A Pipelined, Single Precision Floating-Point Logarithm Computation Unit in Hardware

A Pipelined, Single Precision Floating-Point Logarithm Computation Unit in Hardware

By
Jing Chen B.Eng
IBM®Center for Advanced Studies Fellow
A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University
© Copyright by Jing Chen, July 31, 2012

MASTER OF SCIENCE(2012) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: A Pipelined, Single Precision Floating-Point Logarithm Computation Unit in Hardware

AUTHOR: Jing Chen B.Eng(Capital Normal University)

EMAIL: jingchen_cs@yahoo.com

SUPERVISOR: Dr. Christopher K. Anand

NUMBER OF PAGES: xi, 54

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

A large number of scientific applications rely on the computing of logarithm. Thus, accelerating the speed of computing logarithms is significant and necessary. To this end, we present the realization of a pipelined Logarithm Computation Unit (LCU)[1] in hardware that uses lookup table and interpolation techniques. The presented LCU supports single precision arithmetic with fixed accuracy and speed. We estimate that it can generate 2.9G single precision values per second under a 65nm fabrication process. In addition, the accuracy is at least 21 bits while lookup table size is about 7.776KB. To the best of our knowledge, our LCU achieves the fastest speed at its current accuracy and table size.

[1]This work is funded by the IBM Center for Advanced Studies

Acknowledgments

I would like to thank my supervisor Dr. Christopher Anand, for the trust and flexibility he gave me, which allowed me to do this project on my own f scratch.

I would like to thank Robert Enenkel from the IBM Toronto Lab, for the literature he gave me, which motivated me for this project.

I would also like to thank my parents, for their never ending support and encouragement.

Contents

List of Figures

List of Tables

Glossary

ALU	Arithemtic Logic Unit is an important component of a CPU. It is used to perform arithmetic and logic operations, 9
CPU	Central Processing Unit is an important hardware component within a computer system. It is used to execute instructions, 9
Cygwin	Cygwin is a collection of tools which provide a Linux look and feel environment for Windows [18], 19
D flip-flop	A flip-flop is a memory storage element. It has two states 0 or 1, 3
DSP slice	Digital Signal Processor slice, it is used to accomodate arithmetic and logic computations. It invovles lookup tables, multipliers, etc, 6
GNU	GNU is a Unix-like operating system that is free software [19], 5
IP block	An IP block refers to those configurable hardware resources of an FPGA chip (i.e. multiplier, embedded memory). The idea of an IP block is similar to the subprograms of software libraries, 6
MOSFET	Metal Oxide Semiconductor Field Effect Transistor. MOSFET is one of the basic components for a large scale integrated circuit design, 45

RAM	Random Access Memory, one can both read from or write to a RAM by specifing an address, 43
ROM	Read Only Memory, one can only read the content of a ROM by specifing an address, 43
SRAM	Static Random Access Memory, it consists of transistors and data which is stored in SRAM is maintainced as long as power is on, 43

Chapter 1

Introduction

1.1 Motivation

A large number of scientific codes rely on computing the natural logarithm. In [1], Vinyals and Friedland state many multimedia applications which require the computation of logarithm. They mention a machine learning algorithm, which is called ICSI speaker diarization engine, in which the time spent on computing the log-likelihood function occupies 80% of its total runtime. Furthermore, in [4], Alachiotis and Stamatakis present biological applications where the logarithm function also plays an important role. From [4], we learn that the logarithm function has been used for facilitating evolutionary reconstruction, processing real-time image applications and skin segmentation algorithms. Thus, to improve the speed of logarithms is significant and necessary.

1.2 Thesis Organization

This thesis is organized as follows: In Chapter 1, we review popular approaches used for implementing the logarithm. In Chapter 2, we describe IEEE-754 Floating-Point Standard, ICSILog and Interpolation Algorithms. Tolerance and Performance Assessments are presented in Chapter 3. In the following Chapter 4, 5 and 6 we present the structure of a Pipelined Adder, Pipelined Multiplier and Pipelined Read-only Memory. We conclude in Chapter 7.

1.3 Novelty

In this thesis, we present a pipelined logarithm implementation with fixed accuracy and speed. The overall pipeline structure includes 240 stages (or a 240 clock latency). The longest latency among those stages is equal to the propagation delay of a D flip-flop plus four levels of logic gates. The pipeline is expected to generate at least 2.9G single precision values per second under a 65nm fabrication process (see section 3.3). Moreover, we use a lookup table for speed and a parabolic function to interpolate the values of logarithm. The lookup table contains 648 entries and occupies roughly 7.776KB of memory space (see chapter 2). The accuracy for a single precision floating-point number is at least 21 bits. In other words, the maximum tolerance is 2 bits. The hardware implementation is mainly composed of three components, which are pipelined adder, pipelined multiplier, and pipelined Read-only Memory as well (see chapter 4, 5 and 6).

1.4 Background

One popular approach used by most researchers to implement complicated math functions is the so-call lookup table based method. Here, the lookup table serves as a map table. The values of a math function have been computed in advance and then stored into a lookup table. Thus, function evaluation is equivalent to searching for the lookup table and then retrieving the desired item. Besides, almost no any other additional computations are needed. The time spent on function evaluation is equal to memory access and retrieval time. We could say the lookup table based method makes function evaluation faster and even more precise. However, it does have weaknesses. The size of lookup tables for most math functions are very large. In most cases, they are too big to fit into cache. For example, the lookup table for single precision logarithm contains 8M (2^{23}) items, which in turn occupies 32MB ($2^{23}*$ 4 bytes) of memory space, with each item taking up 4 bytes. This causes data to be frequently exchanged between cache and memory during function evaluation, and makes memory access and retrieval time even longer. This situation will be worse if several lookup tables have been used when different math functions are being evaluated at the same time. Besides waiting for hardware innovations (i.e. a faster and larger cache or memory), how can we further speed up function evaluation? Actually, several attempts have been made by researchers and they are discussed in the following section.

1.5 Methods used for Accelerating Function Evaluation

1.5.1 Shrink Lookup Table Size

Since not all scientific applications require as high precision as those provided by the GNU math library, some researchers construct smaller lookup tables for logarithm with adjustable accuracies (as discussed in [1]).The size of the lookup table represents a trade-off between greater accuracy with a larger table, and faster lookup and fewer resources used with a smaller table. To take advantage of this, users need to choose table size based on their applications. This method is mainly implemented in software since it is easier to configure and load lookup tables. By using this method, it is about 7 times faster running on an identical computer than GNU while 16 bits accuracy has been enforced for single precision values. The improvement mainly comes from the smaller lookup table, which is able to effectively reduce the time spent exchanging data between cache and memory.

1.5.2 Multi-core Technique

A large lookup table can be split into a number of smaller tables, each of which can fit into cache. Those tables will in turn be loaded into different processors of a multi-core system. As a result, each processor will keep a fraction of the original lookup table. During function evaluation, all processors compute in parallel and each processor accesses and retrieves items within the range of its own lookup table. Thus, more values can be generated per second.

The idea behind the multi-core technique is very similar to the idea that a job can be done faster if more people participate. However, several problems are inherent when designing a multi-core system. For example, how to write programs so that codes which can be executed in parallel are distributed to different processors of a multi-core system? As far as I imagine, there could be two approaches. One approach is by using a programming language which has concurrent language constructs. Those constructs are able to make the compiler aware of potential parallisms. The other approach tries to build a more powerful compiler, which can discover parallelism within a segment of

code. I think the first approach may complicate programming since programmers need to design their algorithms from a concurrent perspective. For the second approach, it introduces a new topic – the design of a compiler with a parallelism identification facility. Besides, a network is required to facilitate data distribution and collection. For these reasons, the design of a multi-core system involves a lot of software and hardware designs.

1.5.3 Pipeline Technique

By effectively using the pipeline technique, one is able to increase the throughput of a system. Before the pipeline technique was introduced in the field of computer design, it was already widely used at the assembly lines of factories to increase work efficiency. It tries to split a task into a series of smaller sub-tasks. Each sub-task is made as a separate stage. A pipeline is composed of a number of stages. As soon as a sub-task in one stage has been done, it will be forwarded to the next stage for further processing. A task is thought to be completed as long as it goes through every stage of a pipeline. One difficulty in designing a pipeline lies in how to effectively split a task into a series of smaller sub-tasks, so that the workload of each sub-task is balanced. In addition, the pipeline approach does not reduce the latency of executing a single task. Sometimes, it has even longer latency than non-pipelined designs. Thus, the pipeline technique is more appropriate to process a batch of data.

1.5.4 FPGA Technique

An FPGA (Field Programmable Gate Array) is a chip containing reconfigurable logic. Unlike conventional logic, which requires a very time-consuming and expensive design process, which is only affordable for large projects at big companies, FPGAs are custom hardware which can be designed by individual engineers and scientists, frees hardware design from the few hardware vendors and brings many of the advantages of custom logic to. A designer can build the desired hardware by configuring IP blocks in FPGAs. New generation FPGA chips contain embedded memory blocks and DSP slices. For example, the Altera Stratix IV EP4SGX230 FPGA chip (see [2]), it includes but is not limited to 17,133Kbit of embedded memory (operating at 600 MHz) and 1,288 18×18 embedded multipliers (operating at 600 MHz) as well.

In [3], Dinechin, Joldes and Pasca present a pipelined floating-point logarithm unit by using Xilinx Virtex Series FPGAs. The maximum through-

put under single precision is 244MHz. Furthermore, in [5], Alachiotis and Stamatakis present a pipelined floating-point exponential unit for Xilinx Virtex 2, Virtex 4 and Virtex 5 FPGAs. It occupies 5% of hardware resources on the Virtex 5 SX95T. The maximum throughput under single and double precisions is 168MHz and 252MHz respectively.

Although FPGA provides a fast and efficient way to design hardware, the performance of the configured hardware is largely constrained by the frequency of IP blocks on the target FPGA. Suppose we intend to implement hardware with a pipelined structure, then the overall throughput can hardly exceed the highest frequency of the target FPGA. Thus, in order to implement a pipelined logarithm with 1G operations per second throughput, we need to try other design alternatives.

Chapter 2

Algorithm and Implementation

2.1 Design Principle

My goal is to implement a single precision, pipelined logarithm computation unit (LCU) with 1G operations per second throughput. The LCU is expected to get high precision while using a small lookup table. A small lookup table implies less decoding time and less hardware resources, so that it can be incorporated into the ALU of a CPU.

In order to reach this goal, I chose to use the pipeline technique. Compared with aforementioned approaches in Chapter 1, the pipeline technique has a number of advantages. First, the principle for the pipeline technique is straightforward. A pipelined system is able to get high throughput as long as it can be split into a series of sub-tasks with equal workload. Here, the expected latency for each stage is less than or equal to the propagation delay of a D flip-flop plus a 1 bit full adder. Second, the behaviour of the pipeline can be simulated both by hardware and software. Our hardware implementation was built from scratch, by using logic gates and D flip-flops only. Before implementing the hardware, we used C language to simulate the behaviors of the hardware and give the expected tolerance.

2.2 IEEE-754 Floating Point Standard

Figure 2.1 illustrates how floating point numbers are represented in single and double precision formats in the IEEE-754 standard [6]. Table 2.1 shows the represented values in single precision format as defined by the IEEE-754 standard.

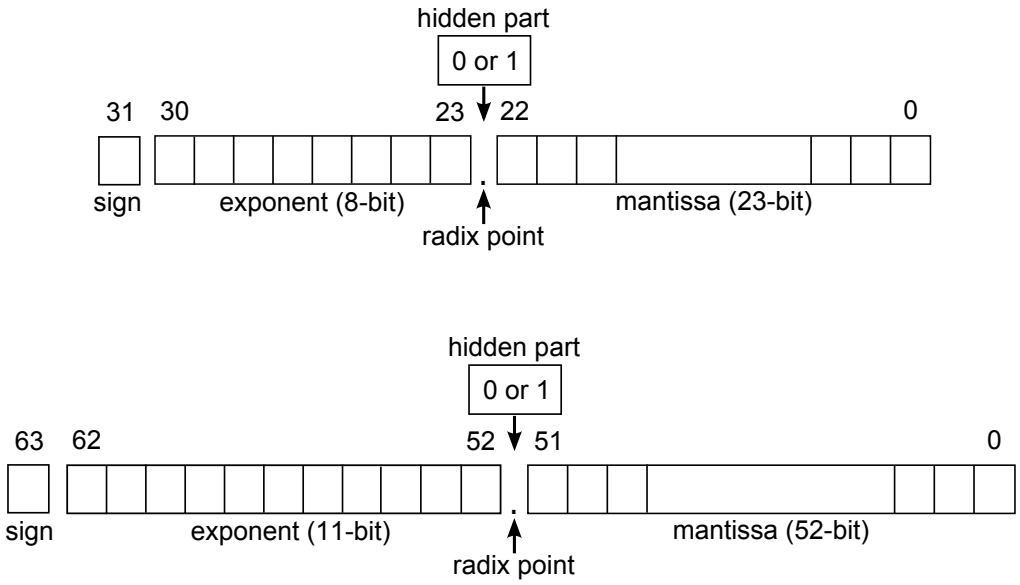

Figure 2.1: Single and double precision floating-point number in IEEE 754.

Table 2.1: Represented Values for Single Precision Format in IEEE-754.

Sign (binary)	Exponent (hexadecimal)	Mantissa (hexadecimal)	Value (decimal)
0 or 1	e=0x00	m=0x0000	±0.0
0 or 1	e=0x00	m≠0x0000	$\pm 2^{-127}$*0.m
0 or 1	0x00<e<0xff	random	$\pm 2^{-127}$*1.m
0 or 1	e=0xff	m=0x0000	±inf
0 or 1	e=0xff	m≠0x0000	±NaN

A floating point number as defined by IEEE-754 contains three parts, which are the sign, exponent and mantissa. The single precision format requires 4 bytes (32 bits) for one floating number. The most significant bit represents the sign. The following 8 bits represent the exponent and last 23 bits represent the mantissa. The double precision format requires 8 bytes (64

bits) for one floating number. The most significant bit represents the sign as well. The following 11 bits represent the exponent and the last 52 bits represent the mantissa. Any floating-point number can be represented as (the following two formulas are quoted from [4]):

$$number = sign * 2^{exponent} * mantissa \tag{2.1}$$

Here, sign represents positive or negative, so:

$$number = (+/-) * 2^{exponent} * mantissa \tag{2.2}$$

By observing Figure 2.1, we learn that double precision arithmetic is more complicated and requires more resources. Thus, we chose to implement the single precision logarithm first.

2.3 ICSILog Algorithm

In this section, we introduce ICSILog algorithm [1]. By applying logarithmic product and power rules, the value of log_2 for a positive number can be represented as follows (the following formulas and formula reasoning are quoted from [1]):

$$\begin{aligned} \log_2(number) &= \log_2(2^{exponent} * mantissa) && (2.3)\\ &= \log_2(2^{exponent}) + \log_2(mantissa) && (2.4)\\ &= exponent * \log_2(2) + \log_2(mantissa) && (2.5)\\ &= exponent + \log_2(mantissa) && (2.6) \end{aligned}$$

Implementing log_2 is simpler than logarithm with other bases since $\log_2(2^{exponent}) = exponent$ holds in this case. By applying the logarithmic change base rule, logarithm with bases other than two can be represented as follows:

$$\begin{aligned} \log_n(number) &= \log_2(number)/\log_2(n) && (2.7)\\ &= [exponent + \log_2(mantissa)] * [1/\log_2(n)] && (2.8) \end{aligned}$$

Let n be a positive number other than two. The above formula demonstrates that the value of a logarithm with a base other than two can be obtained by multiplying a factor into the value of log_2. Here, the factor refers to $[1/log_2(n)]$. Thus, as long as the value of log_2 is pre-computed, the value of a logarithm with other bases can be derived from it.

By observing the above formula, one might feel that implementing log_2 is not difficult. However, it is not as simple as it seems. Implementing log_2 with high accuracy and small lookup table size involves a lot of work.

In order to get the value for log_2, we need to obtain the value of $log_2(mantissa)$. We get this value from a lookup table. If we treat all of the 23 bits of mantissa to index the lookup table, the table will have 8M items and need 32MB of memory space, with each item taking up 4 bytes. Table 2.2 illustrates the contents of a 32MB lookup table.

Table 2.2: 32MB lookup table with 23-bit index.

Location	Index	Content
0	$\underbrace{00000000000000000000}_{23}$	$\log_2(1.\underbrace{00000000000000000000}_{23})$
1	00000000000000000001	$\log_2(1.00000000000000000001)$
...	...	...
$2^{23}-1$	11111111111111111111	$\log_2(1.11111111111111111111)$

We place the single precision value of $\log_2(1.00000000000000000000)$ in the first location. Similarly, the values of $\log_2(1.00000000000000000001)$ and $\log_2(1.11111111111111111111)$ are placed in the second and last locations of the table.

To shrink the table size, we use only part of the 23 bit mantissa, say, high 16 bits, as an index. In this case, the new table will have 64K items and need 256KB memory space. Table 2.3 illustrates the contents of a 256KB lookup table.

Table 2.3: 256KB lookup table with 16-bit index.

Location	Index	Content
0	$\underbrace{0000000000000000}_{16}$	$\log_2(1.\underbrace{0000000000000000}_{16}0000000)$
1	0000000000000001	$\log_2(1.00000000000000010000000)$
...	...	...
$2^{16}-1$	1111111111111111	$\log_2(1.11111111111111110000000)$

Table 2.2 has been reduced by 128 times compared to table 2.3. However, this new table leads to a problem. Since it does not store all of the

data, some values of log_2 with neighboring mantissa will map onto the same location. The term "adjustable lookup table" refers to a table whose size can be changed dynamically based on desired accuracy (as presented in [1]).

2.4 Interpolation Algorithm

In previous sections, we realized that it is possible to accelerate the computation speed of our design by shrinking the lookup table size. However, at the same time, it leads to accuracy loss. Does there exist a method that could not only shrink lookup table size but also retain the accuracy to a large extent? One can use interpolation to approximate the intermediate values of a function when the input lies between two adjacent samples of a lookup table. Obviously, interpolation is much better than just returning the sample value as the result. There are a number of interpolation methods and each of them has its own properties and result in different effects.

2.4.1 Methods used for Interpolation

In the following sections, we introduce three interpolation methods, which are beeline interpolation, parabolic interpolation and B-spline interpolation. Parabolic interpolation is a special case of B-spline interpolation. Each of the three has both advantages and shortcomings. Thus, the choice largely relies on the specific application. Here, my goal is to find a method that is able to generate precise approximate values since a better interpolation method means a smaller lookup table. Furthermore, the algorithm of this method should not be too complex. A simpler algorithm implies simpler circuits and less hardware resources.

In fact, the algorithms for beeline and parabolic interpolation are simpler than B-spline interpolation. However, parabolic and B-spline interpolation give more precise approximate values than beeline interpolation does. By comparing the features they have, I concluded that parabolic interpolation is more appropriate for our design because the approximate values given by parabolic interpolation are much more precise in comparison to beeline interpolation while the increase of complexity of the algorithm is manageable.

2.4.2 Beeline Interpolation

A curve which represents log_2 is drawn in the graph shown in Figure 2.2.

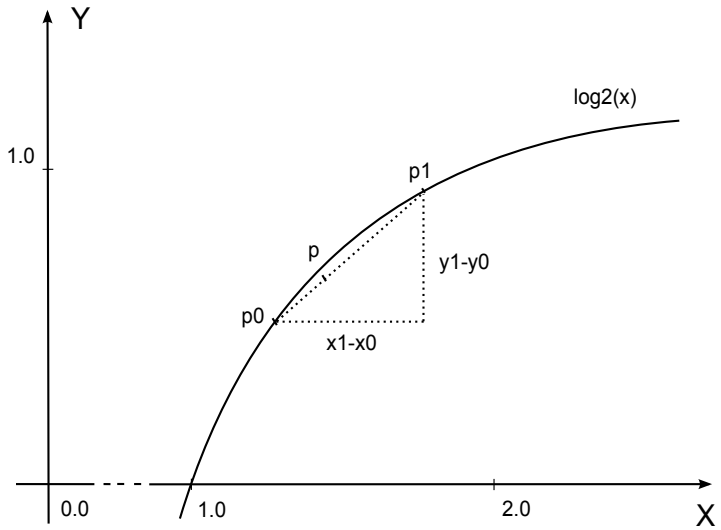

Figure 2.2: Beeline interpolation.

There exist two sample points on this curve, which are $p_0(x_0, y_0)$ and $p_1(x_1, y_1)$. Suppose we would like to evaluate the Y-coordinate value for a point p whose input lies between the two sample points. For beeline interpolation, we draw a line that goes through the two points and then calculate the approximate Y-coordinate value of p on this newly drawn line. The formula for the line is:

$$y = y_0 + k(x - x_0) \tag{2.9}$$

$$k = [(y_1 - y_0)/(x_1 - x_0)] \tag{2.10}$$

Here, k is a constant. It can be pre-computed and then stored in a lookup table in advance. From the above formula, we know beeline interpolation needs one multiplication plus one addition at run time.

2.4.3 Parabolic Interpolation

In Figure 2.3, there are two coordinate systems, which are XY and X'Y'.

A curve which represents log_2 is drawn in X'Y'. There exist three sample points on this curve, which are p_0, p_1 and p_2. Suppose we would like to evaluate

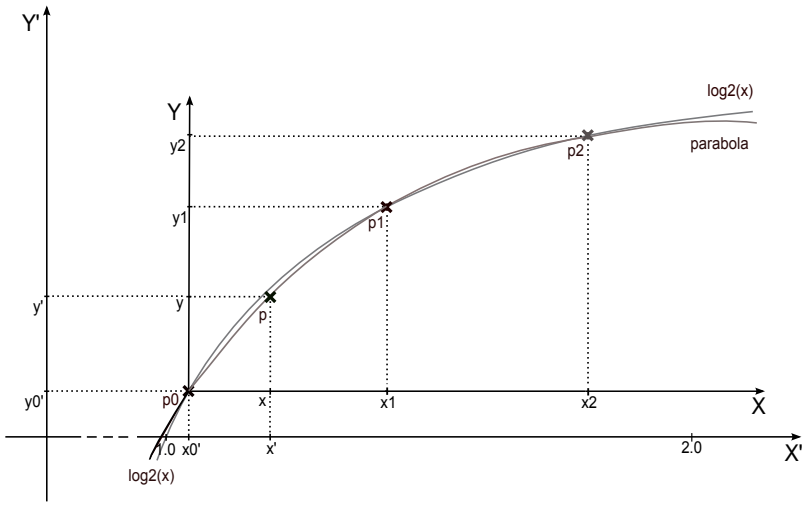

Figure 2.3: Parabolic interpolation.

the Y-coordinate value for a point p on this curve when its input lies between p_0 and p_1. For parabolic interpolation, it is expected to first draw a parabola that goes through the three sample points and then calculate the approximate Y-coordinate value for p on this newly drawn parabola. In order to make things simpler, we build a new coordinate system XY and set its origin at p_0. Then, the XY-coordinate for the three sample points and the point p are p_0 (0, 0), $p_1(x_1, y_1)$, $p_2(x_2, y_2)$ and p(x, y), where $0 < x < x1 < x2$, $0 < y < y1 < y2$. The formula for the approximated parabola in XY is as follows:

$$y = ax^2 + bx \tag{2.11}$$
$$a = (y_2 - 2y_1)/(2x_1^2) \tag{2.12}$$
$$b = (4x_1y_1 - x_1y_2)/(2x_1^2) \tag{2.13}$$

Suppose the X'Y'-coordinate values for p_0 and p are $p_0(x_0', y_0')$ and p (x', y'). Thus, we have:

$$y' = y'_0 + y = y'_0 + ax^2 + bx \quad (2.14)$$
$$x' = x'_0 + x \quad (2.15)$$
$$a = (y_2 - 2y_1)/(2x_1^2) \quad (2.16)$$
$$b = (4x_1y_1 - x_1y_2)/(2x_1^2) \quad (2.17)$$

In 2.16, 2.17 a and b are both constant. They can be pre-computed and stored in a lookup table in advance. From the above formula, we know parabolic interpolation normally needs two multiplications plus two additions at run time since the bx term and the ax^2 term can be calculated in parallel.

2.4.4 Discussion

In the previous two sections, we showed that the workload for parabolic interpolation is almost double the workload for beeline interpolation. If beeline interpolation is to reach the same exact precision as parabolic interpolation does, its lookup table has to be 180 times larger. Parabolic interpolation is favored for our design since it can shrink the size of a lookup table by a lot while at the same time just slightly increasing the workload. The lookup table used for our design contains 648 items and takes up roughly 7.776KB of memory space. In addition, it achieves 21 bits precision as well. In other words, it has at most 2 bits tolerance.

2.4.5 Overall Flow Chart

Figure 2.4 illustrates the flow chart of the whole design.

In order to fit our design into a pipelined structure, all of the involved components have to be pipeline-liked. From the flow chart shown in Figure 2.4, we know our logarithm implementation is mainly composed of three parts: the adder, multiplier and Read-only Memory. Thus, we need to build a pipelined adder, pipelined multiplier and pipelined Read-only Memory as well.

To the best of our knowledge, there is little literature about designing a full pipeline, particularly for read-only memory. However, without finding solutions for them, our implementation may not reach a very high throughput. Thus, all need to be designed from scratch. Placing a large chunk of a single component into one pipeline stage will decrease the throughput dramatically since the throughput of a pipeline is determined by the stage with the longest latency. As a result, it is crucial to have the workload of every stage balanced. In our design, the stage which has the longest latency is equal to the propa-

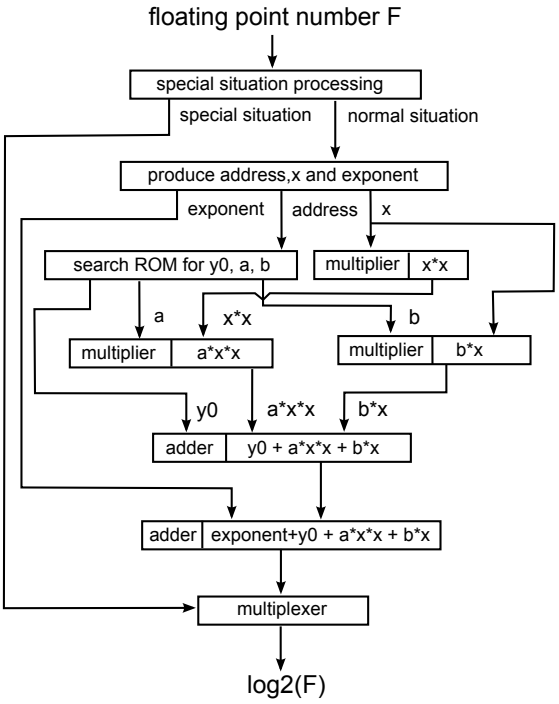

Figure 2.4: Flow chart of our log_2(x) implementation.

gation delay of a D flip-flop plus four levels of logic gates.

Chapter 3

Tolerance and Performance Assessment

3.1 Design Method

Nowadays, CAD (Computer Aided Design) tools greatly facilitate hardware design. It has enhanced design quality, cut down on design time and reduced design cost. Take the Altera Quartus II design software for example, it generally provides two ways to design hardware. One is by using a textual language, say, VHDL. The other is by using schematics. VHDL is a very productive hardware description language. Several lines of VHDL statements may produce a circuit with hundreds or thousands of logic gates. However, it is not easy for a textual language to control the generated circuits in detail. A schematic, on the other hand, replaces the traditional method of using pencil and eraser to draw the block diagram for hardware. It is less productive than VHDL since one needs to specify every tiny detail for a circuit (i.e. logic gates or wire connections). In order to design a high throughput pipeline system, we need to control the way that how each component is pipelined, so schematics is more appropriate for us.

Before implementing the hardware, we used C language to validate the approximations used by doing exhaustive testing of the algorithm. The C code will then exactly match the simulation of the circuit, making it easier to test the circuit at a smaller set of values, and still know the tolerance. Besides, software simulation allows us to easily try new algorithms and generate the expected tolerances rapidly. To obtain higher precision, trying and testing different new algorithms is better than just doing formula reasoning. Sometimes, even different calculating sequences affect the precision of the whole design. As a

result, theoratical analysis and software simulation are both very necessary.

3.1.1 Software Simulation

The software simulation program was designed by using the GNU C Compiler which is embedded into Cygwin. The program simulated a pipelined logarithm realization and we performed exhaustive testing on it.($2^{31} = 2G$ input values in total).

3.1.2 Hardware Verification

The hardware design is implemented by using the Altera Quartus II 7.2 Design Software (Web Edition). The whole design is compiled and simulated[1]. If our hadrware implementation does not have any bugs, values generated by the hardware implementation should be totally consistent with the software simulation program. However, we just performed partial testing on the hardware implementation near 1.0 due to the long simulation time.

3.2 Tolerance Assessment

Different math libraries may use different methods or lookup tables to compute the logarithm, so their generated values may differ from each other. For our design, the lookup table is generated by the GNU math library. We use parabolic interpolation to produce the intermediate values between two adjacent samples of the lookup table, so accuracy loss is not avoidable. Thus, minimizing the tolerance to the largest extent becomes essential. The value of log_2 varies from negative infinite to positive infinite. As a result, the range of the corresponding tolerance is not fixed. It changes as the exponent varies. In the rest of this section, we intend to define the tolerance for our implementation and then give several examples to explain it.

Suppose we have two single precision floating-point numbers, N1 and N2. N1 is represented as 3.1, where x represents one bit:

$$N_1 = 2^{e1} * 1.\underbrace{xxxxxxxxxxxxxxxxxxxxxxx}_{23} \tag{3.1}$$

If N_2 can be represented as 3.2, where y also represents one bit:

[1]Here, the simulation only refers to functional simulation. Functional simulation does not consider timing issues of a circuit, it verifies the behavioral correctness of a circuit.

$$N_2 = N_1 \pm (2^{e1} * 0.\underbrace{00000000000000000000}_{21}\underbrace{yy}_{2}) \tag{3.2}$$

We say N2 has at least 21 bits precision with respect to N1. In other words, N2 has at most 2 bits tolerance. For example, if the value of N1 is:

$$N_1 = 2^0 * 1.\underbrace{10000000000000000000000}_{23} \tag{3.3}$$

and the value of N_2 is:

$$N_2 = 2^0 * 1.\underbrace{01111111111111111111111}_{23} \tag{3.4}$$

Then, we say N_2 has at least 21 bits precision with respect to N_1 since:

$$N_2 = N_1 - 2^0 * 0.\underbrace{00000000000000000000001}_{23} \tag{3.5}$$

The tolerance of log_2 is mainly observed in two intervals, which are [0.5,1.0) and [1.0,2.0). It becomes worse when the input value comes close to the value 1.0. Thus, we pay more attention on those two intervals when we perform testing. Our software simulation program performed exhaustive testing($2^{31} = 2G$ input values in total) and then compared the generated values against the GNU math library. Furthermore, the values generated by our hardware implementation should be totally consistent with our software simulation program if it does not contain any bugs. However, we only perform partial testing on it near 1.0 due to the long hardware simulation time. In the following, two testing examples are given at table 3.1.

Table 3.1: Testing examples

Example	Sign	Exponent	Mantissa	Decimal Value
x1	0	01111110	11111111111111111110	$9.999999*10^{-1}$
$GNUlog_2(x1)$	1	01101000	01110001010101000111100	$-1.719827*10^{-7}$
$MYlog_2(x1)$	1	01101000	01110001010101000111111	$-1.719827 * 10^{-7}$
x2	0	01111111	00000000000000000000001	$1.000000*10^{0}$
$GNUlog_2(x2)$	0	01101000	01110001010101000111010	$1.719826*10^{-7}$
$MYlog_2(x2)$	0	01101000	01110001010101000110111	$1.719826*10^{-7}$

Here, x_1 represents a 32 bits single precision floating-point input. *GNU* log_2 represents the corresponding single precision value $log_2(x_1)$ as computed by the GNU math library. $MYlog_2$ represents the single precision value generated by our software simulation program. We guarantee that all the values calculated by our logarithm implementation have at least 21 bits precision. Compared with the precision of log_2 given by the GNU math library, our precision is fairly good given that our lookup table contains 648 items. Tables 3.2 and 3.3 illustrates the overall tolerance analysis reports.

Table 3.2: Overall tolerance analysis report 1

Range of input x		Accuracy
binary	decimal	
$2^{-2}*1.00000000000000000000000$ to $2^{-2}*1.11111111111111111111111$	0.25 to 0.5	22 bits
$2^{-1}*1.00000000000000000000000$ to $2^{-1}*1.11111111111111111111111$	0.5 to 1.0	21 bits
$2^{0}*1.00000000000000000000000$ to $2^{0}*1.11111111111111111111111$	1.0 to 2.0	21 bits
$2^{1}*1.00000000000000000000000$ to $2^{1}*1.11111111111111111111111$	2.0 to 4.0	22 bits

Table 3.3: Overall tolerance analysis report 2

Input(in decimal)	Accuracy	Maximum tolerance	Average tolerance
0.25 to 0.5	22 bits	1 bit	less than 0.59-bit
0.5 to 1.0	21 bits	2 bits	less than 0.59-bit
1.0 to 2.0	21 bits	2 bits	less than 0.59-bit
2.0 to 4.0	22 bits	1 bit	less than 0.59-bit
other areas	22 bits	1 bit	less than 0.59-bit

Tables 3.2 and 3.3 were computed by our software simulation program. The highlighted rows denote precision results for the worst cases. When the input value x of log_2 lies in the intervals [0.5,1.0) and [1.0,2.0), the corresponding output of log_2(x) have an accuracy of 21 bits. In other words, the accuracy loss for those values is at most 2 bits. As we know, software sometimes may not entirely simulate the real performance of its corresponding hardware circuits. Thus, the final report can only be obtained after performing comprehensive hardware tests.

3.3 Performance Assessment

Our pipelined logarithm implementation is independent of any specific FPGA platform. It does not use any IP blocks, and is designed by only using logic gates (AND gate, OR gate, Inverter) and D flip-flops. It is also possible to

run our design on FPGAs, but the expected throughput is not expected to be high since our design is not targeted at a specific FPGA. If we intend to realize our design by configuring the IP blocks of a target FPGA, then the expected throughput will be determined by the organization and performance of those chosen IP blocks. Furthermore, the expected throughput can hardly exceed the maximum frequency among the chosen IP blocks. For example, the maximum operating frequency for the embedded multiplier and memory on an Altera Stratix IV EP4SGX230[2]is 600MHz. In order to implement a pipelined logarithm with 1G operations per second, we need to try other design alternatives.

There are two reasons a hardware design obtains very high speed running on an FPGA chip. First, it may use advanced algorithms. Second, it may take full advantage of the hardware resources on the target FPGA. In other words, the designer may configure the IP blocks in a good manner. It is normal to see that a pipelined design with better algorithms obtains lower throughput due to badly configured IP blocks. Also, one pipelined design may have different throughput when configured by different people. Thus, we prefer to treat our pipelined design as a custom circuit and assess its performance by adopting the measurements used for custom circuits.

The throughput of a pipelined system is determined by the stage which has the longest latency. In our design, the longest latency is equal to the propagation delay of a D flip-flop, an AND2[3] gate, an inverter, an AND3 gate and an OR4[4] gate.[5].

Suppose the propagation delay for the aforementioned devices are pd_{dff}, pd_{and2}, $pd_{inverter}$,pd_{and3} and pd_{or4} respectively. Thus, the total latency t is:

$$t = pd_{dff} + pd_{and2} + pd_{inverter} + pd_{and3} + pd_{or4} \tag{3.6}$$

The propagation delay for a device is determined by the specific fabrication process it has been made under. The propagation delay for the aforementioned devices under 65nm [17] and 0.8μm fabrication process are given

[2]A new generation FPGA chip used for the Altera DE4.

[3]AND2 refers to a logic gate with two inputs and performs logical-and operation between the inputs. Similarly, AND3 refers to a logic gate with three inputs and performs logical-and operation between the inputs.

[4]OR4 refers to a logic gate with four inputs and performs logical-or operation between the inputs.

[5]This stage may not be the one which has the longest latency in our pipeline system, or some circuits in our design may violate fanout rule. Since our project is very large (2.6GB), we are likely to make mistakes. Once a mistake is discovered, it can always be fixed by splitting the stage into several smaller ones.

in Table 3.4[6] and Table 3.5.

Table 3.4: Propagation delay under a 65nm fabrication process

Inverter	AND2	OR2	D type flip-flop
10 ps	25 to 40 ps	25 to 45 ps	60 ps

Table 3.5: Propagation delay under a 0.8μm fabrication process

Inverter	AND2	AND3	AND4	OR2	OR3	D type flip-flop
0.1 ns	0.25 ns	0.28 ns	0.29 ns	0.34 ns	0.37 ns	1.48 ns

First, let us calculate the potential throughput for our design under a 65nm fabrication process by using equation 3.6 and table 3.4. Suppose the propagation delay for AND3 and OR4 under a 65nm fabrication process is from 50 to 90ps, which almost doubles the delay for AND2 and OR2 listed in 3.4. Thus, the maximum and minimum latencies are:

$$t_{min} = 60ps + 50ps + 10ps + 50ps + 50ps = 220ps \tag{3.7}$$

$$t_{max} = 60ps + 90ps + 10ps + 90ps + 90ps = 340ps \tag{3.8}$$

We can also calculate the maximum and minimum throughput by using values obtained from equations 3.7 and 3.8.

$$tp_{min} = \frac{1}{t_{max}} = \frac{1}{340 \times 10^{-12}s} \approx 2.9G/per\ second \tag{3.9}$$

$$tp_{max} = \frac{1}{t_{min}} = \frac{1}{220 \times 10^{-12}s} \approx 4.5G/per\ second \tag{3.10}$$

Second, let us calculate the potential throughput for our design under a 0.8μm fabrication process by using equation 3.6 and table 3.5. Suppose the propagation delay for OR4 under a 0.8μm fabrication process is 0.4ns. Thus, we estimate the latency to be:

$$t = 0.25ns + 0.1ns + 0.28ns + 0.4ns + 1.48ns = 2.51ns \tag{3.11}$$

[6]Data listed in tables 3.4 and 3.5 are not very formal, they were obtained by consulting domain experts.

We can calculate the throughput by using the value obtained from equations 3.11.

$$tp = \frac{1}{t} = \frac{1}{2.51 \times 10^{-9} s} \approx 398M/per\ second \tag{3.12}$$

In summary, the potential throughput for our pipelined logarithm implementation under a 65nm fabrication process is estimated to be from 2.9G to 4.5G[7] single precision values per second. However, when we consider a 0.8μm fabrication process, our design is estimated to generate 398M[8] single precision values per second.

3.4 Performance Comparison

To the best of our knowledge, in [4], Alachiotis and Stamatakis present the fastest pipelined single precision logarithm approximation unit (SP-LAU) implemented on an FPGA with adjustable accuracy. Their implementation is 11 and 1.6 times faster than the GNU and Intel Math Kernel Library (MKL) implementations and up to 1.44 times faster than the FloPoCo reconfigurable logarithm unit (as presented in [3]).

In [4], we learned that, when a lookup table is loaded with 4096 entries (13.5KB), SP-LAU is able to generate 353.5M single precision values per second. The latency for the pipeline is 22-26 clock cycles (62.2ns-73.5ns). In addition, we also learned that a medium sized FPGA chip[9] normally contains 1.35MB of embedded memory. Thus, SP-LAU-4096[10] takes nearly 1% of the embedded memory provided by the target FPGA (Xillinx Virtex 5 SX95T). Furthermore, average tolerance of SP-LAU-4096 is 9 bits.

In fact, the performance between SP-LAU-4096 and our hardware implementation (SP-LCU[11]) is not comparable. First, SP-LAU-4096 is designed by configuring IP blocks on a target FPGA. Thus, the performance of SP-LAU-4096 will be constrained by the chosen IP blocks (i.e. embedded multiplier frequency, embedded memory size). Second, a design that is targeted at a specific FPGA is less flexible than a design for custom hardware. For example, if there exists a stage in our pipelined implementation that has a

[7]This value is estimated based on table 3.4
[8]This value is estimated based on table 3.5
[9]Xillinx Virtex 5 SX95T
[10]SP-LAU with 4096 entries.
[11]Pipelined Single Precision Logarithm Computation Unit

longer latency than other stages, then it is very easy to split it into a number of stages that has shorter latency. However, for FPGA implementation, it is not easy to make such changes. Sometimes, this kind of change is crucial for pipeline throughput.

As stated above,, we will not give comparison between SP-LAU-4096 and SP-LCU. Nevertheless, the corresponding data of the two implementations are listed below:

Table 3.6: Performance comparison

Unit	Throughput	Maximum Tolerance	Lookup table Size	Pipeline Latency
SP-LAU [4]	0.3535GHz [4]	9 bits [4]	13.5KB [4]	62.2ns-73.5ns [4]
SP-LCU	2.9GHz (is estimated by 3.4)	2 bits	7.776KB	52.8ns-81.6ns (is estimated by 3.4)

Chapter 4

Pipelined Adder

In chapter 4, we introduce a pipelined 4 bit adder (see section 4.5). This demonstrates how to apply the pipeline technique to combinational logic circuit to enhance circuit performance. The reason to choose a 4-bit pipelined adder is because of the simple structure and straightforward principle.

4.1 Informal Description of a Full Adder

Figure 4.1 shows the block diagram for a full adder [7] [8](Block diagram directly comes from Quartus II design software, but altered to our purpose). It takes three inputs and generates two outputs. For the three inputs, a and b are operands, ci is the carry. For the two outputs, s is the sum, co is the carry for the next adjacent full adder.

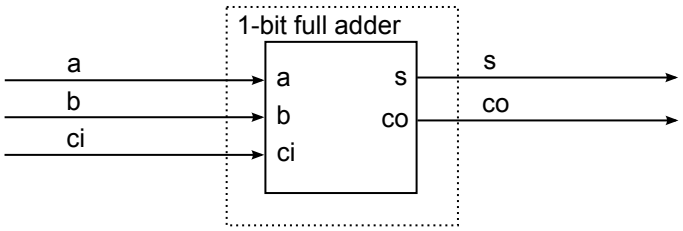

Figure 4.1: Block diagram of a 1-bit full adder

4.2 Switching Equations of a Full Adder

Table 4.1 shows the truth table of a full adder [7] [8]. Since a full adder has three inputs, there will be eight entries in the table.

Table 4.1: Truth table of a full adder

Index	ci	a	b	s	co
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

By Table 4.1, the switching equations [7] [8] for s and co are:

$$s = a'bci' + ab'ci' + a'b'ci + abci \tag{4.1}$$

$$co = aci + bci + ab. \tag{4.2}$$

4.3 Schematics of a Full Adder

Switching equations can be implemented as circuits by replacing logic operations by gates. Figure 4.2 illustrates the schematics for a full adder (the block diagram comes directly from Quartus II design software, but is altered for our purpose).

Schematics shown in Figure 4.2 contains five kinds of logic gate: inverter, AND2, AND3, OR3, OR4. Here, the digit which follows the type of gate denotes the number of inputs each gate has. Normally, the more inputs a gate has the longer propagation delay is needed to generate stable output. Furthermore, the schematic invovles three levels of logic gates. Each input signal needs to propagate through at most three logic gates to get to the output port. Latency of the schematic depends on the most time consuming path in Figure 4.2. Here, it is the path that goes through the inverter, AND3 and OR4.

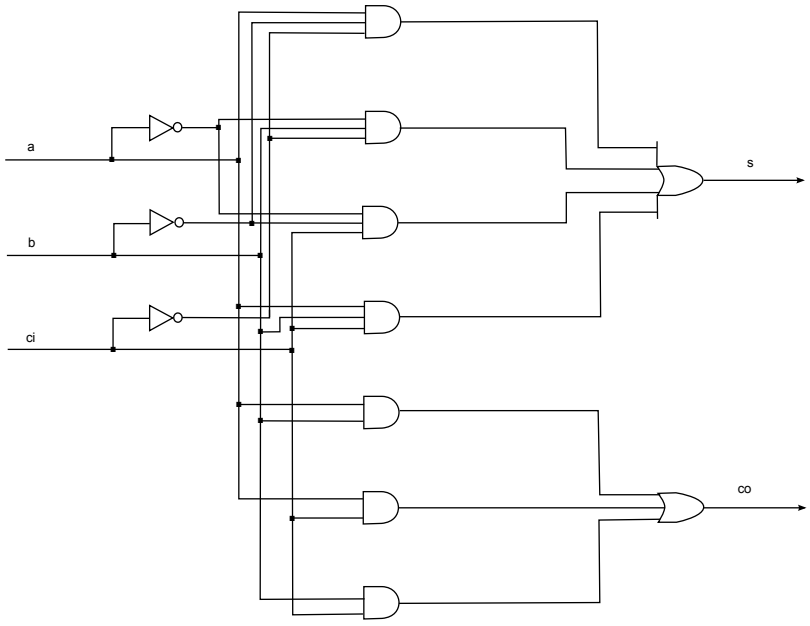

Figure 4.2: Schematics of a full adder.

4.4 A 4-bit Ripple Carry Adder

We can build a 4-bit ripple carry adder [7] [8] by connecting four full adders. Figure 4.3 illustrates the schematics for a 4-bit ripple adder (the block diagramcomes directly from Quartus II design software, but is altered for our purpose).

Suppose the two 4-bit operands of an add operation are $a[3..0]$ and $b[3..0]$. Here, $a[3]$ represents the most significant bit (MSB) and $a[0]$ represents the least significant bit (LSB). For an add operation in Figure 4.3, FA_1 is used for adding $a[0]$ and $b[0]$. The third input ci is set to logical-0 since there is no carry at this time. FA_1 generates $s[0]$ and co. co will be passed to the next adjacent adder, FA_2. Based on the same rationale, FA_4 will not give the right result until the carry generated from FA_3 is ready. Thus, carries have been propagated in a sequential fashion from FA_1 to FA_4.

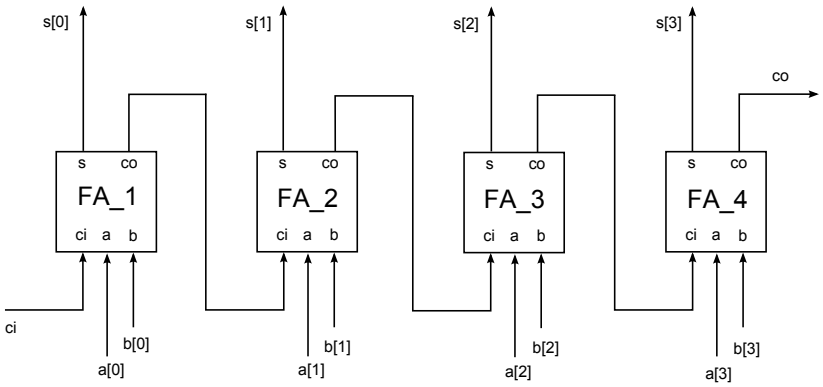

Figure 4.3: Schematics of a 4-bit ripple carry adder.

4.5 A 4-bit Pipelined Ripple Carry Adder

By observing the 4-bit ripple carry adder shown in Figure 4.3, we see that each full adder is in charge of partial evaluation. The operation of a full adder with a higher index relies on the carry generated from an adjacent lower indexed adder. That is to say, FA_4 will stay idle until all of the three previous adders (FA_1,FA_2,FA_3) produce stable results. Obviously, this ripple carry design does not take full advantage of all the hardware resources since many adders will stay idle most of the time. In order to accelerate the circuit speed, we intend to use the pipeline technique to make all possible hardware components work in parallel. Figure 4.4 illustrates the block diagram for a 4-bit pipelined ripple carry adder[1](the block diagram comes directly from Figure 1 in [13],

[1]Our 4-bit pipelined ripple carry adder was designed without referencing the literatures. However, since we did not conduct a thorough bibliographical search, such circuits might have been designed and applied by others already.

but is altered for our purpose).

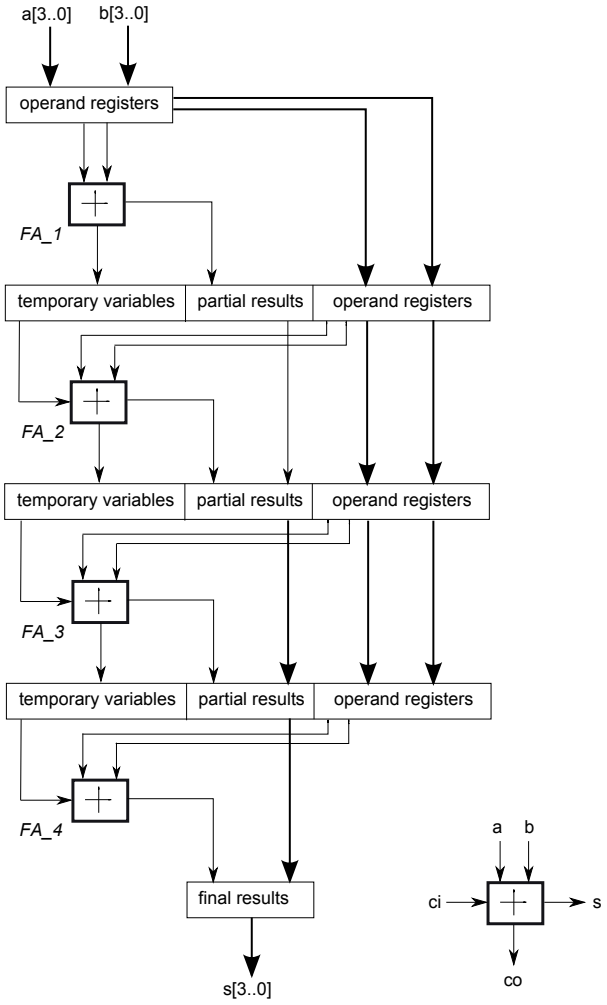

Figure 4.4: A 4-bit pipelined ripple carry adder.

The principle underlaying Figure 4.4 is presented below. In the first clock cycle, we load FA_1 with a pair of operands, say, $a_0[3..0]$ and $b_0[3..0]$. They will be forwarded to FA_2 as soon as FA_1 finishes partial evaluation. Thus, in the second clock cycle, FA_1 will be loaded with a second pair of operands, and FA_2 will be in the process of partially evaluating $a_0[3..0]$ and $b_0[3..0]$. We can see how each full adder is a separate stage in the pipeline structure shown in Figure 4.4. Furthermore, a number of registers are required

to store the temporary variables and partial results. When some of them are useful for later evaluation, they will be maintained. Otherwise, if they become redundant, they will be discarded before entering into the next stage. Due to the introduction of these registers, the schematic shown in Figure 4.4 becomes a sequential circuit rather than a combinational circuit. Sequential circuits are more complicated than combinational circuits, since they introduce timing issues. Registers are storage elements. They store data on either the rising or falling edge of a clock cycle. Here, we make all registers work under one unified clock. In addtion, the clock cycle period needs to be carefully calculated so that each stage is able to finish their work on time. Another key issue that needs to be addressed is how to make the workload of each stage in the pipeline as balanced as possible. If the workload of a specific stage is heavier than the others, the clock cycle period will be set based on it. Obviously, this can increase the latency of a pipeline due to most stages finishing their work earlier. Usually, a stage with a heavier workload can be split into a number of stages with less work. This problem can be ignored here since each stage in our design does identical work.

The pipeline technique increases the throughput of a system rather than decreaseing the time for doing a single item of work. Sometimes, applying the pipeline technique to a system will increase the latency of a single item of work. In our case, we do not speed up the time for completing a single 4-bit add operation. We do, however, make all full adders work in parallel. Thus, more add operations will finish over the same period of time. Since there are four stages in our case, the pipelined ripple carry adder is expected to be four times faster than the non-pipelined version, so long as all of the stages are fully loaded.

The pipeline structure presented in Figure 4.4 is much simpler than those used for a general purpose processor because it does not need to handle cases like forwarding, stalls, flushing and prediction.

Chapter 5

Pipelined Multiplier

In chapter 5, we introduce a 4-bit pipelined array multiplier. It applies both the pipeline technique and the carry save algorithm to improve the circuit's speed. Furthermore, three kinds of non-pipelined array multipliers are presented and compared. They are the ripple carry array multiplier (see section 5.2), carry look-ahead array multiplier (see section 5.3) and carry save array multiplier (see section 5.4).

5.1 A 4×4 Unsigned Multiplication

Before starting to design a multiplier, let us take a look at Table 5.1 which demonstrates a 4×4 multiplication of two unsigned operands.

Table 5.1: An example of a 4×4 unsigned multiplication

				1	1	0	1
×				0	1	1	1
+				1	1	0	1
+			1	1	0	1	
+		1	1	0	1		
+	0	0	0	0			
0	1	0	1	1	0	1	1

The values of the first and second operand are 1101 and 0111 in binary. Here, we call the first operand the multiplicand and the second operand multiplier. In the first step, we perform a logical-and operation between the LSB of the multiplier and each bit of the multiplicand to produce the first partial product, and then shift it one bit right. In the second step, we perform

a logical-and operation between the second bit of the multiplier and each bit of the multiplicand, and then add it to the shifted first partial product to produce the second partial product. After that, we shift the second partial product one bit right. Similarly, in the third step, the third partial product will be generated, and then it will be shifted one bit right. In the fourth step, we obtain the final product after the fourth partial product is generated and shifted. By examining this example, we see that multiplication is actually composed of only three primitive operations. They are the logical-and operation, add operation and shift operation. Therefore, any circuit for a multiplier can be implemented by using logical-and gates, full adders and shifters.

5.2 A 4-bit Ripple Carry Array Multiplier

In section 5.1, we reviewed the basic principle of multiplication. In this section, we will start to design a non-pipelined 4-bit multiplier. There are two design alternatives. For the first approach, we will use one shifter, one non-pipelined 4-bit adder and several registers in our design. In step one, as soon as the first partial product is obtained, shift it one bit right and then put it into a specific register. Here, we call it the pp register, which stands for partial product register. In step two, move the number stored in the pp register and the second addend into a non-pipelined 4-bit adder. As soon as the adder gives a stable result, shift it one bit left and then place it into the pp register again. Now, the new number stored in the pp register is the second partial product. In the subsequent steps, step two will be repeated until the fourth partial result is generated and placed into the pp register. Using this approach, a multiplication will be completed in multiple clock cycles. Because there is only one 4-bit adder, it needs to be used repeatedly. One advantage of this approach is that it saves hardware resources by effectively reusing the adder, shifter and registers. However, the circuit speed of this approach is very slow. For a 32×32 multiplication, it will wait 32 clock cycles to get the final product. Aside from that, this approach can hardly be implemented using a pipelined structure since most of the blocks keep getting reused. In order to overcome the aforementioned weaknesses, we introduce the array multiplier [11]. Figure 5.1 shows a 4-bit array multiplier (the block diagram comes directly from Figure 1(c) in [12], but is altered for our purpose).

The multiplier consists of sixteen full adders that are organized as a rectangular array. Furthermore, lines are drawn across the rectangular array. Among them, there are four horizontal lines where each line represents one bit of the multiplier $b[3..0]$. Also, there are four slanted lines where each line

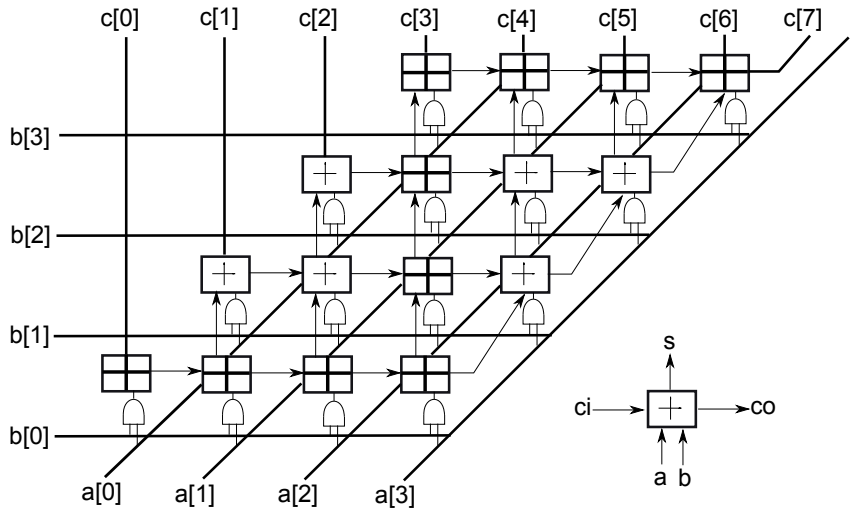

Figure 5.1: A 4-bit ripple carry array multiplier.

represents one bit of the multiplicand $a[3..0]$. At each intersection between a horizontal and slanted line, there is a full adder and a logical-and gate. The full adder is used for calculating the partial result. Besides the horizontal and slanted lines are eight vertically lines that denote the 8 bits product $c[7..0]$. It should be noted that the circuit shown in Figure 5.1 is a combinational logic. It employs four 4-bit ripple carry adders (sixteen full adders in total). Each ripple carry adder is in charge of the calculation of one partial product. The way in which adders are arranged makes a shifter unit unnecessary. Compared with the first design alternative, a multiplication product will be generated in one cycle rather than being calculated over several clock cycles.

5.3 A 4-bit Carry Look-ahead Array Multiplier

We could further speedup the multiplier shown in Figure 5.1 by replacing the full adder with a carry look-ahead adder[1] [9] [10]. In section 4.4, we presented a 4-bit ripple carry adder. It was concluded that the calculation of a higher indexed adder relies on the carry generated from a neighboring lower indexed adder. Since the carry is passed in a sequential fashion from a lower indexed adder to a neighboring higher indexed adder. However, a carry look-ahead adder breaks these sequential dependencies. The circuit of a 4-bit carry look-ahead adder contains two parts. One part is used to calculate addition results. This part is implemented by a number of half-adders. The other part is used for accelerating carry propagation. This part's design relies on the switching equations derived in section 4.4. The derivation process is traced below (the following formulas and reasoning come directly from [9] [10], but are altered for our purpose). First, we get five derived equations.

$$ci_1 = ci_1 \tag{5.1}$$
$$ci_2 = a[1]ci_1 + b[1]ci_1 + a[1]b[1] \tag{5.2}$$
$$ci_3 = a[2]ci_2 + b[2]ci_2 + a[2]b[2] \tag{5.3}$$
$$ci_4 = a[3]ci_3 + b[3]ci_3 + a[3]b[3] \tag{5.4}$$
$$co_4 = a[4]ci_4 + b[4]ci_4 + a[4]b[4] \tag{5.5}$$

Each equation is used for describing an input variable ci. For equation 5.2, $a[1]$ represents the second bit of operand a, $b[1]$ represents the second bit of operand b, and ci_1 denotes the input variable ci of FA_1. Similarly, ci_2 denotes the input variable ci of FA_2. After merging similar items, we get the following equations:

$$ci_1 = ci_1 \tag{5.6}$$
$$ci_2 = (a[1] + b[1])ci_1 + a[1]b[1] \tag{5.7}$$
$$ci_3 = (a[2] + b[2])ci_2 + a[2]b[2] \tag{5.8}$$
$$ci_4 = (a[3] + b[3])ci_3 + a[3]b[3] \tag{5.9}$$
$$co_4 = (a[4] + b[4])ci_4 + a[4]b[4] \tag{5.10}$$

[1]Our carry look-ahead array multiplier is designed without referencing the literature, however, since we did not conduct a thorough bibliographical search, they might have been designed and applied by others already.

When we replace the (a[i] + b[i]) and (a[i]b[i]) terms with new variables pi and qi respectively, equations 5.6, 5.7, 5.8, 5.9, 5.10 are changed to:

$$ci_1 = ci_1 \tag{5.11}$$
$$ci_2 = p_1 ci_1 + q_1 \tag{5.12}$$
$$ci_3 = p_2 ci_2 + q_2 \tag{5.13}$$
$$ci_4 = p_3 ci_3 + q_3 \tag{5.14}$$
$$co_4 = p_4 ci_4 + q_4 \tag{5.15}$$

By replacing variable ci_2 in equation 5.12 with equation 5.13, we get:

$$ci_3 = q_2 + p_2 q_1 + p_1 p_2 ci_1 \tag{5.16}$$

Similarly, by replacing variable ci_3 in equation 5.14 with equation 5.16, we get:

$$ci_4 = q_3 + p_3 q_2 + p_2 p_3 q_1 + p_1 p_2 p_3 ci_1 \tag{5.17}$$

Finally, replacing variable ci_4 in equation 5.15 with equation 5.17, we get:

$$co_4 = q_4 + p_4 q_3 + p_3 p_4 q_2 + p_2 p_3 p_4 q_1 + p_1 p_2 p_3 p_4 ci_1 \tag{5.18}$$

By observing equations 5.11, 5.12, 5.16, 5.17, 5.18, we realize that the calculation induced by any input variable ci depends only on the two operands (a and b) and ci_1. When they are given, all of the carries can be computed in parallel at the same time. Obviously, this technique improves the speed of an addition operation remarkably. However, the circuit for a carry look-ahead adder is much more complex than for the ordinary adder shown in Figure 4.3. A 4-bit carry look-ahead adder is shown in Figure 5.2(the block diagram comes directly from Quartus II design software, but is altered for our purpose).

In Figure 5.2, the circuits used for accelerating carry propagation are placed beside the four half-adders. These circuits are designed from equations 5.11, 5.12, 5.16, 5.17, 5.18. Figure 5.3 illustrates a 4-bit carry look-ahead array multiplier (the block diagram comes directly from Figure 1(c) in [12], but is altered for our purpose).

Here, for simplicity, the circuit of a carry look-ahead adder is not shown. Each rectangle in Figure 5.3 represents a 4-bit carry look-ahead adder. It is important to note that carries generated between each rectangle are still propagated to its neighboring rectangle in a sequential fashion. Carries generated within each rectangle, however, are propagated in parallel due to the carry propagation accelerating circuits.

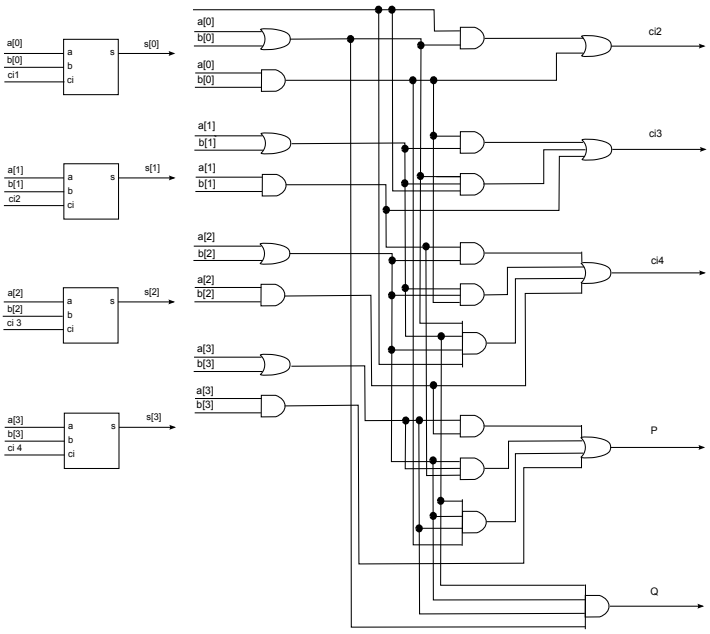

Figure 5.2: A 4-bit carry look-ahead adder.

5.4 A 4-bit Carry Save Array Multiplier

A carry look-ahead adder increases the calculating speed by using complex circuits. Thus, it consumes more hardware resources than an ordinary adder. Surprisingly, there exists an algorithm called the carry save algorithm [9] [10], that can not only improve the speed, but also save considerable hardware resources. Unlike the carry look-ahead adder, the carry save adder merely changes the way that carries are propagated between two neighboring full adders. It does nothing to optimize the full adder itself. Figure 5.4 illustrates a 4-bit carry save array multiplier (the block diagram comes directly from Figure 1(c) in [12], but is altered for our purpose).

You might notice that carries are not propagated horizontally in Figure 5.4. The adder which is placed at the intersection of line a_0 and line b_1 generates two output signals, s and co. s will be passed to the next vertically adjacent adder, but co, on the other hand, will not be propagated to the adjacent adder on the right. It will be propagated to the next upper right neighboring adder instead. This makes adders on the same row produce two

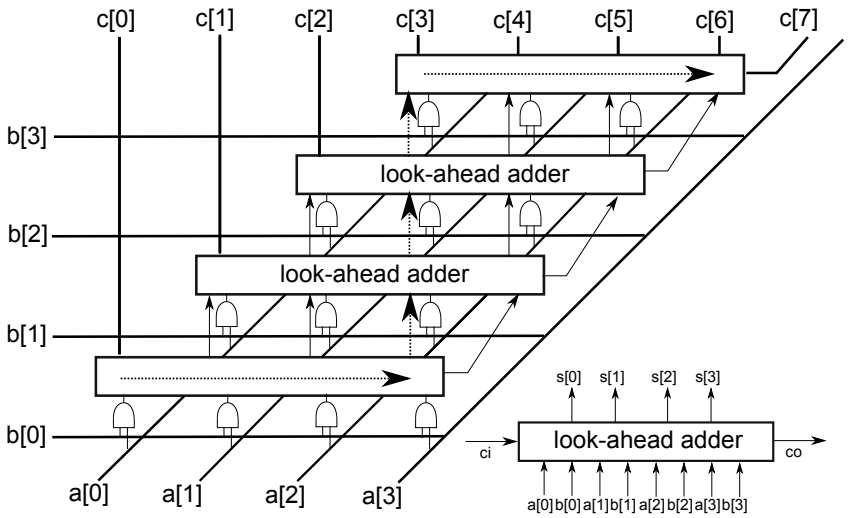

Figure 5.3: A 4-bit carry look-ahead array multiplier.

outputs for the next row. Notice that the adders on the fourth row from the bottom in Figure 5.4 generate two outputs. Since the fourth row is regarded as the last row, an additional adder is required. Here, we call it the final adder. The final adder is used for adding the two outputs generated from the forth row together to produce the final product. To further accelerate the speed, we could replace the final adder with a carry look-ahead adder.

5.5 Performance Comparison

In this section, we intend to analyze the calculating speed for the aforementioned three array multipliers (as presented in [14]). The longest path for each multiplier has been emphasized in Figure 5.1, Figure 5.3 and Figure 5.4 respectively. This allows us to find the calculating speed of a multiplier by measuring its longest path latency. In Figure 5.1, the longest path is highlighted by bold lines. It covers ten full adder blocks. Here, we treat each full adder block as a unit of computation so the longest path delay will be measured by the number of full adders involved. From this we can infer that the longest path for an

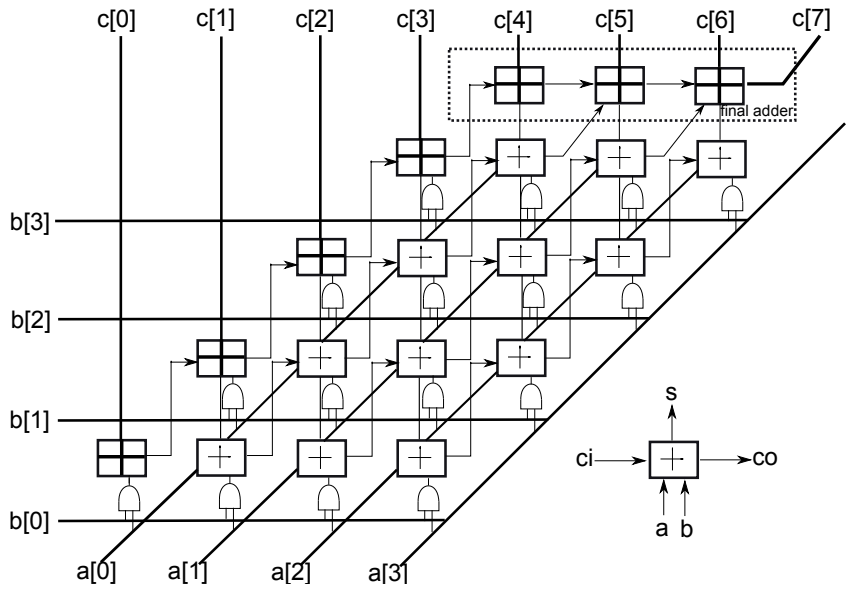

Figure 5.4: A 4-bit carry save array multiplier.

N-bit array multiplier will contain $3N-2$ full adder blocks. In Figure 5.3, the longest path is highlighted in a dashed line. You might notice it is very similar to the path drawn in Figure 5.1. This is because the carries are still propagated in a sequential fashion between two adjacent rectangles even though carries generated within each rectangle are propagated in parallel. Here, the exact number of full adder blocks involved in the longest path is not given due to the fact that a carry look-ahead adder does not contain any full adders. Instead, it is implemented by half-adders and carry propagation accelerating circuits. In Figure 5.4, the longest path is highlighted by bold lines as well. It covers seven full adder blocks. From this we can infer that an N-bit array multiplier's longest path will $2N-1$ full adder blocks. By comparison, we know that the array multiplier based on full adders is the slowest one among the three. Both of the other two multipliers are very fast. However, the carry save array multiplier is more appropriate for our design since it is easier to convert into a pipelined multiplier and it costs less hardware resources.

5.6 A Pipelined Carry Save Array Multiplier

By comparing the features of the aforementioned three multipliers, we have learned that the array multiplier based on the carry save algorithm is the best choice for our design. In this section, we will convert it into a pipelined multiplier. Figure 5.5 illustrates a 4-bit pipelined carry save array multiplier[2] (the block diagram comes directly from Figure 1 in [13], but is altered for our purpose).

For the multiplier shown in Figure 5.4, full adders appearing on the same row are incorporated into one separate stage. Therefore, besides the final adder, the remaining circuit is made up of four stages. Each stage produces two outputs. One output is the partial result, the other is the partial carry. For the final adder, each full adder is made as one separate stage, so it contains three stages in total. The workloads between these stages are fairly even since each stage does exactly the same thing. The latency of one stage is equal to the latency of a D-flip flop plus the latency of a full adder. Similarly, a floating point pipelined multiplier can be designed in the same fashion. It can be implemented by two pipelines. One pipeline is used to perform multiplication between the two mantissas, the other is used to perform addition between the two exponents. These two pipelines are executed in parallel. If one pipeline finishes earlier than the other one, the generated results are kept in a number of registers until the other pipeline is done.

[2]Our 4-bit pipelined carry save array multiplier is designed without referencing the literature, however, since we did not conduct a thorough bibliographical search, it might have already been designed and applied by others.

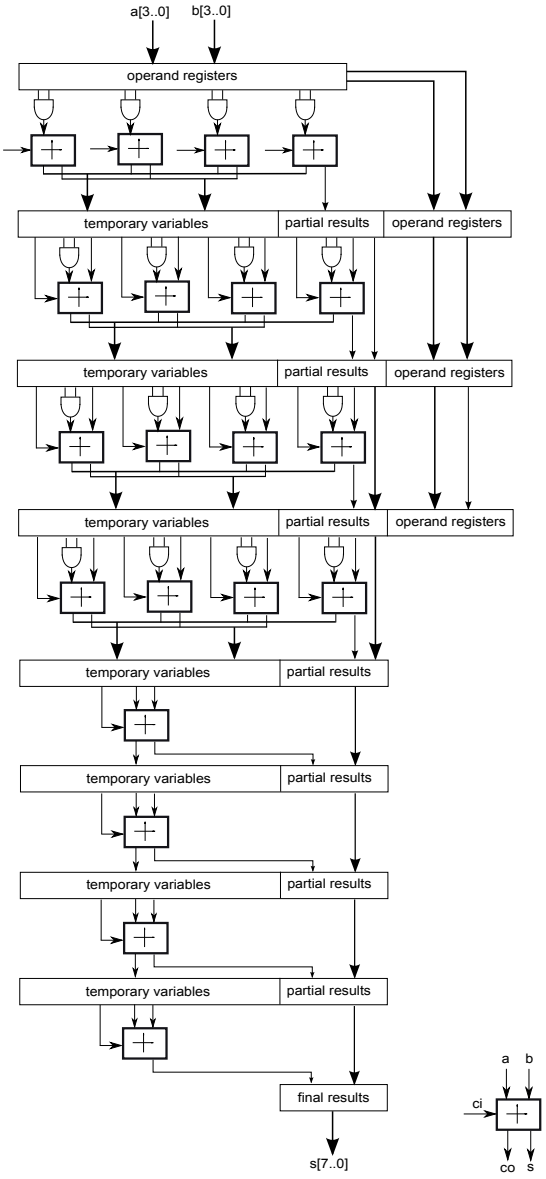

Figure 5.5: A 4-bit pipelined carry save array multiplier.

Chapter 6

Pipelined Read-only Memory

Read-only memory plays an important role in our design since the approach we used to implement the logarithm is table based. Normally, a table is implemented as ROM or RAM. Retrieve time for an item from a table is equal to the time required to access and retrieve the content of a specific address from a ROM or RAM. Thus, memory access and retrieve time has significant influence on the overall performance. In fact, there are two methods to speedup memory access and retrieve time. One method aims to reduce the table size to the largest possible extent since the smaller a table is, the less access and retrieve time it entails. A large memory usually spends a lot of time on address decoding. The other method tries to apply the pipeline technique to exploit parallelism between the memory access and retrieve steps. For our implementation, both of the two methods are adopted. Before we introduce pipelined read-only memory, we will first present the principle behind and architecture of SRAM and ROM [15].

6.1 SRAM Principle and Architecture

When treating SRAM as a black box, it usually has four input ports and one output port. Figure 6.1 illustrates a 32×8 SRAM (the block diagram comes directly from Quartus II design software, but is altered for our purpose).

The four input ports are: CLK, ADDRESS, DATA and WREN. The CLK port is connected to a clock signal. Note that SRAM can be triggered by either the rising or falling edge of a clock. The ADDRESS port holds the address of a specific memory location. The DATA port holds the data that will be written into the memory location specified by the ADDRESS port.

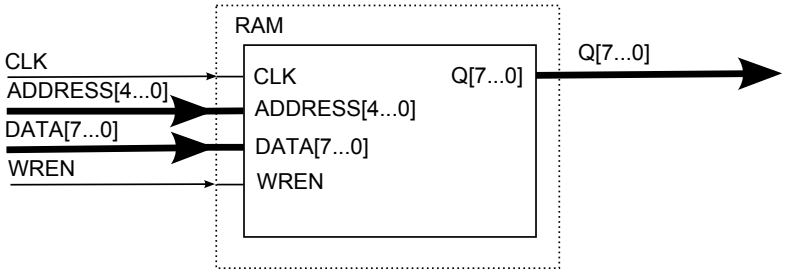

Figure 6.1: A 32×8-bit SRAM

The WREN port stands for write enable. When it is held low (logical-0), then a read operation is performed. Otherwise, if it is held high (logical-1), a write operation is performed. The only ouput port, port Q, produces the contents of a specific memory location.

Within the black box, the organization of SRAM can be divided into three parts: the row decoder, memory cell array and column decoder. Figure 6.2 summarizes the organization of a 256 bit SRAM. It has eight address lines in total (the block diagram comes directly from Fig.1.4.and Fig.1.5(b) in [15], but is altered for our purpose.).

Row Decoder

A row decoder is used to select one row of memory cells in the memory cell array. In Figure 6.2, the row decoder is implemented as a 1 of 32 decoder. It has five inputs and thirty-two outputs. When a read operation is performed, a specific 8-bit address is placed in the ADDRESS port. Then, the high five bits of the address are taken as inputs to the decoder. By decoding, only one of the thirty-two outputs is held high which in turn will activate one of the many wordlines in the memory cell array. Both the row decoder and column decoder are essential parts of SRAM because their speed and the power they consume have significant influence on the performance of SRAM. Ideally, we should avoid using a large decoder, since the larger a decoder becomes, the longer its latency.

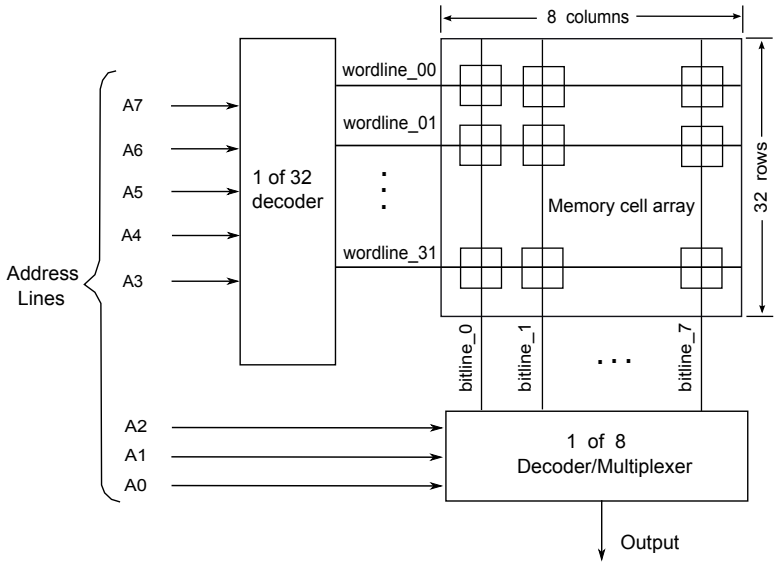

Figure 6.2: A 256-bit SRAM.

Memory Cell Array

The memory cell array is organized as a matrix, with each element positioned within an SRAM cell. Each cell can store a single bit-1 or 0. Cells which are on the same row are connected by a horizontal line (Wordline). Also, cells which are on the same column are connected by a vertical line (Bitline). In Figure 6.2, the 256 bit Memory cell array is organized as a matrix with thirty-two rows and eight columns. Therefore, thirty-two wordlines and eight bitlines are required. When one of the thirty-two wordlines is held high, all eight memory cells connected by that wordline are activated. In a read operation, data stored in those memory cells will flow out via the bitlines. In a write operation, data will flow into those memory cells via the bitlines.

Each SRAM cell is made up of transistors and simple analog circuits. Either a bipolar or MOSFET can be used to fabricate an SRAM cell. Compared to a bipolar transistor, a MOSFET has a number of advantages (as presented in [15]). First, it consumes less power. Second, it achieves high packing density. Finally, fabrication of a MOSFET has fewer steps than re-

quired for a bipolar transistor. Also, MOSFET fabrication is much simpler that that required for a bipolar transistor. An SRAM cell normally contains six MOSFET. Figure 6.3 illustrates a 6-transistor SRAM cell (the block diagram comes directly from Fig.1.23(b) in [15], but is altered for our purpose). The four transistors in the center are used to store data. The other two transistors on each side are used to activate the current SRAM cell. They are connected to both the wordline and bitline.

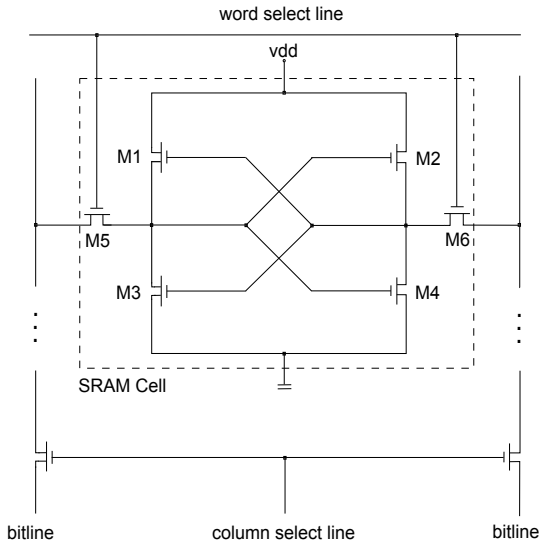

Figure 6.3: A 6-transistor SRAM cell.

Column Decoder

After data flows out from the memory cell array, it needs to be further decoded. For the 256 bit SRAM shown in Figure 6.2, there are eight bitlines. Only one of the eight bitlines is selected as output. Thus, the column decoder is implemented as a 1 of 8 decoder. It has three inputs and eight outputs. Each of the outputs serves as a select signal, it is used to control a specific bitline. The function of a select signal is similar to a switch. When a switch is turned on, data on the bitline it controls will flow out. Otherwise, when the switch is turned off, data on the bitline it controls will be blocked. As a result, only one bitline's data will flow at any time while others will be blocked. In fact, more than one approach can be used to implement a column decoder. In some designs, a multiplexer is preferred.

6.2 ROM Principle and Architecture

The principle underlying and architecture of ROM is similar to RAM. Figure 6.4 shows a 32×8 ROM (the block diagram comes directly from Quartus II design software, but is altered for our purpose).

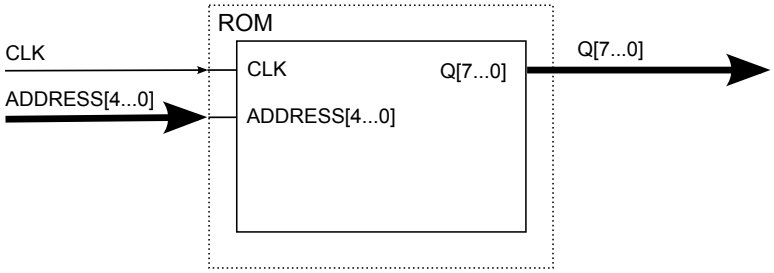

Figure 6.4: A 32×8-bit ROM.

A ROM has two input ports, CLK and ADDRESS, and one output port, Q. The function of each port is exactly the same as those shown in Figure 6.1. ROM also consists of three parts: the row decoder, memory cell array and column decoder. The memory cell array is organized as a matrix as well. It needs row and column select circuits to select the desired data. Like RAM, either bipolar or MOSFETs can be used to fabricate a ROM cell. Each SRAM cell is normally composed of 6 transistors. Each ROM cell, on the other hand, only needs one MOSFET. Figure 6.5 shows an 8×2-bit ROM with a 4×4-bit geometry matrix (the block diagram comes directly from Fig.6.17 in [16], but is altered for our purpose).

The memory cell array shown in Figure 6.5 involves four rows, with each row storing 4 bits of data. To store bit-0, a transistor needs to be placed at the appropriate position. No transistor is needed to store bit-1. In the figure, the data stored in each row are 0001,0110,1010 and 0011, starting from the top. Since the logical structure of the memory cell array shown in Figure 6.5 is 8×2-bit, only three address lines are required, called $A[2..0]$. Here, A_2 is the most significant bit (MSB). A_0 is the LSB. A_2 and A_1 are inputs to the row select circuit. Here, the row select circuit is actually a 2 of 4 decoder. It produces four outputs which are used in turn to connect the gates of the MOSFETs of each row. The function of the horizontal lines across the memory

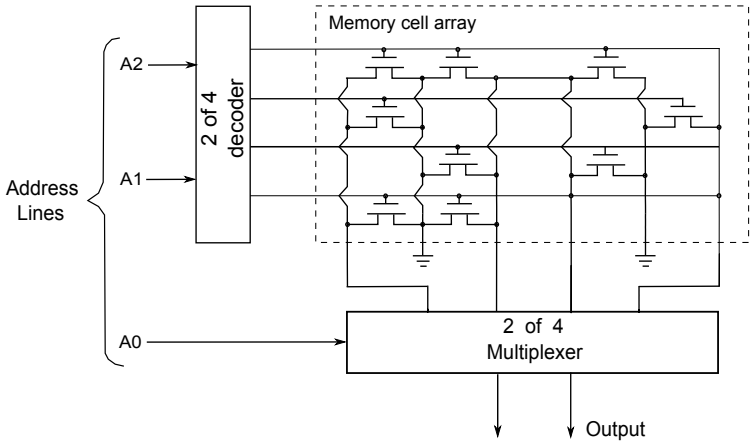

Figure 6.5: An 8×2-bit ROM with a 4×4-bit geometry matrix.

cell array is similar to that of wordline in RAM. A_0 is the input to the column select circuit which is implemented as a 2 of 4 multiplexer. It generates two outputs, which are the final outputs for this ROM. The function of the vertical lines across the memory cell array is similar to that of bitline in RAM. In a read operation, suppose $A[2..0]$ is assigned to 100 in binary. Data stored on the third row will be activated. In the column select circuit, transistors used to control the second and forth columns will be set. As a result, data stored in the second and forth columns of the third row will flow out and then trigger the multiplexer.

6.3 Discussion

In the previous sections, we introduced the principles and architecture of SRAM and ROM. We highlighted that memory design is not limited to digital circuits, say, a decoder or multiplexer, but also contains analog circuits such as the circuits involved in the memory cell array.

Since our CAD tool, the Altera Quartus II 7.2 design software, only supports digital circuit design, we need to convert analog circuits into digital

circuits. One of our attempts tries to replace the cells within the memory cell array with D flip-flops without altering the three-part memory organization. In fact, a wordline within a real memory chip usually drives hundreds or thousands of cells. In reality, the output is not able to drive more than eight devices since digital circuits are TTL (transistor-transistor logic) devices which need to comply with the fanout rule. Analog circuit design, on the other hand, is more flexible. In order to drive many devices, an engineer needs to carefully calculate a series of factors (eg. capacitance, leakage current) and then choose the appropriate devices to design the desired circuit.

Our first attempt does not seem feasible since the converted circuits would violate the fanout rule. Thus, we need to consider this problem from a different perspective. Our solution is to try to design a digital circuit which has equivalent functionality of the ROM, and then modify this circuit with a pipelined structure.

6.4 Motivation for Pipelined ROM

In order to reduce the time spent accessing and retrieving items from memory, we have two design alternatives. The first approach tries to speedup memory access and retrieval without changing the three-part memory architecture. It depends largely on the improvement of hardware techniques. In [17], Jean suggests shrinking circuit component to sizes down to the nanometer range to get faster, more powerful and efficient electronic circuits. The second approach tries to reorganize the memory architecture and apply the pipeline technique to exploit the parallelism between the memory access and retrieve steps. The second approach is more appropriate for our design. In order to adapt memory to a pipeline structure, we intend to split the memory into a number of components with a balanced workload. Furthermore, all of the components are expected to be implemented as digital circuits. We prefer to design these components by using basic logic gates since it can better control the way that the memory is pipelined.

In summary, to design pipelined memory, we need to solve three problems. First, how to reorganize the memory architecture so that it can use a pipelined structure? Second, how to split the memory into a number of components with a balanced workload? Lastly, how to design those components by using basic logic gates? In the next section, we will find a solution for each of them.

6.5 A 64×8-bit Pipelined ROM

In this section, we present the structure of a 64×8-bit pipelined ROM[1]. The reason why we choose ROM rather than RAM is that once the lookup table has been loaded, it will not change. Thus, ROM is more appropriate. Figure 6.6 shows the schematic for a 64×8-bit pipelined ROM.

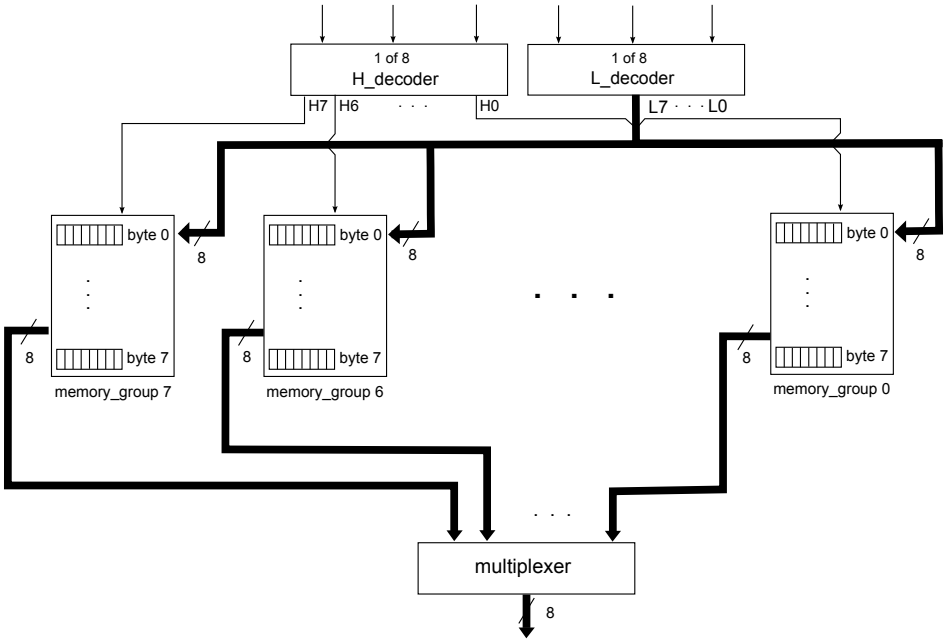

Figure 6.6: Schematics of a 64×8-bit pipelined ROM.

In Figure 6.6 there are six address lines. Thus, two 1 of 8 decoders are needed: the H_decoder and L_decoder. The high three address lines are taken as inputs of the H_decoder. The low three address lines are taken as inputs of the L_decoder. The 64 bytes are organized as eight memory groups. The H_decoder is used to select memory groups. Logical-0 indicates the specified group is not selected, while logical-1 indicates it is selected. The L_decoder, on the other hand, is used to select bytes. Logical-0 indicates the specified byte is not selected, while logical-1 indicates it is selected. After group and byte selection, only the selected byte's value will be output. The two decoders

[1]Our pipelined ROM is designed without referencing any literatures. However, since we do not conduct a thorough bibliographical search, they might be designed and applied by others already.

shown in Figure 6.6 are similar to the row and column decoders used in ordinary RAM. The structure shown in Figure 6.6 has several advantages. First, it can all be implemented by digital circuits. Second, if treating it as a black box, it has the same input and output ports as an ordinary ROM. Third, it is very easy to adapt into a pipeline structure.

Chapter 7

Conclusion

In conclusion, we have demonstrated how to implement single precision floating point logarithm using a pipelined architecture and using only digital circuits. This platform-independent approach enables us to get a fairly high throughput. Unlike previous methods, we did not present the trade-off between lookup table size and accuracy with respect to their affects on speed. In other words, our logarithm design is completely based on fixed accuracy and fixed speed. We have estimated that it can generate 2.9G single precision values per second under a 65nm fabrication process. We have also shown that the lookup table contains 648 entries and occupies roughly 7.776KB of memory space while achieving at least 21 bits precision. Using our design, the performance of log-based applications can be improved by a large extent. In this thesis, the presented hardware components (e.g., Pipelined Adder, Pipelined Multiplier, Pipelined Read-only Memory) were designed without referencing the literature. However, since we did not conduct a thorough bibliographical search, they might have been designed and applied by others already.

Our approach can be extended to implement a double precision logarithm. We could either extend the algorithm from single to double precision or take the single precision implementation as a kernel to design a double precision algorithm. Additionally, techniques presented in this thesis can be adapted to other frequently used math functions, like sin, cos, etc.

There are still a number of things that might be improved in the near future. It is also possible to further reduce the pipeline latency. To this end, we could either refine our hardware components, say, the multiplier, by using more advanced algorithms or make more hardware components execute in parallel.

The pipelined logarithm implementation presented in this thesis is just a start. We expect that with time all of the frequently used math functions will

be implemented in hardware and then be incorporated into a CPU as arithmetic instructions, so that the performance of many scientific computations will be improved dramatically.

Bibliography

[1] Oriol Vinyals, Gerald Friedland. A Hardware-Independent Fast Logarithm Approximation with Adjustable Accuracy, *Tenth IEEE Inter. Symposium on Multimedia*, on pages 61-65, 2008.

[2] Altera Stratix IV Device Handbook, Also on `http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf`

[3] Florent de Dinechin, Mioara Joldes, Bogdan Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation, *21st IEEE Inter. Symposium on Application-specific Systems Architectures and Processors*, on pages 216-222, 2010.

[4] Nikolaos Alachiotis, Alexandros Stamatakis. Efficient Floating-Point Logarithm Unit for FPGAs, *2010 IEEE Inter. Symposium on Parallel & Distributed Processing*, on pages 19-23, 2010.

[5] Nikolaos Alachiotis, Alexandros Stamatakis. FPGA Optimizations for a Pipelined Floating-Point Exponential Unit, *ARC'11 Proceedings of the 7th international conference on Reconfigurable computing*, on pages 316-327, 2011.

[6] I. of Electrical and E. Engineers.IEEE 754-1985: Standard for Binary Floating-Point Arithmetic, 1985. Also on `http://grouper.ieee.org/groups/754/.`

[7] R.H.Katz, G.Borriello. *Contemporary Logic Design*. Prentice Hall, First Edition, 2005.

[8] M.M.Mano, C.R.Kime. *Logic and Computer Design Fundamentals*. Pearson/Prentice Hall, Upper Saddle River, NJ, 2004.

[9] Israel Koren. *Computer Arithmetic Algorithms*. A K Peters/CRC Press, Second Edition, 2001.

[10] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press,1999.

[11] Yongjiang Jiang. *Design of Computer Kernel Components based on Quartus II*. Tsinghua University Press, First Edition, 2007.

[12] Zheng Ying, Jin Wu, Changyuan Chang, Tongli Wei. Comparison of High Speed Multipliers. *Chinese Journal of Electron Devices, Microelectronic Center, Southeast University, Nanjing 210096*. P. R. China, 2003.

[13] Feng Liang, Zhibiao Shao, Haijun Sun. Design of 43-bit Floating-Point Pipelined Multiplier. *Chinese Journal of Electron Devices, School of Electronics and Information Engineering , Xi'an Jiaotong University , Xi'an710049*. P. R. China, 2006.

[14] Jing Chen, Ning Liu, Zhe Deng. Comparison of Multipliers. *Undergraduate Research Project, College of Information Engineering, Capital Normal University*. Beijing, P. R. China, 2008.

[15] Kiyoo Itoh. *VLSI Memory Chip Design*. Springer, First Edition, 2011.

[16] Gerald Luecke, Jack P. Mize, William N. Carr. *Semiconductor Memory Design and Application*. McGRAW-HILL Book Company, 2011.

[17] Rand K. Jean. *Electrical Characteristics of 65-Nanometer MOSFETs*. Also on `http://nanohub.org/resources/796/download/2003_suri_jean_abstract.pdf`

[18] Cygwin Official Site, `http://www.cygwin.com/`

[19] GNU Official Site, `http://www.gnu.org/`