Total Combinational Functions</td>
<td>845 / 18,752 (5%)</td>
<td>1,936 / 18,752 (10%)</td>
</tr>
<tr>
<td></td>
<td>Dedicated Logic Registers</td>
<td>305 / 18,752 (2%)</td>
<td>852 / 18,752 (5%)</td>
</tr>
<tr>
<td>ARRIA II GX</td>
<td>Combinational ALUTs</td>
<td>472 / 36,100 (1%)</td>
<td>1,117 / 36,100 (3%)</td>
</tr>
<tr>
<td>Total pins: 309/404 (76%)</td>
<td>Dedicated Logic Registers</td>
<td>305 / 36,100 (&lt;1%)</td>
<td>852 / 36,100 (2%)</td>
</tr>
<tr>
<td>CYCLONE V</td>
<td>Combinational ALUTs</td>
<td>517 / 112,960 (&lt;1%)</td>
<td>1,114 / 112,960 (&lt;1%)</td>
</tr>
<tr>
<td>Total pins: 309/378 (82%)</td>
<td>Dedicated Logic Registers</td>
<td>305 / 225,920 (&lt;1%)</td>
<td>852 / 225,920 (&lt;1%)</td>
</tr>
</tbody>
</table>
4.2.1.3 Graphs, comparisons and conclusions for area

In this section, a comparison study has been done for the ‘area’ parameter between the two different design approaches (hand coded RAM vs. Altera LPM RAM) for three different devices: Cyclone II, Arria II GX and Cyclone V. Graphs have been made from the Table 4.2 and 4.3 for comparisons. These are shown in Figures 4.14 – 4.20.

FPGA: Cyclone II

Figure 4.14: Total logic elements for FPGA: Cyclone II

Figure 4.15: Total combinational functions for FPGA: Cyclone II
Figure 4.16: Dedicated logic registers for FPGA: Cyclone II

FPGA: Arria II GX

Figure 4.17: Combinational ALUTs for FPGA: Arria II GX

Figure 4.18: Dedicated logic registers for FPGA: Arria II GX
Comparison and Conclusion

From the above graphs, we can conclude that the design based on hand coded RAM occupied more area on all the three chips than the Altera specific LPM RAM for all the area parameters. And the previous condition is true for all the three different arbitrary precision library widths. Hence if a user wants to utilize less space on the chip, it is advisable to use the Altera specific LPM RAM module.
4.2.2 Power

This section compares the power consumed on different FPGA devices. The first section is for the RAM memory module which has been hand coded. The second section is for the Altera LPM RAM module which has been taken from the Altera website. The third section is the comparison study done after plotting graphs.

4.2.2.1 Hand coded RAM

Here is an example PowerPlay Power Analyzer Tool report from the Quartus II CAD Tool which is for a 32 bit library width.

![Figure 4.21: Example PowerPlay Power Analyzer tool report for hand coded RAM](image)

The table given next compares the Power consumed in the chip for the three families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):
Table 4.4: Power comparison for hand coded RAM

<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CYCLONE II</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>172.51 mW</td>
<td>236.31 mW</td>
<td>265.24 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>20.12 mW</td>
<td>25.37 mW</td>
<td>62.35 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>47.54 mW</td>
<td>47.66 mW</td>
<td>47.71 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>104.85 mW</td>
<td>163.28 mW</td>
<td>155.17 mW</td>
</tr>
<tr>
<td><strong>ARRIA II GX</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>366.95 mW</td>
<td>363.10 mW</td>
<td>365.09 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>4.42 mW</td>
<td>5.79 mW</td>
<td>6.85 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>317.35 mW</td>
<td>317.30 mW</td>
<td>317.32 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>45.19 mW</td>
<td>40.00 mW</td>
<td>40.91 mW</td>
</tr>
<tr>
<td><strong>CYCLONE V</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>161.48 mW</td>
<td>154.22 mW</td>
<td>155.78 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>3.19 mW</td>
<td>3.85 mW</td>
<td>5.09 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>100.66 mW</td>
<td>100.60 mW</td>
<td>100.61 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>57.63 mW</td>
<td>49.77 mW</td>
<td>50.08 mW</td>
</tr>
</tbody>
</table>

4.2.2.2 *Altera LPM RAM*

Here is an example PowerPlay Power Analyzer Tool report from the Quartus II CAD Tool which is for a 32 bit library width.

![PowerPlay Power Analyzer Summary](image)

Figure 4.22: Example PowerPlay Power Analyzer tool report for Altera LPM RAM
The table given next compares the Power consumed in the chip for the three families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):

<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLONE II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>314.91 mW</td>
<td>351.69 mW</td>
<td>455.00 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>31.40 mW</td>
<td>32.79 mW</td>
<td>74.23 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>47.80 mW</td>
<td>47.87 mW</td>
<td>48.07 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>235.71 mW</td>
<td>271.02 mW</td>
<td>332.71 mW</td>
</tr>
<tr>
<td>ARRIA II GX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>690.11 mW</td>
<td>827.31 mW</td>
<td>775.60 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>34.94 mW</td>
<td>40.85 mW</td>
<td>53.13 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>320.91 mW</td>
<td>322.44 mW</td>
<td>321.86 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>334.26 mW</td>
<td>464.02 mW</td>
<td>400.61 mW</td>
</tr>
<tr>
<td>CYCLONE V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>215.16 mW</td>
<td>227.59 mW</td>
<td>211.66 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>6.49 mW</td>
<td>7.25 mW</td>
<td>8.09 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>101.14 mW</td>
<td>101.25 mW</td>
<td>101.10 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>107.53 mW</td>
<td>119.09 mW</td>
<td>102.46 mW</td>
</tr>
</tbody>
</table>

4.2.2.3 **Graphs, comparisons and conclusions for power**
In this section, a comparison study has been done for the ‘power’ parameter between the two different design approaches (hand coded RAM vs. Altera LPM RAM) for three different devices: Cyclone II, Arria II
GX and Cyclone V. Graphs have been made from the Tables 4.4 and 4.5 for comparisons and are shown in Figures 4.23 – 4.34.

FPGA: Cyclone II

**Figure 4.23:** Total thermal power dissipation for FPGA: Cyclone II

**Figure 4.24:** Core dynamic thermal power dissipation for FPGA: Cyclone II
Figure 4.25: Core static thermal power dissipation for FPGA: Cyclone II

Figure 4.26: I/O thermal power dissipation for FPGA: Cyclone II

FPGA: Arria II GX

Figure 4.27: Total thermal power dissipation for FPGA: Arria II GX
Figure 4.28: Core dynamic thermal power dissipation for FPGA: Arria II GX

Figure 4.29: Core static thermal power dissipation for FPGA: Arria II GX

Figure 4.30: I/O thermal power dissipation for FPGA: Arria II GX
FPGA: Cyclone V

Figure 4.31: Total thermal power dissipation for FPGA: Cyclone V

Figure 4.32: Core dynamic thermal power dissipation for FPGA: Cyclone V
Comparison and Conclusion

From the above graphs, we can conclude that the design based on hand coded RAM consumed less power on all the three chips than the Altera specific LPM RAM for all the power parameters. And the previous condition is true for all the three different arbitrary precision library widths. Hence if a user wants to utilize minimize the power consumption on the chip, it is advisable to use the hand coded RAM module.
4.2.3 Time

This section compares the frequency of operation of the library and other timing parameters on different FPGA devices. The first section is for the RAM memory module which has been hand coded. The second section is for the Altera LPM RAM module which has been taken from the Altera website. The third section is the comparison study done after plotting graphs.

4.2.3.1 Hand coded RAM

Here is an example TimeQuest Timing Analyzer Tool report from the Quartus II CAD Tool which is for a 32 bit library width.

![Figure 4.35: Example TimeQuest Timing Analyzer tool frequency report for hand coded RAM](image)

Figure 4.35 shows the assigned frequency of operation for a 32 bit library size as 25 MHz which cannot exceed the Fmax frequency as given in the Table 4.6.
Figure 4.36 shows the Multicorner Timing Analysis Summary done by the TimeQuest Timing Analyzer Tool and the worst-case slack is calculated for set-up, hold, recovery, removal and minimum pulse width.

The Table given next compares the frequency of operation and other timing parameters of the library for the three families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):
<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLONE II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>40</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>47.74</td>
<td>46.18</td>
<td>10.23</td>
</tr>
<tr>
<td>Setup</td>
<td>4.053</td>
<td>3.344</td>
<td>2.238</td>
</tr>
<tr>
<td>Hold</td>
<td>0.215</td>
<td>0.215</td>
<td>0.215</td>
</tr>
<tr>
<td>Recovery</td>
<td>20.249</td>
<td>21.278</td>
<td>95.931</td>
</tr>
<tr>
<td>Removal</td>
<td>2.109</td>
<td>1.957</td>
<td>2.106</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>9.936</td>
<td>9.936</td>
<td>47.436</td>
</tr>
<tr>
<td>ARRIA II GX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>100</td>
<td>100</td>
<td>18.18</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>103.93</td>
<td>116.88</td>
<td>18.55</td>
</tr>
<tr>
<td>Setup</td>
<td>0.378</td>
<td>1.444</td>
<td>1.103</td>
</tr>
<tr>
<td>Hold</td>
<td>0.143</td>
<td>0.138</td>
<td>0.137</td>
</tr>
<tr>
<td>Recovery</td>
<td>4.245</td>
<td>3.865</td>
<td>50.011</td>
</tr>
<tr>
<td>Removal</td>
<td>2.266</td>
<td>2.842</td>
<td>2.165</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>3.834</td>
<td>3.829</td>
<td>26.328</td>
</tr>
<tr>
<td>CYCLONE V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>66.67</td>
<td>100</td>
<td>18.18</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>82.96</td>
<td>102.93</td>
<td>18.87</td>
</tr>
<tr>
<td>Setup</td>
<td>2.892</td>
<td>0.285</td>
<td>1.181</td>
</tr>
<tr>
<td>Hold</td>
<td>0.089</td>
<td>0.089</td>
<td>0.116</td>
</tr>
<tr>
<td>Recovery</td>
<td>8.448</td>
<td>4.217</td>
<td>48.389</td>
</tr>
<tr>
<td>Removal</td>
<td>2.948</td>
<td>2.734</td>
<td>3.305</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>6.346</td>
<td>3.859</td>
<td>26.349</td>
</tr>
</tbody>
</table>
4.2.3.2 Altera LPM RAM

Here is an example TimeQuest Timing Analyzer Tool report from the Quartus II CAD Tool which is for a 32 bit library width.

![Figure 4.37: Example TimeQuest Timing Analyzer tool frequency report for Altera LPM RAM](image)

Figure 4.37 shows the assigned frequency of operation for a 32 bit library size as 25 MHz which cannot exceed the Fmax frequency as given in the Table 4.7.
Figure 4.38: Example TimeQuest Timing Analyzer tool timing analysis report for Altera LPM RAM

Figure 4.38 shows the Multicorner Timing Analysis Summary done by the TimeQuest Timing Analyzer Tool and the worst-case slack is calculated for set-up, hold, recovery, removal and minimum pulse width.

Given next is the Table which compares the frequency of operation of the library for the three families Cyclone II, Arria II GX, Cyclone V) for three library sizes (32 bit, 64 bit and 96 bit):
<table>
<thead>
<tr>
<th>LIB SIZE</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>96 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLONE II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>40</td>
<td>40</td>
<td>13.33</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>53.92</td>
<td>54.21</td>
<td>13.82</td>
</tr>
<tr>
<td>Setup</td>
<td>6.453</td>
<td>6.554</td>
<td>2.658</td>
</tr>
<tr>
<td>Hold</td>
<td>0.215</td>
<td>0.215</td>
<td>0.215</td>
</tr>
<tr>
<td>Recovery</td>
<td>19.845</td>
<td>21.460</td>
<td>70.975</td>
</tr>
<tr>
<td>Removal</td>
<td>2.297</td>
<td>1.904</td>
<td>2.081</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>9.936</td>
<td>9.936</td>
<td>34.936</td>
</tr>
<tr>
<td>ARRIA II GX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>100</td>
<td>111.11</td>
<td>20</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>104.14</td>
<td>127.26</td>
<td>21.54</td>
</tr>
<tr>
<td>Setup</td>
<td>0.398</td>
<td>1.142</td>
<td>3.581</td>
</tr>
<tr>
<td>Hold</td>
<td>0.142</td>
<td>0.136</td>
<td>0.097</td>
</tr>
<tr>
<td>Recovery</td>
<td>3.163</td>
<td>4.612</td>
<td>45.213</td>
</tr>
<tr>
<td>Removal</td>
<td>2.830</td>
<td>1.971</td>
<td>2.146</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>3.829</td>
<td>3.329</td>
<td>23.828</td>
</tr>
<tr>
<td>CYCLONE V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>111.11</td>
<td>111.11</td>
<td>20.0</td>
</tr>
<tr>
<td>F Max (MHz)</td>
<td>131.94</td>
<td>121.45</td>
<td>22.2</td>
</tr>
<tr>
<td>Setup</td>
<td>1.363</td>
<td>0.715</td>
<td>4.310</td>
</tr>
<tr>
<td>Hold</td>
<td>0.077</td>
<td>0.070</td>
<td>0.084</td>
</tr>
<tr>
<td>Recovery</td>
<td>2.473</td>
<td>3.053</td>
<td>43.372</td>
</tr>
<tr>
<td>Removal</td>
<td>2.997</td>
<td>2.836</td>
<td>3.260</td>
</tr>
<tr>
<td>Minimum Pulse Width</td>
<td>3.357</td>
<td>3.357</td>
<td>23.855</td>
</tr>
</tbody>
</table>
4.2.3.3 **Graphs, comparisons and conclusions for time**

In this section, comparison study has been done for the ‘time’ parameter between the two different design approaches (hand coded RAM vs. Altera LPM RAM) for three different devices: Cyclone II, Arria II GX and Cyclone V. Graphs have been made from the Tables 4.6 and 4.7 for comparisons. They are shown in Figures 4.39 – 4.56.

**FPGA: Cyclone II**

![Figure 4.39: Maximum frequency for FPGA: Cyclone II](image)

![Figure 4.40: Setup time for FPGA: Cyclone II](image)

![Figure 4.41: Hold time for FPGA: Cyclone II](image)
Figure 4.42: Recovery time for FPGA: Cyclone II

Figure 4.43: Removal time for FPGA: Cyclone II

Figure 4.44: Minimum pulse width for FPGA: Cyclone II

FPGA: Arria II GX

Figure 4.45: Maximum frequency for FPGA: Arria II GX
Figure 4.46: Setup time for FPGA: Arria II GX

Figure 4.47: Hold time for FPGA: Arria II GX

Figure 4.48: Recovery time for FPGA: Arria II GX

Figure 4.49: Removal time for FPGA: Arria II GX
Figure 4.50: Minimum pulse width for FPGA: Arria II GX

FPGA: Cyclone V

Figure 4.51: Maximum frequency for FPGA: Cyclone V

Figure 4.52: Setup time for FPGA: Cyclone V

Figure 4.53: Hold time for FPGA: Cyclone V
Comparison & Conclusion

From the above graphs, we can conclude that the design based on Altera specific LPM RAM runs at a higher frequency on all the three chips compared to a hand coded RAM. And the previous condition is true for all the three different arbitrary precision library widths. Hence if a user wants to maximize the operating frequency of the chip, it is advisable to use the Altera specific LPM RAM. The other parameters are put in Table 4.8 for convenience since they depend on the device onto which the library
is put into and on the library widths too. The corresponding RAM is put against the device based on the maximum parameter value.

Table 4.8: Table based on timing parameters

<table>
<thead>
<tr>
<th></th>
<th>Cyclone II</th>
<th>Arria II GX</th>
<th>Cyclone V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 64 96</td>
<td>32 64 96</td>
<td>32 64 96</td>
</tr>
<tr>
<td>Set Up</td>
<td>LPM</td>
<td>LPM</td>
<td>LPM</td>
</tr>
<tr>
<td>Hold</td>
<td>SAME</td>
<td>SAME</td>
<td>HCRAM</td>
</tr>
<tr>
<td>Recovery</td>
<td>SAME</td>
<td>SAME</td>
<td>HCRAM</td>
</tr>
<tr>
<td>Removal</td>
<td>LPM</td>
<td>HCRAM</td>
<td>LPM</td>
</tr>
<tr>
<td>Min Pulse</td>
<td>SAME</td>
<td>SAME</td>
<td>HCRAM</td>
</tr>
<tr>
<td>Width</td>
<td>HCRAM</td>
<td>SAME</td>
<td>HCRAM</td>
</tr>
</tbody>
</table>

* HCRAM: Hand coded RAM, LPM: Altera LPM, SAME: same for both Hand coded and LPM RAM

4.3 Test bench and the annotated waveforms

In this section, the arbitrary precision integer arithmetic library test results are shown as the annotated exported waveforms. Also the test strategy which had been used for testing the four arithmetic operations is explained. The arbitrary precision library is synthesized onto the Cyclone II device with the hand coded RAM module design and this architecture is verified for its functionality. The library has been tested for three different library sizes: 32 bit, 64 bit and 96 bit.

From the Quartus II Tool, the library modules can be exported (Tools -> Run Simulation Tool -> RTL Simulation) and tested with the ModelSim Altera Starter Edition 10.0 d simulator using an appropriate test bench. We can have the test bench also as a part of the entire library project by doing the
appropriate settings in the Quartus II Tool: Settings -> EDA Tool Settings -> Simulation -> Test Benches. The previous setting helps to launch the ModelSim simulator from the Quartus II and runs the test bench on its own.

Please note that before the library starts its operation by selecting the appropriate op code, all the blocks for the input operands have to be clocked into the input RAM module, 32 bits every positive clock edge. Input RAM can only be written one port every positive clock edge even if it has dual write ports. Also the test bench has to take care of setting the write enable signal and the proper write addresses for the corresponding operands which has to be stored in the input RAM. Once all the values are stored in the input RAM, the test bench can assert the library to start its operation by selecting the appropriate op code. Also the test bench has to give the library, the corresponding address that has to be read from the input RAM every positive clock edge. Please refer to the Data Manual in the Appendix which explains in detail the usage of the library for each and every op code.

Given next are the annotated waveforms and the test strategies which explain how the library has been tested for the four different op codes.

4.3.1 ADDITION

This section explains in detail how the addition operation has been tested for three different library widths: 32, 64 and 96.

4.3.1.1 Library size 32 bits

4.3.1.1.1 No CARRY, no OVERFLOW:

The first test case is to test the basic 32 bit addition operation without any carry or any overflow. In this case, the sign bit is not overwritten by the addition operation. Operands A and B are selected
accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, '0000' is selected by the test bench.

Given inputs:

OPERAND A = 32'b000000001000000010000000100000001

OPERAND B = 32'b000000001000000010000000100000001

Expected Outputs:

SUM = 32'b00000010000000100000001000000010

CARRY = 1'b0

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the 'add_sub_tb_overflow' pin not being set indicating there was no overflow.

Figure 4.57: Addition 32 bit; no CARRY, no OVERFLOW

4.3.1.1.2. With CARRY, with OVERFLOW:
The second test case is to test the 32 bit addition operation with an overflow. In this case, the sign bit will be overwritten by the addition operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, ‘0000’ is selected by the test bench. Here negative operands are selected and the sign bit ‘1’ is overwritten with a ‘0’ after the addition so that it is the example of an overflow.

Given Inputs:
OPERAND A = 32'b10000001000000010000000100000001
OPERAND B = 32'b10000001000000010000000100000001

Expected Outputs:
SUM = 32'b00000010000000100000001000000010
CARRY = 1'b1

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow.

Figure 4.58: Addition 32 bit; with CARRY, with OVERFLOW
4.3.1.2 Library size 64 bits

4.3.1.2.1. No CARRY, with OVERFLOW:

The first test case is to test the basic 64 bit addition operation without any carry but with an overflow. In this case, the sign bit (which is the leftmost bit in SUM Block 1) is overwritten by the addition operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, ‘0000’ is selected by the test bench. Here positive operands are selected and the sign bit ‘0’ is overwritten with a ‘1’ after the addition so that it is the example of an overflow.

Given Inputs:

OPERAND A

Block 1                      Block 0
010000010000000100000000100000000    10000001000000010000000100000001

OPERAND B

Block 1                      Block 0
01000001000000010000000100000000    10000001000000010000000100000001

Expected Outputs:

Block 1                      Block 0
SUM
10000010000000100000001000000001    00000010000000100000001000000010
CARRY = 1'b0

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow.
4.3.1.2.2. With CARRY, with OVERFLOW:

The second test case is to test the 64 bit addition operation with carry and with an overflow. In this case, the sign bit (which is the leftmost bit in SUM Block 1) is overwritten by the addition operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, ‘0000’ is selected by the test bench. Here negative operands are selected and the sign bit ‘1’ is overwritten with a ‘0’ after the addition so that it is the example of an overflow. Also it generates a carry bit since it adds both the sign ‘1’ bits.

Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000001000000010000000100000000</td>
<td>00000001000000010000000100000001</td>
</tr>
</tbody>
</table>

OPERAND B

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000001000000010000000100000000</td>
<td>00000001000000010000000100000001</td>
</tr>
</tbody>
</table>
Expected Outputs:

SUM

00000010000001000000010000000000  0000001000000100000001000000010

CARRY = 1'b1

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow.

4.3.1.3 Library size 96 bits

4.3.1.3.1. No CARRY, no OVERFLOW:

The first test case is to test the basic 96 bit addition operation without any carry and without an overflow. In this case, the sign bit is not overwritten by the addition operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, ‘0000’ is selected by the test bench.
Given Inputs:

OPERAND A

Block 2
00000001000000010000000100000001

Block 1
00000001000000010000000100000001

Block 0
00000001000000010000000100000000

OPERAND B

Block 2
00000001000000010000000100000001

Block 1
00000001000000010000000100000001

Block 0
00000001000000010000000100000000

Expected Outputs:

SUM

Block 2
00000001000000010000000100000001

Block 1
00000001000000010000000100000001

Block 0
00000001000000010000000100000001

CARRY = 1'b0

SUM Waveform: The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the three blocks for the result SUM.
Figure 4.61: Addition 96 bit; SUM BLOCKS, no CARRY, no OVERFLOW

Carry Waveform: The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow.

Figure 4.62: Addition 96 bit; CARRY BIT and OVERFLOW FLAG, no CARRY, no OVERFLOW
4.9.1.3.2. With CARRY, with OVERFLOW:

The second test case is to test the 96 bit addition operation with carry and with an overflow. In this case, the sign bit (which is the leftmost bit in SUM Block 1) is overwritten by the addition operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for ADD, ‘0000’ is selected by the test bench. Here negative operands are selected and the sign bit ‘1’ is overwritten with a ‘0’ after the addition so that it is the example of an overflow. Also it generates a carry bit since it adds both the sign ‘1’ bits.

Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000010000000100000001</td>
<td>100000010000000100000001</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>100000010000000100000001</td>
<td></td>
</tr>
</tbody>
</table>

OPERAND B

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000010000000100000001</td>
<td>100000010000000100000001</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>100000010000000100000001</td>
<td></td>
</tr>
</tbody>
</table>

Expected Outputs:

SUM

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000010000000100000001000000011</td>
<td>00000010000000100000001000000001</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>00000010000000100000001000000010</td>
<td></td>
</tr>
</tbody>
</table>

CARRY = 1'b1
SUM Waveform: The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the three blocks for the result SUM.

![Image of SUM Waveform](image)

Figure 4.63: Addition 96 bit; SUM BLOCKS, with CARRY, with OVERFLOW

Carry Waveform: The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow.

![Image of Carry Waveform](image)

Figure 4.64: Addition 96 bit; CARRY BIT and OVREFLOW FLAG, with CARRY, with OVERFLOW
4.3.2 SUBTRACTION

This section explains in detail how the subtraction operation has been tested for three different library widths: 32, 64 and 96.

4.3.2.1 Library size 32 bits

4.3.2.1.1 No OVERFLOW (positive number - positive number = negative number):

The first test case is to test the basic 32 bit subtraction operation without overflow. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, B is selected greater than A and hence we expect a negative result after subtraction (with the sign bit set, 1’b1).

Given Inputs:

OPERAND A = 32'b0000000000000000000000000100000 (32)
OPERAND B = 32'b00000000000000000000000001000001 (65)

Expected Output:

DIFFERENCE = 32'b11111111111111111111111111011111 (-33)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 1 indicating it is a negative output.
4.3.2.1.2 With OVERFLOW (negative number - positive number = positive number):

The second test case is to test the 32 bit subtraction operation with an overflow. In this case, the sign bit is overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, '0001' is selected by the test bench. In this case, A is a negative number and B is a positive number and the magnitudes of both are selected such that the subtraction operation results in an overflow. Hence we get a positive number after subtraction (instead of a negative number) which is an overflow and we expect the sign bit not to be set, 1'b0.

Given Inputs:

OPERAND A = 32'b10000000000000000000000000000000 (-2147483648)
OPERAND B = 32'b01111111111111111111111111111111 (2147483647)
Actual Output:
A large negative number, -4294967295 which needs more than 32 bits to store and its binary representation are: 64'b111111111111111111111111111111110000000000000000000000000001

Expected Library Output:
DIFFERENCE = 32'b00000000000000000000000000000001

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 0 indicating it is a positive output.

Figure 4.66: Subtraction 32 bit; with OVERFLOW

4.3.2.1.3 No OVERFLOW (number - zero = same number):

The third test case is to test the 32 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the
test bench. In this case, A is a positive number and B is a zero. Hence we get expect the same number after subtraction.

Given Inputs:

OPERAND A = 32'b00000000000000000000001111101000 (1000)
OPERAND B = 32'b00000000000000000000000000000000 (0)

Expected Library Output:

DIFFERENCE = 32'b00000000000000000000001111101000 (1000)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow.

Figure 4.67: Subtraction 32 bit special case 1; no OVERFLOW
4.3.2.1.4 No OVERFLOW (number – same number = zero):

The fourth test case is to test the 32 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ’0001’ is selected by the test bench. In this case, A is a positive number and B is the same number. Hence we get expect a zero after subtraction.

Given Inputs:
OPERAND A = 32'b00000000000000000000001111101000 (1000)
OPERAND B = 32'b00000000000000000000001111101000 (1000)

Expected Library Output:
DIFFERENCE = 32'b00000000000000000000000000000000 (0)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow.
4.3.2.2  Library size 64 bits

4.3.2.2.1. No OVERFLOW (positive number – positive number = positive number):

The first test case is to test the basic 64 bit subtraction operation without overflow. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is selected greater than B and hence we expect a positive result after subtraction (with the sign bit not set, 1'b0).

Given Inputs:

OPERAND A

Block 1                                                                 Block 0
01111111111111111111111111111111 00000000000000000000000000100000
(922372032559808544, a large number which need 64 bits for storage)

OPERAND B

Block 1                                                                 Block 0
00000000000000000000000000000001 00000000000000000000000001000001
(4294967361, another large number which need 64 bits for storage)

Expected Library Output:

DIFFERENCE

Block 1                                                                 Block 0
01111111111111111111111111111101 11111111111111111111111111011111
(9223372028264841183, the result which can be represented in 64 bits)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being not set indicating
there was no overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 0 indicating a positive output.

![Figure 4.69: Subtraction 64 bit; no OVERFLOW (positive – positive = positive)](image)

4.3.2.2.2. No OVERFLOW (positive number – positive number = negative number):

The second test case is to test the basic 64 bit subtraction operation without overflow and with the input operands switched. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, B is selected greater than A and hence we expect a negative result after subtraction (with the sign bit set, 1’b1).

Given Inputs:

OPERAND A
Block 1                          Block 0
00000000000000000000000000000001                00000000000000000000000001000001
(4294967361, a large number which need 64 bits for storage)
OPERAND B

Block 1
01111111111111111111111111111111
(9223372032559808544, another large number which need 64 bits for storage)

Block 0
00000000000000000000000000100000

Expected Library Output:

DIFFERENCE

Block 1
10000000000000000000000000000010
(-9223372028264841183, the result which can be represented in 64 bits)

Block 0
00000000000000000000000000100001

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 1 indicating a negative output.

Figure 4.70: Subtraction 64 bit; no OVERFLOW (positive – positive = negative)
4.3.2.2.3. With OVERFLOW (positive number – negative number = negative number):

The third test case is to test the 64 bit subtraction operation with an overflow. In this case, the sign bit is overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is a negative number and the magnitudes of both are selected such that the subtraction operation results in an overflow. Hence we get a negative number after subtraction (instead of a positive number) which is an overflow and we expect the sign bit to be set, 1’b1.

Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000001</td>
<td>00000000000000000000000001000001</td>
</tr>
</tbody>
</table>

(4294967361, a large number which need 64 bits for storage)

OPERAND B

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000000000000000000000000000</td>
<td>11111111111111111111111111100000</td>
</tr>
</tbody>
</table>

(-9223372032559808544, another large number which need 64 bits for storage)

Actual Output:

9223372036854775905 which needs more than 64 bits to store

Expected Library Output:

DIFFERENCE

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000000000000000000000000000</td>
<td>00000000000000000000000001100001</td>
</tr>
</tbody>
</table>

(-9223372036854775711, which is the result of an Overflow)
The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating there has been an overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 1 indicating a negative output.

![Waveform Image]

Figure 4.71: Subtraction 64 bit; with OVERFLOW

4.3.2.2.4. No OVERFLOW (number - zero = same number):

The fourth test case is to test the 64 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is a zero. Hence we get expect the same number after subtraction.
Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000001</td>
<td>00000000000000000000000001000001</td>
</tr>
</tbody>
</table>

(4294967361, a large number which need 64 bits for storage)

OPERAND B

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000</td>
<td>00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

(0)

Expected Library Output:

DIFFERENCE

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000001</td>
<td>00000000000000000000000001000001</td>
</tr>
</tbody>
</table>

(4294967361, the same number)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the 'add_sub_tb_overflow' pin not being set indicating there was no overflow.
4.3.2.2.5. No OVERFLOW (number – same number = zero):

The fifth test case is to test the 32 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is the same number. Hence we get expect a zero after subtraction.

Given Inputs:

OPERAND A

Block 1                                      Block 0
00000000000000000000000000000001                00000000000000000000000001000001

(4294967361, a large number which need 64 bits for storage)
OPERAND B

Block 1

00000000000000000000000000000001

(4294967361, the same number which need 64 bits for storage)

Expected Library Output:

DIFFERENCE

Block 1

00000000000000000000000000000000

(0)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the 'add_sub_tb_overflow' pin not being set indicating there was no overflow.

Figure 4.73: Subtraction 64 bit special case 2; no OVERFLOW
4.3.2.3 Library size 96 bits

4.3.2.3.1. No OVERFLOW (positive number – positive number = positive number):

Please note that to do the 96 bit operations, the online calculator has been downloaded from [29]
The first test case is to test the basic 96 bit subtraction operation without overflow. In this case, the sign
bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are
clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test
bench. In this case, A is selected greater than B and hence we expect a positive result after subtraction
(with the sign bit not set, 1’b0).

Given Inputs:

OPERAND A

Block 2                               Block 1
01111111111111111111111111111111    00000000000000000000000000100000

Block 0

00000000000000000000000001000001

(39614081238685424860501377089, a large number which need 96 bits for storage)

OPERAND B

Block 2                               Block 1
00000000000000000000000000000001    00000000000000000000000000100000

Block 0

00000000000000000000000001000001

(18446744352882425860, another large number which need 96 bits for storage)
Expected Library Output:

**DIFFERENCE**

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111111111111111111111111101</td>
<td>11111111111111111111111111011111</td>
</tr>
</tbody>
</table>

Block 0

00000000000000000000000000111101

(39614081220238680507618951229, the result which can be represented in 96 bits)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being not set indicating there was no overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 0 indicating a positive output.

![Waveform](image)

**Figure 4.74: Subtraction 96 bit; no OVERFLOW (positive – positive = positive)**

4.3.2.3.2. No OVERFLOW (positive number – positive number = negative number):

The second test case is to test the basic 96 bit subtraction operation without overflow and with the input operands switched. In this case, the sign bit is not overwritten by the subtraction operation.
Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, B is selected greater than A and hence we expect a negative result after subtraction (with the sign bit set, 1'b1).

Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000001</td>
<td>00000000000000000000000000000100</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000001000001</td>
<td></td>
</tr>
</tbody>
</table>

(18446744352882425860, a large number which need 96 bits for storage)

OPERAND B

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111111111111111111111111111</td>
<td>00000000000000000000000000100000</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000001000001</td>
<td></td>
</tr>
</tbody>
</table>

(39614081238685424860501377089, another large number which need 96 bits for storage)

Expected Library Output:

DIFFERENCE

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000000000000000000000000010</td>
<td>00000000000000000000000000000100</td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>11111111111111111111111111111111</td>
<td></td>
</tr>
</tbody>
</table>

(-39614081220238680507618951229, the result which can be represented in 96 bits)
The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 1 indicating a negative output.

**Figure 4.75:** Subtraction 96 bit; no OVERFLOW (positive – positive = negative)

### 4.9.2.3.3. With OVERFLOW (positive number – negative number = negative number):

The third test case is to test the 96 bit subtraction operation with an overflow. In this case, the sign bit is overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is a negative number and the magnitudes of both are selected such that the subtraction operation results in an overflow. Hence we get a negative number after subtraction (instead of a positive number) which is an overflow and we expect the sign bit to be set, 1'b1.
Given Inputs:

OPERAND A

Block 2                                                                 Block 1
01111111111111111111111111111111               00000000000000000000000000100000

Block 0
00000000000000000000000000000001

(39614081238685424860501377089, a positive large number which need 96 bits for storage)

OPERAND B

Block 2                                                                 Block 1
10000000000000000000000000000000               11111111111111111111111111011111

Block 0
11111111111111111111111111111111

(-39614081238685424860501377089, a negative large number which need 96 bits for storage)

Actual Output:

79228162477370849721002754178, which needs more than 96 bits to store

Expected Library Output:

DIFFERENCE

Block 2                                                                 Block 1
11111111111111111111111111111110                00000000000000000000000001000000

Block 0
00000000000000000000000010000010

(-36893487872541196158, a negative number which is the result of an Overflow)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin being set indicating
there has been an overflow. Also note the sign bit in the DIFFERENCE (MSB, leftmost bit) which is a bit 1 indicating a negative output.

Figure 4.76: Subtraction 96 bit; with OVERFLOW

4.9.2.3.4. No OVERFLOW (number - zero = same number):

The fourth test case is to test the 96 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is a zero. Hence we get expect the same number after subtraction.

Given Inputs:

OPERAND A

<table>
<thead>
<tr>
<th>Block 2</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000001</td>
<td>000000000000000000000000000000000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000000000000000000000001000000</td>
</tr>
</tbody>
</table>

Given

99
(18446744352882425860, a large number which need 96 bits for storage)

OPERAND B

Block 2                              Block 1
00000000000000000000000000000000    00000000000000000000000000000000

Block 0
00000000000000000000000000000000

(0)

Expected Library Output:

DIFFERENCE

Block 2                              Block 1
00000000000000000000000000000001    00000000000000000000000000000100

Block 0
00000000000000000000000000000100

(18446744352882425860, the same number)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow.
Figure 4.77: Subtraction 96 bit special case 1; no OVERFLOW

4.9.2.3.5. No OVERFLOW (number – same number = zero):

The fifth test case is to test the 96 bit subtraction operation with another test case. In this case, the sign bit is not overwritten by the subtraction operation. Operands A and B are selected accordingly and are clocked into the input RAM as discussed above. Then the op code for SUB, ‘0001’ is selected by the test bench. In this case, A is a positive number and B is the same number. Hence we get expect a zero after subtraction.

Given Inputs:

OPERAND A

Block 2

00000000000000000000000000000001

Block 1

00000000000000000000000001000001

Block 0

000000000000000000000100

(1844674435282425860, a large number which need 96 bits for storage)

OPERAND B

Block 2

00000000000000000000000000000001

Block 1

00000000000000000000000001000001

101
The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. We can see the ‘add_sub_tb_overflow’ pin not being set indicating there was no overflow.

Figure 4.78: Subtraction 96 bit special case 2; no OVERFLOW
4.3.3 MULTIPLICATION

As we pointed out earlier, there is no overflow scenario for multiplication since the output register will always be twice the size of the input operand so that it can hold the entire product. Also the multiplication works only with the unsigned integers. Hence the testing of multiplication does not need to cover the overflow scenarios and for this reason, each and every library widths need only one test case per library.

4.3.3.1 Library size 32 bits

The test case is to test the 32 bit multiplication is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for MULT, ‘0010’ is selected by the test bench. As we discussed earlier, 32 bit multiplication does not take many clock cycles as other library widths since it uses the Altera multiplier accumulator module for an increased performance. Also for 32 bit multiplication, the products are written to the Output RAM and can be read back from them.

Given Inputs:

OPERAND A = 32'b10000000000000000000000000000011 (2147483651)

OPERAND B = 32'b10101010101010101010101010101010 (2863311530)

Expected Output:

PRODUCT

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010101010101010101010101010110</td>
<td>11111111111111111111111111111110</td>
</tr>
</tbody>
</table>

(6148914698394796030, a large number which needs 64 bits for storage)
The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start getting the results from the output RAM after four clock cycles, from the fifth positive clock edge onwards.

![Waveform Diagram]

**4.3.3.2 Library size 64 bits**

The test case is to test the 64 bit multiplication is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for MULT, '0010' is selected by the test bench. The test bench has to take care of the input RAM address for A and B on every alternate clock edges to read from as per the multiplication algorithm. Also for 64 bit multiplication, the Output RAM is busy to hold all the intermediate products which are then fed onto the accumulator for respective shifting and adding. So there is an output pin provided ‘acc_out_for_test’ using which the final product blocks can be read. The pin ‘acc_out_for_test’ will start reading the values from the 7\(^{th}\) clock \((2 \times ((\text{library width}) / (\text{data width}))) + 3\) onwards after the op code ‘0010’ has been selected by the test bench. From there, it
will take 256 \((4*64, 2 * ((\text{library width}) / (\text{data width})) * \text{library width})\) clock cycles to complete the multiplication operation. The final product will have \(2 * 64 = 128\) bits and hence we have to read 4 data blocks from the output pin as the final product. So the application code can start to accept the values from the pin ‘acc\_out\_for\_test’ after 253 \((4 * 64 - 4 + 1, (2 * ((\text{library width}) / (\text{data width})) * \text{library width}) - (2 * ((\text{library width}) / (\text{data width})) + 1))\) clock cycles.

Given Inputs:

**OPERAND A**

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111111111111111111111111111</td>
<td>000000000000000000000000100000</td>
</tr>
</tbody>
</table>

(922372032559808544, a large number which need 64 bits for storage)

**OPERAND B**

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000000000000000000001000011</td>
<td>100000000000000000000000100011</td>
</tr>
</tbody>
</table>

(922372039002259523, another large number which need 64 bits for storage)

**Expected Library Output:**

**DIFFERENCE**

<table>
<thead>
<tr>
<th>Block 3</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111111111111111111111111111111</td>
<td>11000000000000000000000000110000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111111111111111111111101</td>
<td>0000000000000000000000010000110000</td>
</tr>
</tbody>
</table>

(85070591710427576141168026852280764512, the result which can be represented in 128 bits)

The waveform given next is the ModelSim simulator output waveform captured for the input operands storage in the input RAM and the start point of the algorithm with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start
getting the intermediate results from the pin ‘acc_out_for_test_tb’ after 7 clock cycles, and to get the final product, we have to wait for 253 clock cycles.

The waveform given next is the ModelSim simulator output waveform captured for the final outputs with values annotated for reading convenience. As shown in the figure, after waiting for 253 clock cycles, we start getting the final products from the pin ‘acc_out_for_test_tb’ and the four product blocks are as shown in the figure.

**Figure 4.80: Multiplication 64 bit input operands**
4.3.3.3 Library size 96 bits

The test case is to test the 96 bit multiplication is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for MULT, ‘0010’ is selected by the test bench. The test bench has to take care of the input RAM address for A and B on every alternate clock edges to read from as per the multiplication algorithm. Also for 96 bit multiplication, the Output RAM is busy to hold all the intermediate products which are then fed onto the accumulator for respective shifting and adding. So there is an output pin provided ‘acc_out_for_test’ using which the final product blocks can be read. The pin ‘acc_out_for_test’ will start reading the values from the 9th clock \((2 \times ((\text{library width}) / (\text{data width})) +3)\) onwards after the op code ‘0010’ has been selected by the test bench. From there, it will take 576 \((6 \times 96, (2 \times ((\text{library width}) / (\text{data width})) \times \text{library width}))\) clock cycles to complete the multiplication operation. The final product will have \(2 \times 96 = 192\) bits and hence we have to read 6 data blocks from the output pin as the final product. So the application code can start to accept the values from the pin ‘acc_out_for_test’ after 571 \(6 \times 96 - 6 + 1, (2 \times ((\text{library width}) / (\text{data width})) \times \text{library width}) - (2 \times ((\text{library width}) / (\text{data width})) + 1))\) clock cycles.
Given Inputs:

OPERAND A

Block 2
01111111111111111111111111111111
Block 1
00000000000000000000000000010000
Block 0
00000000000000000000000000010000

(39614081238685424791781900320, a large number which need 96 bits for storage)

OPERAND B

Block 2
10000000000000000000000000000001
Block 1
00000000000000000000000001000001
Block 0
00000000000000000000000000010000

(39614081275578913149654401089, another large number which need 96 bits for storage)

Expected Library Output:

PRODUCT

Block 5
00000000000000000000100000100000
Block 4
01000000000000000000000000000000
Block 3
00000000000000000000000000000001
Block 2
01111111111111111111111111111111
Block 1
01000000000000110000110000
Block 0
00000000000000000000110000110000

(1569275433846670204400100849139371848147624981499897448480, the result which can be represented in 192 bits)

The waveform given next is the ModelSim simulator output waveform captured for the input operands storage in the input RAM and the start point of the algorithm with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start
getting the intermediate results from the pin ‘acc_out_for_test_tb’ after 9 clock cycles, and to get the final product, we have to wait for 571 clock cycles.

Figure 4.82: Multiplication 96 bit input operands

The waveform given next is the ModelSim simulator output waveform captured for the final outputs with values annotated for reading convenience. As shown in the figure, after waiting for 571 clock cycles, we start getting the final products from the pin ‘acc_out_for_test_tb’ and the six product blocks are as shown, split into two figures.
Figure 4.83: Multiplication 96 bit first three product blocks

Figure 4.84: Multiplication 96 bit, last three product blocks
4.3.4 DIVISION

As we pointed out earlier, there is no overflow scenario division since the quotient and remainder registers will always be the size of the dividend and divider inputs so that it can hold the results. Also the division works only with the unsigned integers. Hence the testing of division does not need to cover the overflow scenarios and for this reason, each and every library widths need only one test case per library.

4.3.4.1 Library size 32 bits

The test case is to test the 32 bit division is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for DIV, ‘0011’ is selected by the test bench. Also the test bench has to disable the clock enable signal for the arbitrary precision divider after 34 clock cycles ((library width) + 2 * ((library width) / (data width))) and the output RAM is read back from that clock onwards. Quotient block is the first to read from the output RAM and remainder block follows.

Given Inputs:

OPERAND A = 32'b10000000000000000000000000000000   (2147483648)

OPERAND B = 32'b00000000000000000000000000000101     (5)

Expected Library Outputs:

QUOTIENT   = 32'b00011001100110011001100110011001 (429496729)

REMAINDER = 32'b00000000000000000000000000000011 (3)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start getting the results from the output RAM after 34 clock cycles.
4.3.4.2 Library size 64 bits

The test case is to test the 64 bit division is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for DIV, ‘0011’ is selected by the test bench. Also the test bench has to disable the clock enable signal for the arbitrary precision divider after 70 clock cycles \(((\text{library width}) + 2 * ((\text{library width}) / (\text{data width})) + 2)\) and the output RAM is read back from that clock onwards. Quotient block is the first to read from the output RAM and remainder block follows.

Given Inputs:

OPERAND A

Block 1

\(00000000000000000000000000000001\)

Block 0

\(00000000000000000000000000000010\)

(4294967298, a large number which need 64 bits for storage)

OPERAND B

Block 1

\(00000000000000000000000000000000\)

Block 0

\(00000000000000000000000000000010\)
Expected Library Outputs:

**QUOTIENT**

Block 1  
00000000000000000000000000000000 
Block 0  
01000000000000000000000000000000 

(1073741824)

**REMAINDER**

Block 1  
000000000000000000000000000000000 
Block 0  
00000000000000000000000000000010 

The waveform given next is the ModelSim simulator output waveform captured for the input operands storage and with values annotated for reading convenience. Also the algorithm start point is annotated in the graph.

![Figure 4.86: Division 64 bit, input operands storage](image)

The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start getting the results from the output RAM after 70 clock cycles.
4.3.4.3 Library size 96 bits

The test case is to test the 96 bit division is given here. After the Operands A and B are clocked into the input RAM as discussed above, the op code for DIV, ‘0011’ is selected by the test bench. Also the test bench has to disable the clock enable signal for the arbitrary precision divider after 104 clock cycles (\(((\text{library width}) + 2 \times ((\text{library width}) / (\text{data width})) + 2)\) and the output RAM is read back from that clock onwards. Quotient block is the first to read from the output RAM and remainder block follows.

Given Inputs:

OPERAND A

Block 2

| 00000100000000000000000000000000 |

Block 1

| 00000000000000000000000000000101 |

Block 0

| 00000000000000000000000000000000101 |

(12379400392853802748991229, a large number which need 96 bits for storage)
OPERAND B

Block 2
00000000000000000000000001000000
Block 1
00000000000000000000000000000000
Block 0
00000000000000000000000000000000

(1180591620717411303424, another large number which need 96 bits for storage)

Expected Library Output:

QUOTIENT

Block 2
00000000000000000000000000000000
Block 1
00000000000000000000000000000000
Block 0
00000000000000000000000000000000

(1048576)

REMAINDER

Block 2
00000000000000000000000000000000
Block 1
00000000000000000000000000000000
Block 0
00000000000000000000000000000000

(5)

The waveform given next is the ModelSim simulator output waveform captured for the input operands storage and with values annotated for reading convenience. Also the algorithm start point is annotated in the graph.
The waveform given next is the ModelSim simulator output waveform captured and with values annotated for reading convenience. As shown in the figure, after the op code has been selected by the test bench, we start getting the results from the output RAM after 104 clock cycles. The quotient and remainder results are split into two graphs for convenience.
This completes the section for the test bench and the annotated waveforms.

This completes the results chapter for the arbitrary precision integer arithmetic library.
Chapter 5

5 Conclusion and future work

This thesis is an arbitrary precision integer arithmetic library implementation in hardware and it is one of the first ones that belong to this genre. Although there are some VLSI chips in the industry which can do some encryption algorithms like RSA or AES, there are no hardware solutions for a generic arbitrary precision library and that was the starting idea of this work. The library is hand coded in Verilog HDL for an arbitrary precision using the algorithms designed and later on, compiled and synthesized onto an FPGA device from Altera. Even if this work has been an FPGA implementation, this can be extended to full custom ASIC logic as well, since the library is hand coded in Verilog. So this work is like an open source hardware solution which can suit ASICs or FPGAs. As the library is an integer library, there is no floating point support offered, since almost all the floating point operations are done by keeping the decimal point separate, doing the math on the integer blocks and finally adjusting the decimal point in the result. Another point to note is the algorithms involved in doing the integer arithmetic operations and at this moment, they are the basic ones. Addition uses basic ripple carry adders, subtraction uses the basic 2’s complement method, multiplication uses the basic pen and paper method and division algorithm is the extended one from the basic 32 bit division. The basic algorithms are only selected because this is a proof of concept that such an arbitrary precision library can have a hardware solution. The library has been tested for three design parameters, area, power and time for three different library sizes (32, 64 and 96 bit) on three different Altera FPGA families: Cyclone II, Arria II GX and Cyclone V. Then a comparison study has been done for the design parameters for two different design architectures, one with a hand coded Verilog RAM module and another with an Altera specific Library of Parameterized Module RAM. Finally the algorithms for all four operations are tested with test benches
which have test cases which cover all the corner cases of the algorithms and it has been shown that the algorithms can work on an FPGA for arbitrary precision logic.

There are some more aspects which can be researched as a part of this thesis. First is an extension to a floating point arithmetic library. Even if many of the public key cryptography algorithms may not need the floating point arithmetic at all, there might be some scenarios where we may need them. In future if a user would like to use a library which can support them, then this library can be extended with the decimal point manipulation feature and made onto a floating point arithmetic library as well. The second point is with respect to the efficiency of the algorithms which had been used in this library. All the four algorithms can be replaced with suitable algorithms which are faster to this implementation. As an example, the ripple carry adders can be replaced with carry look ahead adders which can make the addition and subtraction operation faster. As of now, the area occupied by the hardware on the chip is reduced by lowering the speed of operation of the chip since the ripple carry adders won’t take much space compared with the carry look ahead adders. However the speed of operation is in the MHz range and since the implementation was on an FPGA that does not matter much. However for an ASIC design, the clock frequencies can be still high and in that case, we might have to think about algorithms that can run faster. That is another future extension to this library.
Bibliography


APPENDIX

Library usage manual: how to configure & use the multiple precision libraries

This section discusses the use of the arbitrary precision library from an application programmer’s perspective. This section has two separate parts for the user: 1) How to write values to the input RAM and 2) How to set the proper op code to read the output RAM and interpret the results.

A.1 How to write the input RAM

The application code should write both the operands operand A and operand B to the input RAM before the library wrapper is clocked for doing the respective arithmetic operation. The input RAM has to be written by clocking the RAM and asserting the write enable signal for each data write port. The input RAM is a True Dual Port RAM with a single clock. Figure A.1 gives the clock timing diagram which explains how to write to the input RAM. Please note that the input RAM write timing diagram is for a library width of 64 bits. Since the data chunk size of the library is 32 bits, the operand A will have 2 blocks (2*32 bits) and operand B will have 2 blocks (2*32 bits). These blocks have to be serially loaded into the input RAM memory with the clock signal provided by the application code to the library. The library clock is tied to the clock input of the input RAM so that the RAM write can be synchronized with the application which triggers the library. Also the write addresses of the ports for A and B to write to the input RAM can be selected by the application program. It is suggested to follow a linear order in addressing for the operands A and B. Example, for a 64 bit library we have to write A 2 times * 32bit and B 2 time * 32bit where the address port is 6 bit wide (as of now in the implementation it is fixed to 6 bits, it may be changed later). This is how the operand A and operand B have to be arranged in memory:
Table A-1: Assignment of input blocks

<table>
<thead>
<tr>
<th></th>
<th>Block 1 (32 bits)</th>
<th>Block 0 (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Address 1</td>
<td>Address 0</td>
</tr>
<tr>
<td>B</td>
<td>Address 3</td>
<td>Address 2</td>
</tr>
</tbody>
</table>

Since the address port is 6 bits wide, the LSB block of A, block 0 will go to the address 6'b000000 and the next block of A, that is block 1 will go to the address 6'b000001. The LSB block of B, block 0 will go to the address 6'b000010 and the next block of B, which is block 1 will go to the address 6'b000011. Hence in general if there are ‘n’ blocks for an operand to write to the input RAM, the first ‘n’ blocks of address in the Input RAM are reserved for the first and the next ‘n’ blocks of address in the Input RAM are reserved for the second operand (assuming that, to perform an arithmetic operation, we need two operands).

A.2 Input RAM write timing diagram for a 64 bit library

Figure A.1 describes the input RAM write timing diagram for a 64 bit library. It shows how the application code should enable the write enable signals for the two ports A and B of the input RAM. Also please note the way in which the input RAM write address and the write data are provided.
A.3 How to set the proper op code, read the output RAM and interprets the results

The four bit op codes for different arithmetic operations supported by the library are: Addition: 0000, Subtraction: 0001, Multiplication: 0010, Division: 0011.

After the input RAM has been written with data for operands A and B, the application code should start clocking the library for doing the respective arithmetic operation by setting the ‘op code’ signal. The number of clock cycles required varies with the op code that the application requires the library to do. The table given next gives the number of clock cycles taken for doing the four different arithmetic operations. That means the number of clock cycles required for completely reading all the output blocks after performing the library operation.
Table A-2: Number of clock cycles taken

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>CLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>$2n+3$</td>
</tr>
<tr>
<td>SUB</td>
<td>$2n+1$</td>
</tr>
<tr>
<td>MUL</td>
<td>$2 \times n \times \text{(library width)}$</td>
</tr>
<tr>
<td>DIV</td>
<td>$2 \times n + \text{(library width)} + 2$</td>
</tr>
</tbody>
</table>

Here $n = \frac{\text{(library width)}}{\text{(data width)}}$

After the op code has been clocked into the library, the library starts reading from the input RAM with the read address given by the application code. The two data ports in the True Dual Port Input RAM can be read in the same clock cycle with two addresses provided by the application. In this case, the application program is supposed to know the address of the ports to read from the input RAM and this address should be in the order in which the input RAM has been written in the write stage. Since the input RAM has been written with data blocks in a linear method for the two operands, the read from the input RAM also follows the same order. As per our previous 64 bit example, to read the two blocks for two different operands from the input RAM we have to read from the addresses 6'b000000 for A and 6'b000010 for B on this high clock edge. In general, if there are ‘n’ blocks for an operand to read from the input RAM, on the first high clock edge, the 0th block of address in the input RAM is read to get the first block of operand A and the nth block of address in the input RAM is read to get the first block of operand B. On the next high even clock edge, the 1st block of address in the input RAM is read to get the second block of operand A and the (n+1)st block of address in the Input RAM is read to get the second block of operand B and so on. The read input RAM operation is completed in 2n such clock cycles. It takes more than n clock cycles since the reading from input RAM and writing result to the output RAM cannot happen in a single clock cycle due to timing issues. The ways that reads and writes are handled in the library are as discussed below. There are four different actions out of which one has to be performed by the library on each and every high clock edge. The input RAM read (to get the two operands), performing the arithmetic operation (to calculate the results), the output RAM write (to store the results) and the output RAM read (to read the results and present to the application code).
The input RAM read, performing the operation and the output RAM read are done on even clock edges (0, 2, 4 etc.) and the output RAM write is done on odd clock edges (1, 3, 5 etc.). The output RAM read has an extra latency of a clock cycle since the RAM has to be written before it is read for results. Hence the Output RAM reads will happen only from the 2\textsuperscript{nd} clock cycle onwards. In short, for a 64 bit library where the aspect factor, \(\frac{\text{library width}}{\text{data width}} = \frac{64}{32} = 2\), it takes a total of \(2*2+3\) clock cycles for the entire ADD arithmetic library operation (if we count the Input RAM write clock cycles as well, then a total of \(2*2+2*2+3\) clock cycles) to complete. We see some extra latency since we have to read the Carry Bit as well from the output RAM in case of a positive number system. In general if \(n\) is \(\frac{\text{library width}}{\text{data width}}\), then it takes a total of \(2*n+3\) clock cycles (if we have to count Input RAM write clock cycles as well, then \(2*2*n+3\) clock cycles) for an ADD operation. With the previous explanation it will be easy to understand an example timing diagram for a 64 bit ADD operation, reading of the input data from the input RAM and finally the reading of the results from the Output RAM w.r.t the clock cycle and the op code as shown in Figure A.2:

![Figure A.2: 64 bit addition timing diagram](image-url)
As shown in the waveforms in Figure A.2, the op code 4'b0000 is for the ADD operation and on the third positive clock edge, the valid SUM data is read back to the application from the output result RAM. On the (2*n+3 rd) clock cycle the carry bit is read from the output RAM and read by the application.

For subtraction: The timing is same as addition except for a small change: we need only 2*n+1 clock cycles since we do not need to read the Carry Bit as a part of subtraction. Please note that the subtraction is assumed to work with a 2’s complement number system. Hence the timing waveforms look the same as Figure A.1 and Figure A.2.

For multiplication: The input RAM writes timing is the same here. However when we start clocking the library with the op code, the proper input RAM read address has to be selected by the application code. This can be explained with the following diagrams for a 64 bit MULT operation, reading of the input data from the input RAM and finally the reading of the results from the Output RAM w.r.t the clock cycle and the op code.

Diagrams A.3 and A.4 show the way in which the input RAM has to be addressed.

**Figure A.3: 64 bit multiplication; input RAM addressing part 1**
Figure A.4: 64 bit multiplication; input RAM addressing part 2

Diagram A.5 shows the output pin ACC OUT; the pin starts to read the final product from the 253rd clock cycle onwards. Please note that the count 253 is started from the clock edge in which the 7th CLK is marked as in the Figure given. The product blocks are as marked below.

Figure A.5: 64 bit multiplication; output pin read timing

To generalize for any library width, the output pin ACC OUT will start reading the values from the \((2 \times (\text{library width}) / (\text{data width}) + 3)\) clock edge onwards after the op code ‘0010’ has been selected by the application code and these are the intermediate results. From there, it will take \((2 \times ((\text{library width}) / (\text{data width})) \times (\text{library width}))\) clock cycles to complete the multiplication operation. The final product will have \(2 \times (\text{library width})\) bits and hence we have to read \(2 \times (\text{library width}) / (\text{data width})\) blocks from the output pin as the final product. So the application code can start to accept the values from the pin
ACC OUT after \((2 * (\text{library width}) / (\text{data width}) \times \text{library width}) - (2 * (\text{library width}) / (\text{data width}) + 1)\) clock cycles.

For division: The input RAM writes timing is the same here as addition. So is the way the application code addresses the input RAM for reading the operands. The point to note for division is the clock edge to expect the quotient value and the remainder value to be read back from the output RAM. Also on the same edge the application code has to disable the clock enable signal for the arbitrary precision divider module.

Diagram A.6 shows the clock edge to read the final product from the output RAM. Please note that the count 70 is marked at the clock edge in which clock enable signal is disabled. The quotient and remainder blocks are as marked below.

Figure A.6: 64 bit division; output RAM read timing

To generalize for any library width, the output RAM can start giving the results, \((\text{library width} + 2 \times (\text{library width}) / (\text{data width}) + 2)\) clock edges onwards after the op code ‘0011’ has been selected by the application code. From there, it will take \(2 * (\text{library width}) / (\text{data width})\) clock cycles to complete the
division operation. The final result (quotient and remainder) will have $2 \times$ (library width) bits in total and hence we have to read $2 \times \frac{\text{library width}}{\text{data width}}$ blocks from the output RAM.

### A.4 Number systems supported by the library

Arbitrary precision integer arithmetic library support both signed and unsigned number systems.

Unsigned number system: For n bits, its range is $[0, (2^n) – 1]$. It can represent only the positive numbers and zero.

Signed number system: Two’s complement system is used for storing negative numbers. For n bits, its range is $[-2^{(n-1)}, (2^{(n-1)}) -1]$.

Table A.3 shows the arithmetic operations and the number systems the library support.

<table>
<thead>
<tr>
<th>Op code</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Not Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Division</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

### A.5 Overflow handling

This section describes the way in which the library handles arithmetic overflow. The term arithmetic overflow or simply overflow has the following meanings.

1. In a machine, the condition that arises when a result produced by calculation is greater in magnitude than that which a given register or storage location can store or represent.

2. In a machine, the amount by which a calculated value is greater in magnitude than that which a given register or storage location can store or represent. In this case another location might be used to hold the overflow.
The two kinds of overflow conditions can be distinguished in almost all the machines. When the result of an addition or subtraction, considering the operands and result as unsigned numbers, does not fit in the result register or the storage, a carry occurs. Hence, after adding or subtracting numbers that are interpreted as unsigned values it is useful to check the carry flag. For signed numbers, the approach is different. An overflow proper occurs when the result does not have the sign that one would predict from the signs of the operands (e.g. a negative result when adding two positive numbers). Therefore, it is useful to check the overflow flag after adding or subtracting numbers that are represented in two’s complement form (i.e., they are considered signed numbers).

The following section discusses the way in which the arbitrary precision library handles the various overflow cases.

A.5.1 Addition

An extra pin ‘add_sub_overflow’ is added in the arbitrary precision library for detecting the overflow which can occur in addition and subtraction. The application code should read this pin and do the operations accordingly. Also the interpretation of the results depends on the number system which is being used by the application.

Unsigned Number System: For addition, the ‘add_sub_overflow’ pin can be discarded. This is because there is no sign overflow in an unsigned number system addition. Overflow in this scenario is handled by the carry bit. The carry bit is read back from the output RAM (as shown in the timing diagram Figure A.2) by the application code.

Signed Number System: For a signed number system, before reading the carry bit from the RAM, the application has to consider the ‘add_sub_overflow’ signal. The overflow rule for addition in two’s complement number system is used to infer the output data. If two two's complement numbers are added, and they both have the same sign (either positive or both negative), then overflow occurs if and
only if the result has the opposite sign. Overflow never occurs when adding operands with different signs. That is, a) Adding two positive numbers must give a positive result for no overflow and b) Adding two negative numbers must give a negative result for no overflow. Overflow occurs if (I) \((+A) + (+B) = -C\) or (II) \((-A) + (-B) = +C\).

The below table discusses the various cases that can occur while adding two two’s complement numbers in the library.

<table>
<thead>
<tr>
<th>CASES</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive number + positive number</td>
<td>(MSB 0) with the ‘add_sub_overflow’ signal not set. This has to be interpreted as no overflow and the carry bit read here will always be zero (that means the actual result is positive)</td>
</tr>
<tr>
<td>negative number + negative number</td>
<td>(MSB 0) with the ‘add_sub_overflow’ signal set. This has to be interpreted as an overflow and the carry bit read here is the sign bit overflowed which will be one (that means the actual result is negative)</td>
</tr>
<tr>
<td>positive number + negative number</td>
<td>(MSB 0) with the add_sub_overflow signal not set. This means there is no overflow and the application should check the carry bit</td>
</tr>
</tbody>
</table>
The previous table helps an application programmer to interpret the results as a result of ADD from the library accordingly.

### A.5.2 Subtraction

Please note that the subtraction is implemented with the two’s complement number system (see Table 4.3). Two’s complement subtraction is normally accomplished by negating the subtrahend (what is being subtracted) and adding it to the minuend (what it is being subtracted from). The overflow rule for Subtraction states: If two two's complement numbers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend.

Overflow occurs if

- \((+A) - (-B) = -C\)
- \((-A) - (+B) = +C\)

The table given next discusses the various cases that can occur while subtracting two two’s complement numbers in the library.
Table A-5: All cases for subtraction

<table>
<thead>
<tr>
<th>CASES</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive number – positive</td>
<td>(MSB 0) with the ‘add_sub_overflow’ signal not set. This has to be interpreted as no overflow</td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>negative number – negative</td>
<td>(MSB 0) with the ‘add_sub_overflow’ signal not set. This has to be interpreted as no overflow</td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>positive number – negative</td>
<td>(MSB 0) with the add_sub_overflow signal not set. This has to be interpreted as no overflow</td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>negative number – positive</td>
<td>(MSB 0) with the add_sub_overflow signal set. This has to be interpreted as an overflow</td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

A.5.3 Multiplication and division

Multiplication and division in the library support only the unsigned number system. Also care has been taken to make the output registers large enough to hold the products, quotient and the remainder.

Hence there won’t be any overflow scenario for these two operations.

This completes the Library Usage Manual.