Stony Brook University

Parallel and Flexible Hardware Implementation of Fletcher Checksum

A Thesis presented

by

Maria Isabel Mera Collantes

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Electrical Engineering

Stony Brook University

May 2014

Stony Brook University

The Graduate School

Maria Isabel Mera Collantes

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis

Peter Milder, Ph. D. - Thesis Advisor
Assistant Professor, Department of Electrical Engineering and Computer Engineering

Emre Salman, Ph. D. - Second Reader
Assistant Professor, Department of Electrical Engineering and Computer Engineering

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

Parallel and Flexible Hardware Implementation of Fletcher Checksum

by

Maria Isabel Mera Collantes

Master of Science

in

Electrical Engineering

Stony Brook University

2014

Checksums are utilized in many contexts such as communications, storage and reliable processing. The balance between checksum strength, implementation cost and obtained throughput often pose a challenge for present day system designers. In this research we propose two new methods for implementing the Fletcher Checksum (FC) in a parallelized context. We determined an extended parallel definition from the original FC and applied it to two different hardware implementation approaches. We then created a generator that would automatically output parameterized designs. We controlled the input word length, number of parallel inputs and architecture of the designs, and we then synthesized these designs for FPGA and ASIC. Our results show that parallelization of FC is feasible and the system throughput is proportional to the cost defined by resources used, area and power consumption. In our results, we demonstrate designs with throughput up to 375 Gbits/sec in ASIC and up to 110 Gbits/sec in FPGA, depending on the specific parameters.

Dedication Page

To Mariana and Cristóbal, for their unconditional support.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Checksums provide a manner for verifying the integrity of data being transmitted or processed in a system. They are used in communications [1–3], data storage [4] and reliable processor systems [5,6]. The concept of checksum is loosely used to term an error detection code appended to the transmitted data sequence. The calculations that are performed to obtain the code value are usually, but not limited to, summations. There are several checksums that provide varying degrees of error detection effectiveness, and, as is expected, different implicit trade-offs. These often depend on their implementation algorithm, nature of processed data, type of errors to be detected, intended hardware platform, and combinations of these.

Arithmetic checksums are generally considered to be lower-cost than alternative methods [3, 7, 8] when they are implemented in software. One such checksum is the Fletcher Checksum [3]. The Fletcher Checksum is particularly interesting because it has better error-detecting capabilities than other simple checksums, yet is still comprised of simple arithmetic. This is a serial addition-based checksum algorithm that takes a series of input words one at a time and computes two verification values. It is imperative for us to propose solutions that can offer more processing speed due to the ever growing amount of data that present day systems are managing especially in contexts like execution stream compression for reliable systems, where many bytes of data potentially need to be processed per clock cycle. One way to accomplish this is to assign fixed computation-heavy tasks to dedicated circuits as well as increasing data rate processing, so we considered parallelization in hardware.

In this thesis, our aim is to determine the feasibility and trade-offs of implementing the serial Fletcher Checksum into a parallel design in hardware. First, we extend the serial checksum definition into parallel format. After that, we establish two different methods for parallelizing the Fletcher Checksum in order to improve the throughput of the system. Then, we create a generator that outputs synthesizable Verilog files. Last, we evaluate various designs that offer certain advantages at different costs and present a comparison.

For comparison of the costs and benefits of both parallel methods, we

synthesized and evaluated a wide space of automatically generated Fletcher Checksum designs targeting FPGA (Altera Arria II) and ASIC (45 nm standard cell) implementations. We propose two different designs that scale easily. The first is purely combinational, and the second is characterized for its sequential computations which reduce the need for repetitive logic. The specific implementations that are compared in our work depend on the following parameters: architecture, input word length, and number of inputs.

Through the realization of this work, we contribute information on an alternative scalable method for executing a well known checksum on hardware. We provide a parallel definition of the Fletcher Checksum, an HDL code generator for the checksum, a quantitative comparison of different parallel designs, and conclusions regarding hardware implementation.

Prior work in the literature has studied the performance, effectiveness and cost of the implementation of various checksums (including the Fletcher Checksum) in software as well as recommendations on implementations. Stone et al. have examined the behavior of checksums over real data [9]. Nakassis provided insight on implementation pitfalls and improvements that could be made to the Fletcher Checksum in [10]. Fletcher checksum input bit values of 8, 16 [11], 32 and 64 [7] have also been studied and compared to other checksums.

Maxino and Koopman [8] have compared the trade-offs between several checksum algorithms and and have shown that the Fletcher Checksum offers less effective error detection and is less computationally costly than cyclic redundancy codes which are more efficient at detecting errors. Yet, when compared to other methods, such as exclusive or checksum, one's and two's complement addition checksum, and Adler Checksum, the authors recommend using the Fletcher Checksum for non bursty data transmissions. These analyses are done in software with the algorithms tested to run on general purpose processors.

Checksums are used in communications, embedded networks and other applications. Due to the Fletcher Checksum being sensitive to sum order, it has an application of being used to trace data paths in wireless sensor networks (WSN) for fault detection [12]. It is also taken into consideration in improvements of existing protocols because it is more computationally ef-

ficient in software implementations than other checksums [13].

So far, most hardware implementations of checksums are focused on CRCs [6, 14–22], but very few people have explored flexible Fletcher Checksum hardware implementations.

In Chapter 2 we explore the original Fletcher Checksum and present a parallel version. Then, in Chapter 3 we determine different ways in which to implement the parallel version in hardware. In Chapter 4 we present a generator created to test our designs and we discuss the results we obtain from synthesis. Lastly, we state our findings in Chapter 5.

A preliminary version of this work was included in [23].

Chapter 2 The Fletcher Checksum

2.1 Concept

The Fletcher Checksum is an error detection method that takes an input sequence of B bit numbers and produces two B bit checksums for which the present state is a function of all past inputs [3]. We will refer to these two B bit checksums as s and t. Both s and t are arithmetic checksums calculated by adding cumulative input values, x_ℓ, which can be interpreted as integers. The addition can be performed with modulo 2^B (two's complement) or $2^B - 1$ (one's complement) arithmetic, with B being the number of bits in each word. One's complement arithmetic can be carried out by doing unsigned arithmetic followed by a modulus operation, which is performed by adding the overflow bit back into the B lower bits of the sum. For our implementations we chose one's complement arithmetic instead of two's complement because it offers better error detection [8], but our system can easily be reconfigured to support two's complement arithmetic.

Let x_ℓ represent the input word at time ℓ. Then, the two Fletcher Checksum outputs s and t are calculated according to:

$$s_{\ell+1} \leftarrow s_\ell + x_\ell \tag{1}$$

$$t_{\ell+1} \leftarrow t_\ell + s_\ell \tag{2}$$

To compute the checksums we first initialize all the variables to zero ($s_0 = 0$, $t_0 = 0$). Then, the first input value is added to the first result of the checksum. Consequently this checksum result is added to the second checksum result. In this manner we compute the checksum results for each new word. In other words, s is the sum of all prior and current inputs x, and t is the sum of all prior and current values of s. The first and second checksum results, s and t, are the same bit size as the input words x. The total checksum result is the concatenation of the last two computed checksum values of s and t.

For example, the first four checksum values can be calculated as follows. Recall, all arithmetic operations are performed modulo $2^B - 1$.

$$
\begin{aligned}
s_0 &= 0 & t_0 &= 0 \\
s_1 &= s_0 + x_1 = x_1 & t_1 &= t_0 + s_1 = x_1 \\
s_2 &= s_1 + x_2 = (x_1) + x_2 & t_2 &= t_1 + s_2 = 2x_1 + x_2 \\
s_3 &= s_2 + x_3 = (x_1 + x_2) + x_3 & t_3 &= t_2 + s_3 = 3x_1 + 2x_2 + x_3 \\
s_4 &= s_3 + x_4 = (x_1 + x_2 + x_3) + x_4 & t_4 &= t_3 + s_4 = 4x_1 + 3x_2 + 2x_3 + x_4
\end{aligned}
$$

The result of each final checksum, after four inputs (x_4), are the values of s_4 and t_4. Generalizing this idea, we can represent the checksum values purely in terms of the inputs x.

$$s_\ell = \sum_{\ell=0}^{n} x_\ell \tag{3}$$

$$
\begin{aligned}
t_\ell &= \sum_{\ell=0}^{n} s_\ell \\
&= \sum_{\ell=0}^{n} \sum_{k=0}^{\ell} x_k \\
&= \sum_{\ell=0}^{n} (n - \ell + 1) x_\ell
\end{aligned} \tag{4}
$$

A serial hardware representation of the checksum and one's complement arithmetic is illustrated in Figure 1 and Figure 2. We perform the modulo $2^B - 1$ operation when we add the carry bit or bits back into the sum.

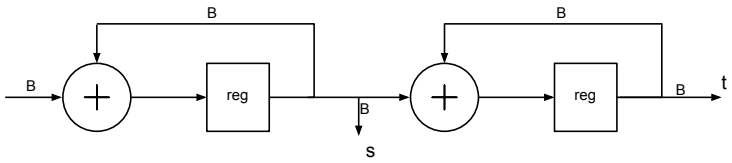


Figure 1: Hardware design of Fletcher Checksum

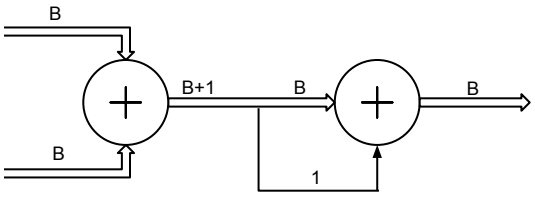


Figure 2: Hardware design of one's complement arithmetic

2.2 Mathematical Description of the Parallelized Fletcher Checksum

The Fletcher Checksum was originally created to process serially transmitted data; because of this it is an inherently serial set of operations. This means that each current output depends on the preceding output. For this reason a typical implementation of the checksum is limited to processing one B bit input word per clock cycle. Consider that if large amounts of input data must be processed (for example, when computing a fingerprint of a processor's internal state), the data must be serialized into B bit words. This limits the speed greatly. On the other hand, if we were to process more words per clock cycle, we would need multiple inputs, but in Figure 3 we see that multiple typical implementations would produce multiple checksum values instead of one overall value. For this reason the serial implementation of the checksum cannot scale to process data from more than one input.

In order to get past the limitation of only being able to process one input word per clock cycle, we have designed two new methods for parallelizing Fletcher Checksum computation that allows a system to have P parallel inputs. This system takes into consideration the dependencies on the previous iterations of the checksums and performs additional arithmetic operations to obtain the same final total checksum result.

As shown in Figure 3, we can see that when we implement P parallel instances of the Fletcher Checksum we obtain P particular checksum results s_i and t_i for each set of specific inputs. In this case P represents the parallelism and signifies the number of simultaneous inputs. Now, s_i and t_i, where $0 \leq i < P$, are the partial checksums calculated for each block. In order to obtain a single unified answer for s and t, we must perform arithmetic oper-

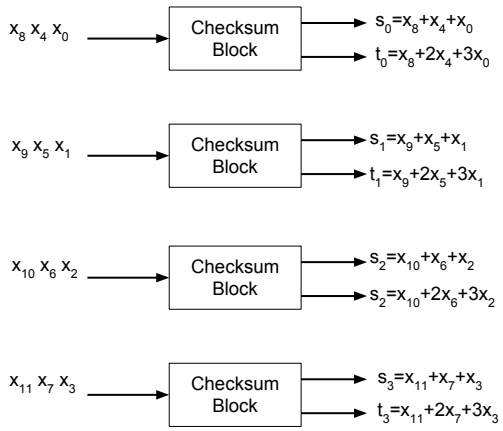


Figure 3: Four Parallel Checksum Blocks

ations on the single individual answers and combine the results in the same proportions as the serial implementation would. The final total checksums, s and t, that are calculated are the outputs that are the Fletcher Checksum.

By reformulating (3) and (4) we can obtain new expressions for s and t in terms of the partial checksums s_i and t_i according to (5) and (6).

$$s = \sum_{i=0}^{P} s_i \tag{5}$$

$$t = P \sum_{i=0}^{P-1} t_i - \sum_{i=0}^{P-1} i s_i \tag{6}$$

For example, let's take a system that has four inputs like the one in Figure 3. After three clock cycles we can apply the anterior equations and confirm their output matches the values obtained from (1) and (2).

According to (5):

$$
\begin{aligned}
s &= s_0 + s_1 + s_2 + s_3 \\
&= (x_0 + x_4 + x_8) + (x_1 + x_5 + x_9) + (x_2 + x_6 + x_{10}) + (x_3 + x_7 + x_{11}) \\
&= x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11}
\end{aligned}
$$

According to (6):

$$
\begin{aligned}
t =& 4(t_0 + t_1 + t_2 + t_3) - (0s_0 + 1s_1 + 2s_2 + 3s_3) \\
=& 4t_0 + 4t_1 + 4t_2 + 4t_3 - s_1 - 2s_2 - 3s_3 \\
=& 4(3x_0 + 2x_4 + x_8) + 4(3x_1 + 2x_5 + x_9) \\
& + 4(3x_2 + 2x_6 + x_{10}) + 4(3x_3 + 2x_7 + x_{11}) \\
& - (x_1 + x_5 + x_9) - 2(x_2 + x_6 + x_{10}) - 3(x_3 + x_7 + x_{11}) \\
=& (12x_0 + 8x_4 + 4x_8) + (12x_1 + 8x_5 + 4x_9) \\
& + (12x_2 + 8x_6 + 4x_1 0) + (12x_3 + 8x_7 + 4x_{11}) \\
& - (x_1 + x_5 + x_9) - (2x_2 + 2x_6 + 2x_{10}) - (3x_3 + 3x_7 + 3x_{11}) \\
=& 12x_0 + 11x_1 + 10x_2 + 9x_3 + 8x_4 + 7x_5 \\
& + 6x_6 + 5x_7 + 4x_8 + 3x_9 + 2x_{10} + x_{11}
\end{aligned}
$$

We have shown that (5) computes the same output as (1), and (6) computes the same output as (2).

Chapter 3 Parallel Implementation Design of the Fletcher Checksum in Hardware

3.1 Implementation Models

In Chapter 2 we saw that the Fletcher Checksum could be reconstructed into a parallel input format described by equations (5) and (6). In this section we will focus on how to carry this out in hardware domain. Our aim is to develop two designs that correctly output the corresponding checksum value and that offer different cost/throughput trade-offs.

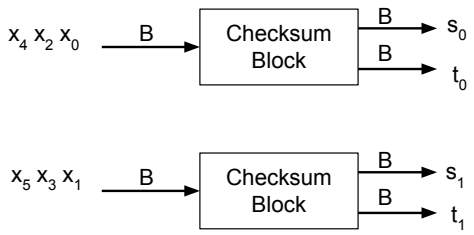
3.1.1 System Architecture: Binary Tree (2^n inputs)

For our first approach we developed a tree architecture design that outputs the checksum values s and t at every clock cycle. Increasing the problem size is straightforward because binary trees are easily scalable.

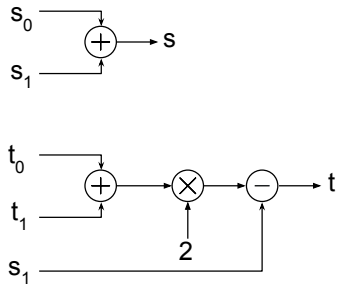
We can see that (5) and (6) are comprised of a combination of the partial checksums s_i and t_i. We can obtain s from a direct summation of the partial checksums s_i by implementing the adders in a tree structure. To calculate t, each partial checksum s_i is multiplied by its corresponding coefficient (ranging from 0 to $P-1$) and added to each other, also using a binary tree. Later, this result is subtracted from the result obtained from multiplying P times the summation of all partial checksums t_i. At every addition stage we perform the modulo $2^B - 1$ operation to account for the possibility of overflow. There is a constant output at every clock cycle once the initial values have finished processing.

The checksum block produces the partial checksums s_i and t_i and is composed of the design in Figure 1. The binary tree addition is implemented in Figure 5. The checksum block and the tree are composed of registers, adders, and multipliers. Due to the to the predominance of registers and logic involved in the implementation of the tree architecture, we can see that this scales linearly. As the number of inputs increases the structure of a perfect binary tree scales accordingly as is illustrated in Figures 4 and 5.

With this structure we studied two basic designs: one that is pipelined and one with no pipelining to determine the quantity of additional used resources

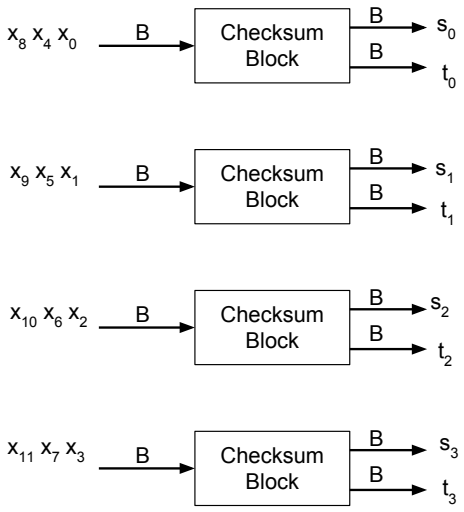


(a) Two basic checksum blocks are used to calculate partial checksum values.

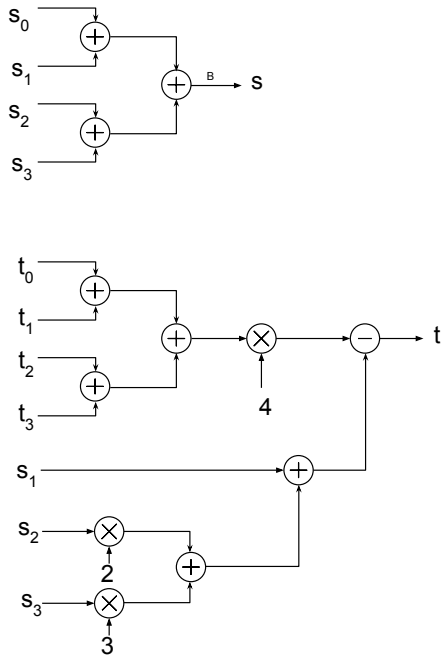


(b) From the two partial checksum values we calculate the complete checksum.

Figure 4: Two parallel input Fletcher Checksum tree design



(a) Four basic checksum blocks are used to calculate partial checksum values.



(b) From the four partial checksum values we calculate the complete checksum.

Figure 5: Four parallel input Fletcher Checksum tree design

and compare the trade-off between that and frequency. In the pipelined design, after each arithmetic unit, a register is added, with the exception of the sum corresponding to the partial checksum, in which adding a register would interfere with the feedback loop and give a sum of two incorrect inputs.

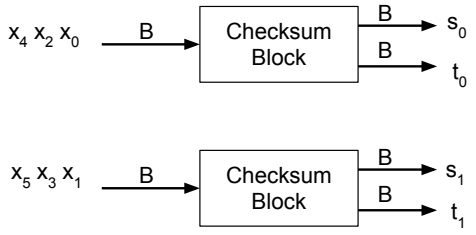
3.1.2 System Architecture: Sequential Processing Datapath

In order to reduce the amount of logic that would be implemented, we redesigned the structure into a sequential datapath of the processed signals. The first design produced an output of the checksum with every clock cycle, but since what is often needed is the checksum at a particular time, we can alter the implementation strategy to produce a more hardware-efficient circuit that will output the checksum on request. This would reduce overall cost but come with different trade-offs, such as increased number of clock cycles before a value is output. We implemented this alternate architecture by using a finite state machine which controls the capture of the values of the partial checksum and initiates the computation on these. Since the Fletcher Checksum has a feedback loop, and therefore memory, it is not necessary to calculate the final checksums during every single clock cycle or store previous results. All the information that is needed is stored within each partial checksum. We are able to calculate s and t using the mathematical formulas (5) and (6) described in Chapter 2 while implementing a sequential datapath controlled by a finite state machine (FSM).

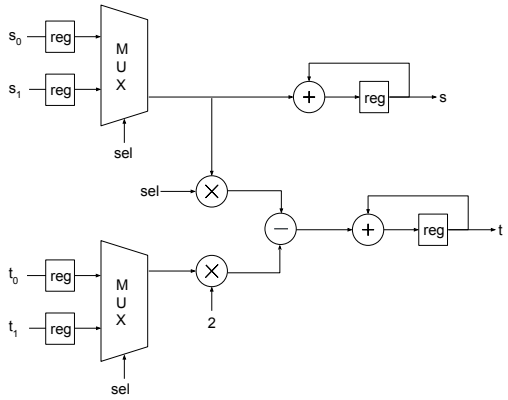
Figures 6 and 7 show the basic block diagram of this architecture. The registers preceding the multiplexer receive and store the partial checksums when an output is requested. These values are then selected by the multiplexer. For calculating s the partial checksums s_i are added to each other using a feedback loop. For calculating t the result of s_i multiplied by i is subtracted from the multiplication of the partial checksums t_i with P and lastly, this value is accumulated through a feedback loop. The FSM controls the output of a multiplexer and the variable value input to the multiplier. This same diagram serves designs for any number of inputs with only the multiplexer size increasing.

In summary, we replaced the adder trees with two multiplexers, one for each of the partial checksums, and an FSM that controls the multiplying

coefficient of s_i and the select signal of said multiplexers.



(a) Two basic checksum blocks are used to calculate partial checksum values.



(b) From the two partial checksum values we calculate the complete checksum.

Figure 6: Two parallel input Fletcher Checksum sequential design

Having previously designed a pipelined and non-pipelined version we noticed that not using pipelining caused operating clock frequencies that were low and not comparable to speeds of implementations of other algorithms used for similar purposes. For this reason we added pipelining stages to the design so that we could get results that could be compared to the previous architecture.

3.2 Estimation of Utilized Components for Each Architecture

The designs that have been implemented are straightforward and allow us to analyze the number of registers, adders, and constant multipliers. We can give an estimate of the area and how we expect it to scale in relation to P.

We use the equations described in (5) and (6) and include the adders that perform one's complement arithmetic for the modulo operation to obtain a relative value of registers, adders and multipliers. The sequential design uses one constant multiplier and one full multiplier, while the tree design uses several constant multipliers. The values relative to P are shown in Tables 1 and 2. If we compare the values in the Total row for each table we can quickly see that the there is more use of resources in the tree architecture.

computation	registers	adders	multipliers
P serial Fletcher Checksum units	$3P$	$2P$	0
$s = \sum_{i=0}^{P} s_i$	$P-1$	$P-1$	0
$P\sum_{i=0}^{P-1} t_i$	P	$P-1$	1
$\sum_{i=0}^{P-1} is_i$	$2P-4$	$P-2$	$P-2$
final subtraction	1	1	0
Total	$7P-4$	$5P-3$	$P-1$

Table 1: Estimated number of registers, adders and multipliers in tree structure design.

computation	registers	adders	multipliers	multiplexers
P serial Fletcher Checksum units	$3P$	$2P$	0	0
$s = \sum_{i=0}^{P} s_i$	2	1	0	1
$P \sum_{i=0}^{P-1} t_i$	3	1	1	1
$\sum_{i=0}^{P-1} i s_i$	2	1	1	0
subtraction	1	1	0	0
accumulation	1	1	0	0
Total	$3P + 9$	$2P + 5$	2	2

Table 2: Estimated number of registers, adders and multipliers in sequential structure design.

Chapter 4 Results

4.1 Fletcher Checksum Generator

Based on the architectures and considerations reviewed in Chapter 3 we developed a generator that automatically outputs specialized Fletcher Checksum modules in Verilog that are ready for synthesis. We are able to assign values to design parameters that influence the trade-offs between cost and throughput. These parameters are: checksum size, number of simultaneous inputs and pipelining.

We verified that our generator was working correctly by testing the output with testbenches we created. The testbenches apply uniform random numbers between 0 to $2^B - 1$ to the design, and an accompanying software implementation provides the correct values.

Noteworthy Features of the Implemented Designs

After creating both designs we have noticed there are a few interesting points regarding P which are the following:

- Generation for the partial checksums is exactly the same for both architectures.
- In the tree architecture, we see the generation of different size trees depends on the number of inputs and is straightforward to implement. The tree needs not be perfect binary, but this restriction clearly helps in the implementation.
- For the sequential datapath architecture, the inputs need not be a power of two because what is affected by changing P is the FSM and multiplexer. Any input value can be easily taken into consideration in the generation of the design because it does not depend on a tree architecture.
- For the sequential datapath architecture, the calculation of the final checksums, given the partial checksums, stays the same regardless of the number of inputs that are implemented and is generated in the same way for every design with different input parameters.

4.2 Evaluation Setup

After using the generator to produce a variety of designs with the parameters specified in Table 3, we synthesized them, first, using an FPGA design flow, and then synthesizing for an ASIC (using the NanGate 45nm Open Cell Library). We obtained metrics for the cost and performance of the implementations.

	Type 1: Tree	Type 2: Sequential
Number of Input Bits	4, 8, 16, 32 bits	4, 8, 16, 32 bits
Number of Parallel Inputs (power of two)	2, 4, 8, 16, 32, 64	2, 4, 8, 16, 32, 64
Pipelining Options	1: Pipelined 2: Not Pipelined	Pipelined

Table 3: Parameters for designs used for synthesis.

The FPGA experiments were performed using Altera Quartus II design software, targeting an Altera Arria II GX FPGA (EP2AGX45DF29C5). After performing synthesis, fitting (including place-and-route) and static timing analysis, we determine for each design: the number of adaptive LUTs (ALUTs) that were used; the number of registers that were used; the maximum clock frequency at which the design could run; and the critical path. ALUTs are logical constructs that represent the combinational resources used. No RAMs or DSPs are used in these designs.

For the ASIC experiments, we used Synopsys DesignCompiler version A-2007.12, and targeted the NanGate 45nm Open Cell Library, based on the NCSU 45nm FreePDK. After performing synthesis using the Synopsys tool, we use its ability to provide estimates for area (μm^2), power (mW), and timing (maximum clock frequency and critical path).

4.3 Synthesis Results

4.3.1 Comparison Between Different Architectures

As detailed in the previous section, we have several parameters that affect the system performance that can be compared and evaluated. In order to evaluate performance we compared the estimated throughput against several metrics. We calculated the throughput (T) for each design, $T = Wf$, where $W = PB$, P is the number of simultaneous parallel inputs, and B is the number bits per word. Each design is characterized through its checksum size M and graphed according to the number of registers or ALUTs determined from Quartus's reports, or the estimated area or power consumption determined by DesignCompiler.

In Figures 8 and 9 we compare throughput reached for four families of tree designs and four families of sequential designs based on checksum size (M) and with varying values of P. Figure 8 shows throughput on the y-axis versus ALUTs (combinational logic) on the x-axis. Figure 9 shows throughput versus registers. The designs are grouped by architecture (tree or sequential, see Chapter 3) and by checksum size (M). As each line goes up and to the right, its value of parallelism is increasing (thus increasing the cost and throughput). We can see that the maximum throughput achieved by the designs (the last marker, when they reach 256 input bits per clock cycle) decreases as M increases in size in both architectures. This means that designs with more inputs and shorter word length obtain higher throughput because higher maximum frequencies are achieved compared to designs with equivalent total input bits (W). We can notice the tree architecture reaches the highest throughputs, but the sequential architecture uses less resources (ALUTs and registers). This is consistent with our theoretical estimation finding of the use of resources per design. Yet, it is notable that, for the same increase in ALUTs, the sequential designs shows significantly more gain in throughput than the tree designs. Also, there's less variation in performance between different values of M for sequential designs than for tree designs. When the sequential design scales to the maximum parameters tested (parallelism $P = 8$, checksum length $M = 64$) we can see that the linear trend is broken. We believe that this is due to unexpected synthesis difficulties with this challenging parallel design.

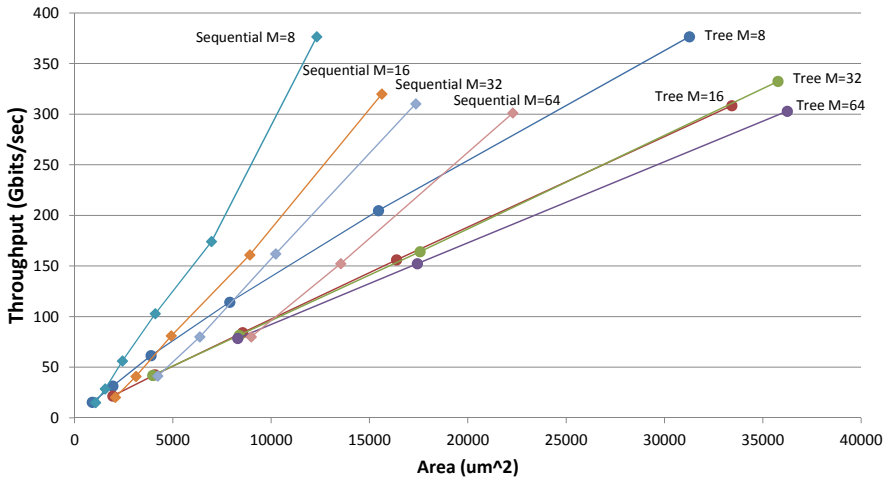


Figure 10: Throughput in Gbits/sec vs estimated implementation area on Nangate 45nm Open Cell Library.

In Figures 10 and 11 we compare throughput versus estimated implementation area and power consumption as calculated with DesignCompiler. We can see that the obtained throughput is about the same for either architecture, varying slightly in favor of the tree designs. This is because the the frequencies obtained where about the same for implementation in both architectures. The estimated design area is consistent with what we saw in the FPGA implementation, but with a larger degree of variation for the sequential architecture as checksum size M increases. We can see similarities between design area and power consumption estimation, which is consistent with what we would expect.

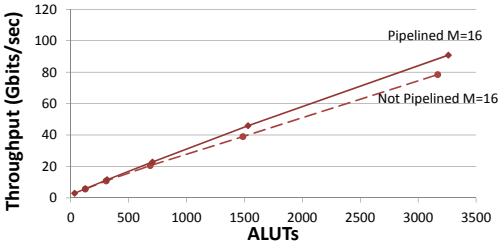
For the tree architecture designs, when we have 2 parallel inputs, the critical path is usually between the two consecutive addition blocks in the checksum stage. This is the only place in the design where, due to the feedback loop, we do not have a pipeline stage between two sums. Then, as the number of inputs increases, and therefore the complexity of the design increases, the critical path tends to migrate to the addition of two branches of the tree structure.

For small sequential architecture designs, when parallelism is 2 or 4, the critical path is also usually the feedback loop that adds the second checksum values. When parallelism increases, we can see that the critical path changes to the path that contains the full multiplier.

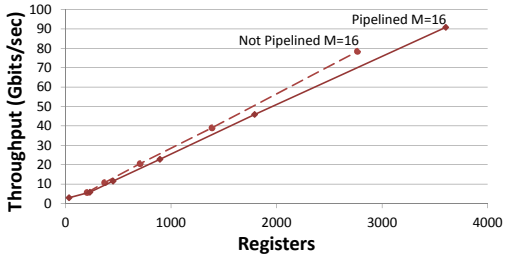
4.3.2 Tree Architecture: Effect of Pipelining

If you recall, from Section 3.1.1, we introduced the notion of adding this design parameter to the tree architecture due to the potential of obtaining substantial increase in clock frequency for the larger designs. After every logical operation, a pipeline stage was added when doing so would not affect the correct functioning of the checksum. After analyzing the results obtained from the synthesis of the pipelined and non pipelined versions of the same design, we determined that, for comparison between architectures, the significant increment in throughput outweighed the costs in logic components and registers. For this reason, the experiments in the previous section used the pipelined version of the tree architecture.

In Figure 12 we can observe the difference in pipelining the tree architecture for $M = 16$. The variance in additional registers needed is more than the variance in increased throughput. In Figure 13 we can recognize a similar trend, results for designs with $M = 16$ and $M = 32$ lie between the represented values. The unpipelined tree architectures achieve a throughput similar to the sequential designs. On the other hand, the pipelined versions provide the highest throughput.

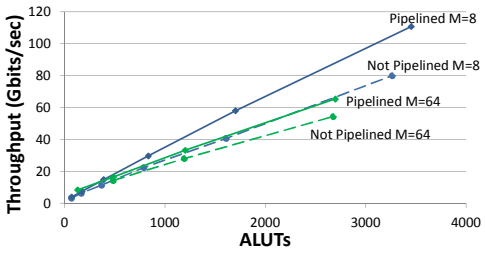


(a) Throughput in Gbits/sec vs ALUTs on FPGA

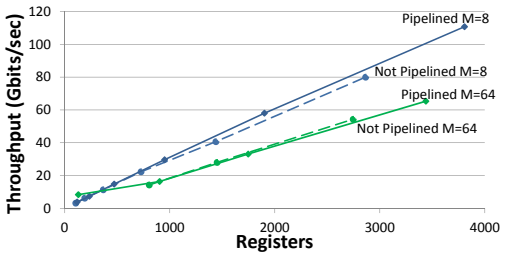


(b) Throughput in Gbits/sec vs registers on FPGA

Figure 12: Throughput vs. ALUTs and registers, with $M = 16$.



(a) Throughput in Gbits/sec vs. ALUTs



(b) Throughput in Gbits/sec vs. registers

Figure 13: Throughput vs. ALUTs and registers, with $M = 8$ and $M = 64$.

Chapter 5 Conclusions

This thesis set out to explore the feasibility and implementation tradeoffs of a multiple simultaneous input design for a Fletcher Checksum. Speed and complexity of the method used to verify the correctness of data transmission is of significant importance in the overall design of the system. We established different methods for parallelizing the Fletcher Checksum and compared the throughput and implementation costs.

We have found that we were able to reconstruct the Fletcher Checksum through combination of the partial checksums corresponding to each simultaneous input. The results have shown that parallelization of this checksum causes an increase in throughput directly proportional with the area necessary for its implementation. When comparing both architectures and maintaining other parameters the same, we notice there are significant differences in choosing a fully pipelined version of the tree and sequential processing datapath. The former achieves greater throughput at a higher cost. On the other hand, the latter on average uses around half of the resources for a given parallelism. We have concluded that if we need to obtain a checksum result for every clock cycle then we should use the tree design in implementation, but if we only need the checksum result upon request, then the sequential design is more efficient. Yet, it is a recurring observation across both architectures, that it is more desirable to handle smaller word lengths and increase parallelism, so that the design can attain higher maximum frequencies, which is a determining factor in achieving faster throughput.

Offloading specific computations to hardware is a way that we can improve data processing in many systems. Future work for this line includes performing tests to determine error-detection capabilities of our designs different input data types. Also, we will consider other algorithms and hardware implementations.

References

[1] W. Peterson and D. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, pp. 228–235, Jan 1961.

[2] "Internet header format," , Internet Protocol DARPA Internet program protocol specification. IETF. STD 5. RFC 791, Sep 1981.

[3] J. Fletcher, "An arithmetic checksum for serial transmissions," *Communications, IEEE Transactions on*, vol. 30, pp. 247–252, January 1982.

[4] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pp. 109–116, ACM, 1988.

[5] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk, "Fingerprinting: bounding soft-error-detection latency and bandwidth," *Micro, IEEE*, vol. 24, pp. 22–29, Nov 2004.

[6] B. Meyer, B. Calhoun, J. Lach, and K. Skadron, "Cost-effective safety and fault localization using distributed temporal redundancy," in *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 125–134, Oct 2011.

[7] D. Feldmeier, "Fast software implementation of error detection codes," *Networking, IEEE/ACM Transactions on*, vol. 3, pp. 640–651, Dec 1995.

[8] T. Maxino and P. Koopman, "The effectiveness of checksums for embedded control networks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, pp. 59–72, Jan 2009.

[9] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRCs over real data," *IEEE/ACM Trans. Netw.*, vol. 6, pp. 529–543, Oct. 1998.

[10] A. Nakassis, "Fletcher's error detection algorithm: How to implement it efficiently and how to avoid the most common pitfalls," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 63–88, Oct. 1988.

[11] K. Sklower, "Improving the efficiency of the OSI checksum calculation," *SIGCOMM Comput. Commun. Rev.*, vol. 19, pp. 32–43, Oct. 1989.

[12] A. R. M. Kamal, C. J. Bleakley, and S. Dobson, "Failure detection in wireless sensor networks: A sequence-based dynamic approach," *ACM Trans. Sen. Netw.*, vol. 10, pp. 35:1–35:29, Jan. 2014.

[13] B. Han, F. Gringoli, and L. Cominardi, "Bologna: Block-based 802.11 transmission recovery," in *Proceedings of the 2010 ACM Workshop on Wireless of the Students, by the Students, for the Students*, S3 '10, pp. 45–48, ACM, 2010.

[14] M. Braun, J. Friedrich, T. Grün, and J. Lembert, "Parallel CRC Computation in FPGAs," in *Workshop on Field Programmable Logic and Applications*, 1996.

[15] G. Campobello, G. Patane, and M. Russo, "Parallel CRC realization," *IEEE Transactions on Computers*, vol. 52, pp. 1312–1319, Oct. 2003.

[16] C. Cheng and K. Parhi, "High-Speed Parallel CRC Implementation Based on Unfolding, Pipelining, and Retiming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 10, pp. 1017–1021, 2006.

[17] W. M. El-Medany, "FPGA Implementation of CRC with Error Correction," in *International Conference on Wireless and Mobile Communications*, pp. 266–271, 2012.

[18] H. F. A. Hamed, F. A. Elmisery, and A. A. H. A. Elkader, "Implementation of Low Area and High Data Throughput CRC Design on FPGA," *International Journal of Advanced Research in Computer Science and Electronics Engineering*, vol. 1, no. 9, pp. 48–54, 2012.

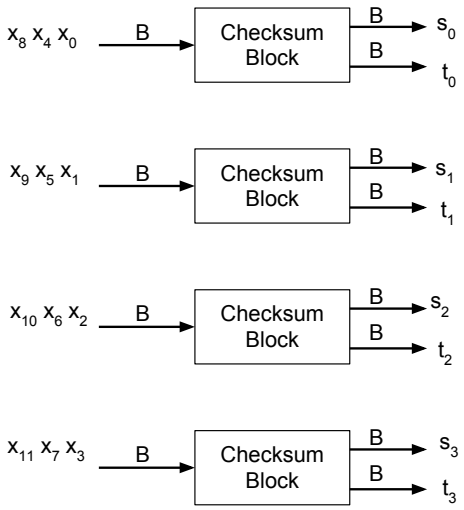
[19] F. Monteiro, A. Dandache, A. M'sir, and B. Lepley, "A Fast CRC Implementation on FPGA Using a Pipelined Architecture for the Polynomial Division," in *International Conference on Electronics, Circuits and Systems*, pp. 1231–1234, 2001.

[20] M. Walma, "Pipelined Cyclic Redundancy Check (CRC) Calculation," in *2007 16th International Conference on Computer Communications and Networks*, pp. 365–370, IEEE, Aug. 2007.

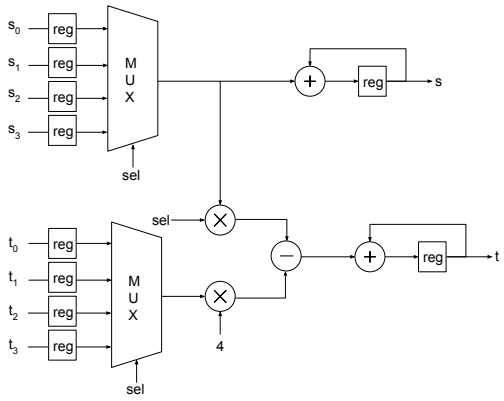
[21] W. Yang and X. Zhou, "CRC circuit design for SRAM-Based FPGA configuration bit correction," *2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology*, pp. 1660–1664, Nov. 2010.

[22] T. Zhang and Q. Ding, "Design and Implementation of CRC Based on FPGA," *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pp. 160–162, Dec. 2011.

[23] J. Caplan, M. I. Mera, P. Milder, and B. H. Meyer, "Trade-offs in execution signature compression for reliable processor systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014.



(a) Four basic checksum blocks are used to calculate partial checksum values.



(b) From the four partial checksum values we calculate the complete checksum.

Figure 7: Four parallel input Fletcher Checksum sequential design

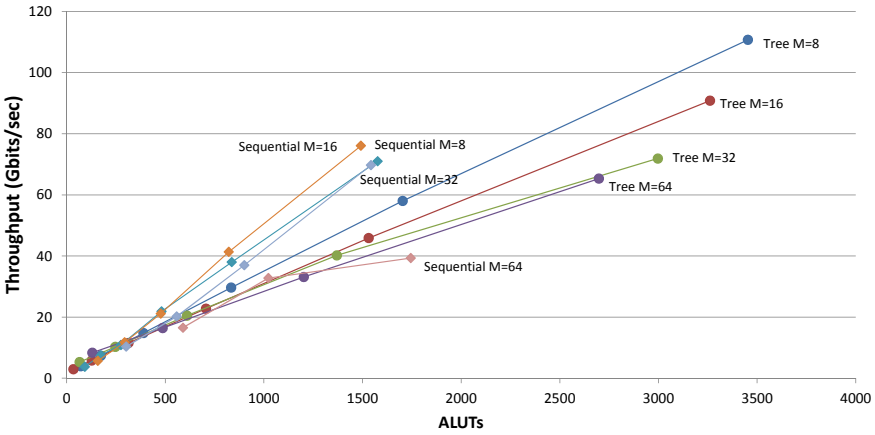


Figure 8: Throughput in Gbits/sec vs ALUTs on FPGA

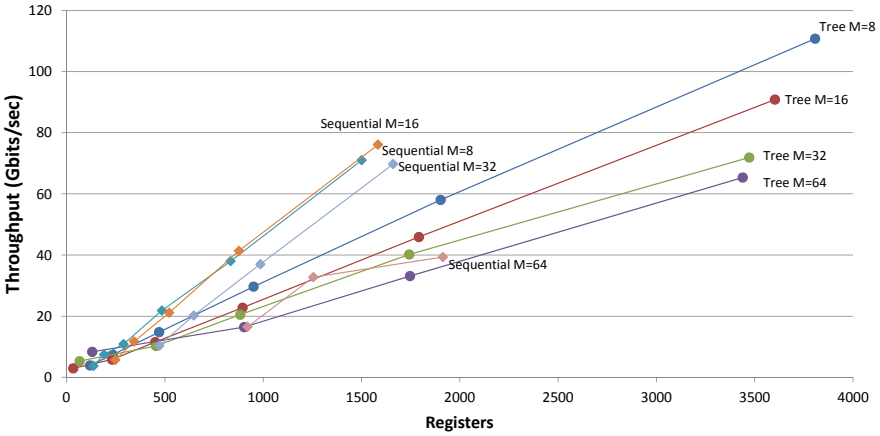


Figure 9: Throughput in Gbits/sec vs registers on FPGA

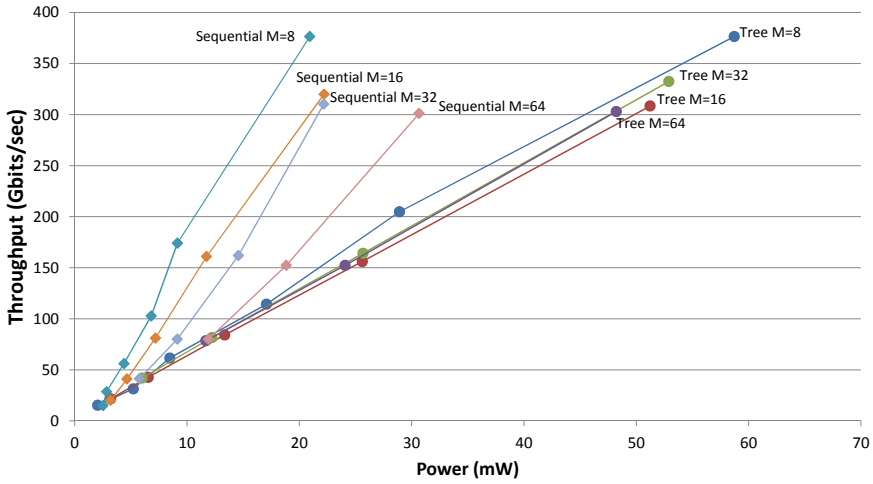


Figure 11: Throughput in Gbits/sec vs estimated power consumption on Nangate 45nm Open Cell Library.