



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

THESIS

**THE CIRCULAR PIPELINE:
ACHIEVING HIGHER THROUGHPUT IN THE SEARCH
FOR BENT FUNCTIONS**

by

Christopher D. Johnson

September 2010

Thesis Co-Advisors:

Jon T. Butler
Pantelimon Stanica

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 2010	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE The Circular Pipeline: Achieving Higher Throughput in the Search for Bent Functions		5. FUNDING NUMBERS	
6. AUTHOR(S) Christopher D. Johnson			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _____ N/A _____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) For the first time, the circular pipeline as a means to significantly improve the throughput achieved in the search for bent functions is presented in this thesis. Linear cryptanalysis attack is a threat to modern symmetric encryption systems. A good defense is the use of a primitive based on Boolean functions having the highest nonlinearity possible—a bent function. Bent functions are extremely rare and, therefore, difficult to find. The implementation of a sieve on a field programmable gate array (FPGA) provides a high throughput (one function per clock) approach to searching for bent functions. With a clock frequency of 100 MHz, throughput is 100,000,000 functions per second. The circular pipeline as a way to achieve an even higher throughput is examined in this thesis. The theoretical maximum speedup is 2^n , where n is the number of variables. The exact achievable speedup has been unknown until now. It is shown that a speedup of 55 is achieved at $n = 6$ with the design proposed in this thesis, which is 86% of the theoretical maximum.			
14. SUBJECT TERMS Circular Pipeline, Boolean Bent Functions, Hardware Complexity, Circuit Complexity, Nonlinearity, Hamming Distance, Cryptography		15. NUMBER OF PAGES 116	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**THE CIRCULAR PIPELINE:
ACHIEVING HIGHER THROUGHPUT IN THE SEARCH FOR BENT
FUNCTIONS**

Christopher D. Johnson
Lieutenant, United States Navy
B.S., University of Michigan, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 2010**

Author: Christopher D. Johnson

Approved by: Jon T. Butler
Thesis Co-Advisor

Pantelimon Stanica
Thesis Co-Advisor

Clark Robertson
Chairman, Department of Electrical & Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

For the first time, the circular pipeline as a means to significantly improve the throughput achieved in the search for bent functions is presented in this thesis. Linear cryptanalysis attack is a threat to modern symmetric encryption systems. A good defense is the use of a primitive based on Boolean functions having the highest nonlinearity possible—a bent function. Bent functions are extremely rare and, therefore, difficult to find. The implementation of a sieve on a field programmable gate array (FPGA) provides a high throughput (one function per clock) approach to searching for bent functions. With a clock frequency of 100 MHz, throughput is 100,000,000 functions per second. The circular pipeline as a way to achieve an even higher throughput is examined in this thesis. The theoretical maximum speedup is 2^n , where n is the number of variables. The exact achievable speedup has been unknown until now. It is shown that a speedup of 55 is achieved at $n = 6$ with the design proposed in this thesis, which is 86% of the theoretical maximum.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	LINEAR CRYPTANALYSIS.....	1
B.	ENUMERATION OF BENT BOOLEAN FUNCTIONS.....	1
C.	SPEEDUP USING A CIRCULAR PIPELINE	1
D.	THESIS GOALS.....	2
E.	THESIS ORGANIZATION.....	3
II.	BENT FUNCTION DISCOVERY USING SIEVE.....	5
A.	FUNCTIONS.....	5
1.	Definitions.....	5
a.	<i>Boolean Functions.....</i>	<i>5</i>
b.	<i>Linear Functions.....</i>	<i>5</i>
c.	<i>Affine functions</i>	<i>5</i>
d.	<i>Nonlinearity (NL_f).....</i>	<i>5</i>
e.	<i>Bent Weight.....</i>	<i>5</i>
f.	<i>Bent Functions.....</i>	<i>6</i>
g.	<i>Throughput (T).....</i>	<i>6</i>
B.	PARALLEL SIEVE ARCHITECTURE	6
1.	XOR Operation.....	7
2.	Ones Count	7
3.	Minimum	8
C.	ADVANTAGES.....	9
D.	DISADVANTAGES.....	9
III.	CIRCULAR PIPELINE SIEVE ARCHITECTURE	11
A.	RESERVOIR.....	11
B.	CIRCULAR PIPELINE	15
1.	Data Flow and Control Logic Complexity Comparison.....	16
C.	FUNCTION GENERATOR	18
1.	With Reservoir	18
2.	Without Reservoir.....	19
D.	PERSISTENCE.....	20
1.	Worst-Case Scenarios.....	24
a.	<i>With Reservoir.....</i>	<i>24</i>
b.	<i>Without Reservoir</i>	<i>24</i>
E.	SUMMARY	25
IV.	IMPLEMENTATION	27
A.	VERILOG IMPLEMENTATION	27
1.	Reservoir.....	27
a.	<i>Priority Encoders</i>	<i>27</i>
b.	<i>Adders.....</i>	<i>28</i>
c.	<i>Registers</i>	<i>29</i>

2.	Circular Pipeline	29
B.	VERILOG DESIGN DEVELOPMENT AND TESTING.....	29
C.	SRC-6 IMPLEMENTATION	31
1.	Macro Characteristics	32
2.	Streaming Output	32
3.	CPU	33
4.	Subroutine and Macro Call	33
5.	Timing	33
6.	FPGA Resources	33
D.	SUMMARY	34
V.	RESULTS	35
A.	SPEEDUP	35
B.	RESOURCES	38
C.	RESERVOIR TRADEOFF.....	39
D.	SUMMARY	41
VI.	CONCLUSIONS AND RECOMMENDATIONS.....	43
A.	CONCLUSION	43
B.	RECOMMENDATIONS FOR FURTHER RESEARCH	43
1.	Multiple Output Stages.....	43
2.	Pipelined Reservoir.....	44
3.	Multiple FPGAs	44
4.	Function Generators.....	44
APPENDIX.	PROGRAMMING CODE.....	45
A.	VERILOG.....	45
1.	Circular Pipeline With Independent Function Generators	45
2.	Circular Pipeline With Reservoir.....	65
B.	SRC-6 IMPLEMENTATION FILES	84
1.	main.c	84
2.	subr.mc.....	86
3.	makefile.....	87
4.	info.....	90
5.	blk.v.....	91
LIST OF REFERENCES		93
INITIAL DISTRIBUTION LIST		95

LIST OF FIGURES

Figure 1.	Sieve Architecture for Bent Function Discovery. From [5]	7
Figure 2.	Bitwise XOR Architecture. From [5].....	7
Figure 3.	Ones Count Architecture. From [5]	8
Figure 4.	Minimum Module's Architecture. From [5].....	8
Figure 5.	Reservoir Architecture.....	12
Figure 6.	Linear Pipeline Information Flow.....	16
Figure 7.	Circular Pipeline Information Flow.....	17
Figure 8.	Circular Pipeline Data with One Stage Output.....	17
Figure 9.	Synplify Pro RTL View of a Circular Pipeline Stage. $n = 4$	30
Figure 10.	Synplify Pro RTL View of the Bent Weight Tester Within a Stage. $n = 4$	30
Figure 11.	Synplify Pro RTL View of a One's Counter Within a Bent Weight Tester. $n = 4$	30
Figure 12.	ModelSim Post-map Simulation Result Excerpt.	31
Figure 13.	Realized Throughput.....	37
Figure 14.	Throughput Normalized to 2^n	38
Figure 15.	Relative Completion Times of the IFG.....	41

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Reservoir Complexity.	15
Table 2.	Throughput and Average Persistence. From [4]	21
Table 3.	Example Computation of Throughput for $n = 4$. From [4]	22
Table 4.	Realized Speedup.	36
Table 5.	Resources Consumed Summary.	39
Table 6.	Circular Pipeline With and Without Reservoir (Res) Comparison.	40

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

DES	Data Encryption Standard
FPGA	Field Programmable Gate Array
FUT	Function Under Test
IFG	Independent Function Generators
LUT	Lookup Table
MUX	Multiplexor
OBM	On Board Memory
RTL	Register Transfer Level

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Computer hardware architecture that speeds up the process of sieving through a pool of functions in search of a set of characteristics is presented in this thesis. This architecture—the circular pipeline—is motivated by the search for the most nonlinear functions, known as bent functions, due to their usefulness in cryptographic applications. Bent functions provide for a defense against linear cryptanalysis attack. A linear attack attempts to break the cipher key using a series of linear approximations for the key. If successful, linear characteristics of the cipher key are exploited and the encryption is broken. Bent functions are the least linear of all functions, making them most resistant to linear cryptanalysis attack.

No analytic method is known to solve for bent functions, so large pools of candidate functions must be tested in order to find bent functions. Bent functions are well defined and testing is straightforward. However, the pools of candidate functions are so large that modern processing power is insufficient to exhaustively sieve through all possibilities. Utilizing the parallelism afforded by reconfigurable computing on the SRC-6, we achieved a speedup of over 60,000 times over a conventional processor at the Naval Postgraduate School. The speedup achieved through parallel processing is improved through more efficient use of the parallel stages in the circular pipeline design.

The conventional parallel design tests a single function per clock period. To discover a bent function, it must be tested against all linear functions; therefore, the conventional design contains tests for all linear functions in parallel. Each test consists of calculating the nonlinearity of the function under test and determining if it is a bent weight. A bent weight is easily defined, and this part of the test is completed with two comparators, one for each of the two bent weights. The nonlinearity is calculated with a bitwise exclusive-OR followed by a tree of adders that sum the resulting number of ones.

The circular pipeline uses the same test modules used in the conventional design, but controls the flow of functions through the stages differently. Rather than applying a single function to all stages simultaneously for testing, a distinct function is applied to

each test module, which is a stage of the circular pipeline. If a bent weight is found, the function is advanced to the following stage, where another test is applied. If a bent weight is not found, the function is discarded and the following stage accepts a new function from the function generator. A function is continually passed to a subsequent stage as long as it passes tests. If a function passes all tests, it is bent. As soon as a function fails a single test, it is ejected, making room for a new function to be inserted to the pipeline and tested. The result is more efficient use of the stages compared to the conventional design that performs simultaneous tests.

Exactly what speedup is achievable is related directly to how much more efficiently the stages are utilized. This efficiency, in turn, is directly related to how many stages functions tend to pass before failing (and being ejected from the pipeline). Due to the rarity of bent functions, a function selected at random is more likely to fail an individual stage test than to pass. Therefore, a great deal of efficiency, realized as throughput and ultimately speedup in total computation time, is gained with circular pipeline architecture.

The circular pipeline requires additional logic to control the additional complexity of information flow through the stages. Conventional speedup gained through parallelism is done so at a cost of doubling logic resources to double throughput. Therefore, the circular pipeline must have a better speedup to increased-logic ratio to be a technological improvement.

Two primary design variations were developed and tested. The first uses a reservoir queuing system to equitably distribute functions from a single function generator to all stages. This design resulted in the greatest speedup, but logic resource consumption was too great to make it practical and could only be realized for very simple cases. The second design implemented independent function generators, one for each stage, in order to eliminate the reservoir and providing an economical speedup. A contribution of this thesis is to demonstrate a speedup to logic-resources-demand ratio of 55:2.3. Conventional parallelism yields a ratio of 1:1. Furthermore, the trend of this ratio improves as complexity (the number of variables) of the circular pipeline increases.

ACKNOWLEDGMENTS

I offer endless appreciation for having been born into this country, which has afforded the endless opportunities that have brought me here. I bear in mind those hardworking individuals around the globe who are not provided the same opportunities.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. LINEAR CRYPTANALYSIS

Matsui [1] introduced the linear cryptanalysis method that succeeded in breaking the Data Encryption Standard (DES) block cipher. DES was endorsed by the United States Bureau of Standards in 1976 and was ubiquitous in data encryption applications into the 2000s. Matsui's linear cryptanalysis method uses a series of linear approximations to decipher the target message. The use of a highly nonlinear Boolean function in the encryption process is an effective defense against such a linear cryptanalysis attack. Bent functions are highly nonlinear, and therefore useful in securely encrypting data.

B. ENUMERATION OF BENT BOOLEAN FUNCTIONS

While the precise definition of a bent function is straightforward, generating a bent function is not. Currently, our approach to enumerating all n -variable bent functions is to exhaustively test a large pool of candidate n -variable functions using a sieve technique. It has been demonstrated that a reconfigurable computer is an efficient way to test functions for bentness [2]. Until now, the architecture implemented on the SRC-6 at the Naval Postgraduate School tests a single function in truth table form simultaneously against all affine functions (or a subset thereof determined to be adequate). The parallelism afforded by the reconfigurable computer to perform simultaneous tests provides a speedup factor of greater than 60,000 over a conventional processor [2].

C. SPEEDUP USING A CIRCULAR PIPELINE

An inherent inefficiency with the current architecture is that a majority of the simultaneously performed tests reconfirm the same conclusion—that the function under test (FUT) is not bent. This is a result of the rare nature of bent functions. Each of the parallel tests is performed with a distance calculator that finds the distance between an affine function and the FUT. All tests must be applied and passed to declare that a

function is bent. That is, only one test needs to fail to determine a function is not bent. In the majority of cases, a function fails many tests. We seek a method in which a function is subject to individual tests sequentially and is immediately ejected when it fails one test. In this way, the test units are more efficiently used and the throughput is greater. FUTs that pass are forwarded to subsequent distance calculator stages until they either fail their first test or pass all tests. In this way, the information obtained from every test conducted is an essential operation. No resources are wasted performing unnecessary tests [4].

With the circular pipeline architecture, the maximum throughput possible is the number of stages S . This is achieved when all functions fail. The average will be less. This compares to a fixed throughput of 1 function per cycle with the conventional sieve architecture [4].

Although the number of distance calculators (each belonging to a stage in the circular pipeline) remain constant, an increase in the pipeline's control unit logic is expected to be required for a circular architecture. This is due to the increase of possible routes for data to flow into and out of each pipeline stage. Each stage of the conventional architecture always accepts a new function from the function generator and always passes its result along. A circular pipeline stage may or may not accept a new function from the function generator, may or may not accept a function from the preceding stage, and may or may not pass a function it tests to the subsequent stage for further testing.

Discovering the exact tradeoff between speedup and additional logic resource requirements of the circular pipeline architecture is a key area of interest.

D. THESIS GOALS

This thesis investigates the amount of speedup realizable with circular pipeline architecture implemented on the SRC-6. Insight into this will guide further advances in bent function discovery using the sieve technique along with possibly providing useful data for high-speed calculation of other mathematical operations amenable to circular pipeline architecture.

E. THESIS ORGANIZATION

A basic overview of this thesis is presented in Chapter I. Background information is presented in Chapter II. The design proposed by this thesis to attain calculation speedup is detailed in Chapter III. Implementation issues are addressed in Chapter IV. Results and analysis are presented in Chapter V. The thesis summary and suggestions for future research in this area, specifically potential improvements to the proposed circular pipeline architecture, are presented in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BENT FUNCTION DISCOVERY USING SIEVE

A. FUNCTIONS

1. Definitions

a. *Boolean Functions*

A Boolean function f on n variables is a map from the n -dimensional vector space $V_n = F^2$ to F_2 , the two-element field. For a function f , let $f_0 = f(0,0,\dots,0)$, $f_1 = f(0,0,\dots,1)$, ..., and $f_{2^n-1} = f(1,1,\dots,1)$. $TT = (f_0 f_1 \dots f_{2^n-1})$ is the truth table representation of f [2].

b. *Linear Functions*

A linear function is the constant zero function or the exclusive-OR (XOR) of one or more variables [2]. There are 2^n linear functions.

c. *Affine functions*

An affine function is a linear function or the complement of a linear function [2]. There are 2^{n+1} affine functions.

d. *Nonlinearity (NL_f)*

The nonlinearity NL_f of a function f is the minimum Hamming distance between f and an affine function, where the Hamming distance between two functions is the number of places where their truth table representations differ [2].

e. *Bent Weight*

A bent weight is defined to be a nonlinearity of $2^{n-1} \pm 2^{\frac{n}{2}-1}$ [1]. If a function is found to have a bent weight for a linear function, it will have also have a bent weight

for that linear function's complement. Therefore, it is sufficient to test only against all linear functions [2].

f. Bent Functions

A bent function has a maximum nonlinearity among n -variable functions, where n is even. A bent function will have bent weights for all 2^n linear functions (and implicitly, all 2^{n+1} affine functions) [2].

It follows that a small portion of the 2^{2^n} functions of an n -variable function are bent. For $n = 4$, $\frac{896}{65,536} = 1.3\%$ of the 4-variable functions are bent. This percentage decreases as n increases. For example, $n = 6$ has a bent function ratio of $5,425,430,528/2^{2^6} = 2.94 \times 10^{-8}\%$ [3].

g. Throughput (T)

Throughput T is the rate at which functions are processed, given in units of functions per clock.

B. PARALLEL SIEVE ARCHITECTURE

An approach to discover all bent functions for n -variable functions is to enumerate all possible truth tables sequentially and apply each to all affine functions simultaneously. As depicted in Figure 1, the FUT is bitwise XOR'd with each affine function, then 'Ones Count' logic determines the number of resulting ones (the Hamming distance), followed by a 'Minimum' circuit that finds the lowest value for all the 'Ones Count' inputs. The output of 'Minimum' is the nonlinearity of the function. Together, these modules are distance calculators, providing the distance between two inputs—an affine function and a FUT. This process is pipelined to achieve a clock rate of 100MHz with throughput of one function per clock on the SRC-6. Each module of the distance calculator will now be discussed in further detail.

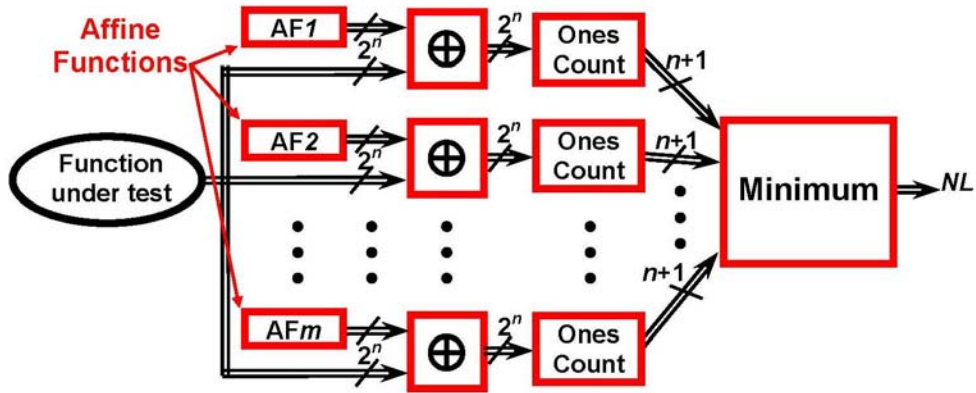


Figure 1. Sieve Architecture for Bent Function Discovery. From [5]

1. XOR Operation

The bitwise XOR operation of bus width 2^n is constructed of $2^n/2$ parallel 2-input XOR gate. This is depicted in Figure 2.

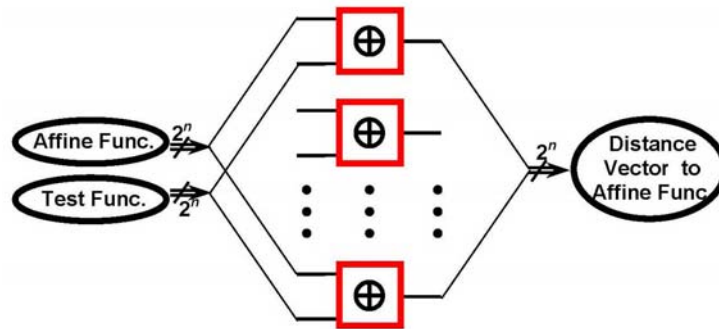


Figure 2. Bitwise XOR Architecture. From [5]

2. Ones Count

The Ones Count circuit is constructed as a tree beginning with $\frac{2^n}{4}$ 4-input adders and ending with a $2n$ -wide adder with an $n+1$ -wide output that is the Hamming distance to the affine function. This design is illustrated by Figure 3.

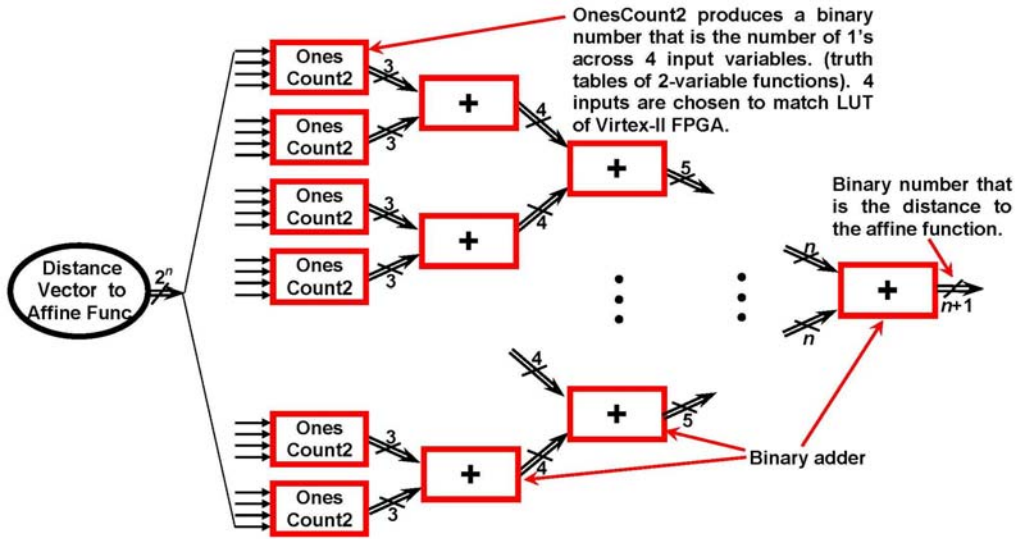


Figure 3. Ones Count Architecture. From [5]

3. Minimum

The minimum circuitry is also constructed as a tree, with each building block receiving two $n+1$ -wide inputs (the results from the Ones Counts modules) and producing the $n+1$ -wide nonlinearity in binary. This architecture is depicted in Figure 4.

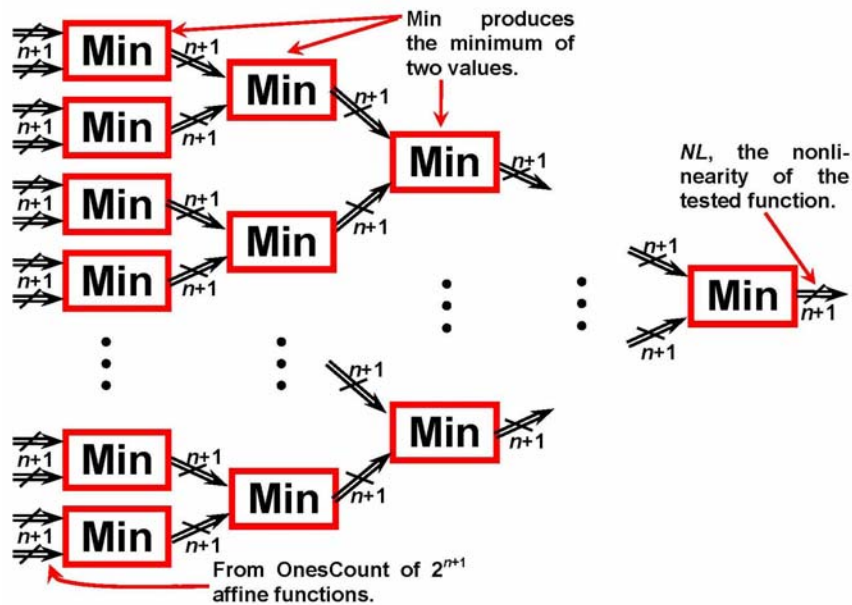


Figure 4. Minimum Module's Architecture. From [5]

C. ADVANTAGES

The principle advantage of this architecture is that a large number of operations are performed in parallel that would otherwise have to be executed in serial on a conventional CPU. For example, a bitwise XOR operation is required for each affine function, which amounts to a total of 2^{n+1} operations, or more if the conventional processor cannot accommodate a 2^n -wide bitwise XOR. The ability to execute all of these operations in parallel amounts to a significant time savings over conventional processors for large n -variable functions [5].

D. DISADVANTAGES

The principle disadvantage of this parallel sieve technique is that, for any one cycle, the distance calculators provide redundant information about each non-bent FUT, which typically fail many of the parallel tests.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CIRCULAR PIPELINE SIEVE ARCHITECTURE

An improvement in computational time to discover all bent functions for a given n is sought by achieving greater utilization of the distance calculators. The sieve consists of 2^n stages that each computes the distance between f and one of the 2^n linear functions. Then, it determines if its distance is a bent weight $2^{n-1} \pm 2^{n/2-1}$.

Persistence (P_i)

Persistence is the number of stages a function f_i is subjected to before removal from the circular pipeline. P_i is equal to the number of passed tests for bentness (one per stage) plus one (for the stage that removes f). P is the average persistence over all functions.

If a function f_i is found to have a bent weight, its persistence P_i is incremented and it is passed to the next stage. If f is found not to have a bent weight, it is ejected from the circular pipeline and the following stage accepts a new function. In the case that f_i is bent, P_i will grow to 2^n . Then, f_i is removed from pipeline and stored [4].

The speedup of the circular pipeline depends on the throughput, which will be $1 \leq T \leq 2^n$. The lower bound occurs if all functions in the pipeline are bent, while the upper bound occurs when none of the functions in the pipeline have a bent weight and are therefore ejected after one cycle [4].

A. RESERVOIR

For each cycle, 2^n functions must be made available to the circular pipeline in case all previously tested functions were ejected. The sieve procedure begins with a single function generator very similar to that used in the conventional design providing these sets of functions. However, not all of these 2^n functions will be accepted by the circular pipeline because some functions in the circular pipeline will persist, blocking a new input. To achieve exhaustive testing, a reservoir for these unaccepted functions must

be provided so they may be inserted into the pipeline at a later time. Further, a mechanism to provide the functions stored in the reservoir to the circular pipeline, vice a new set from the function generator, must be incorporated.

The reservoir is shown in Figure 5. Functions enter through a multiplexor (MUX) that is sourced with two complete sets of 2^n functions one from the function generator and the other from the reservoir. If a stage in the circular pipeline is available, a function f_i provided by the MUX is inserted. If not, the f_i is routed to the lowest available of the $2^{n+1} - 1$ registers, beginning with L_0 .

Figure 5 is an illustration of the reservoir for $n = 2$. The circular shape at the top of Figure 5 is the circular pipeline with the 4 stages for $n = 2$. L_0 through Q_2 are the $2^{n+1} - 1$ registers required to ensure registers are available for rejected functions in the worst-case scenario. The blocks labeled I are the 2^n functions applied by the MUX.

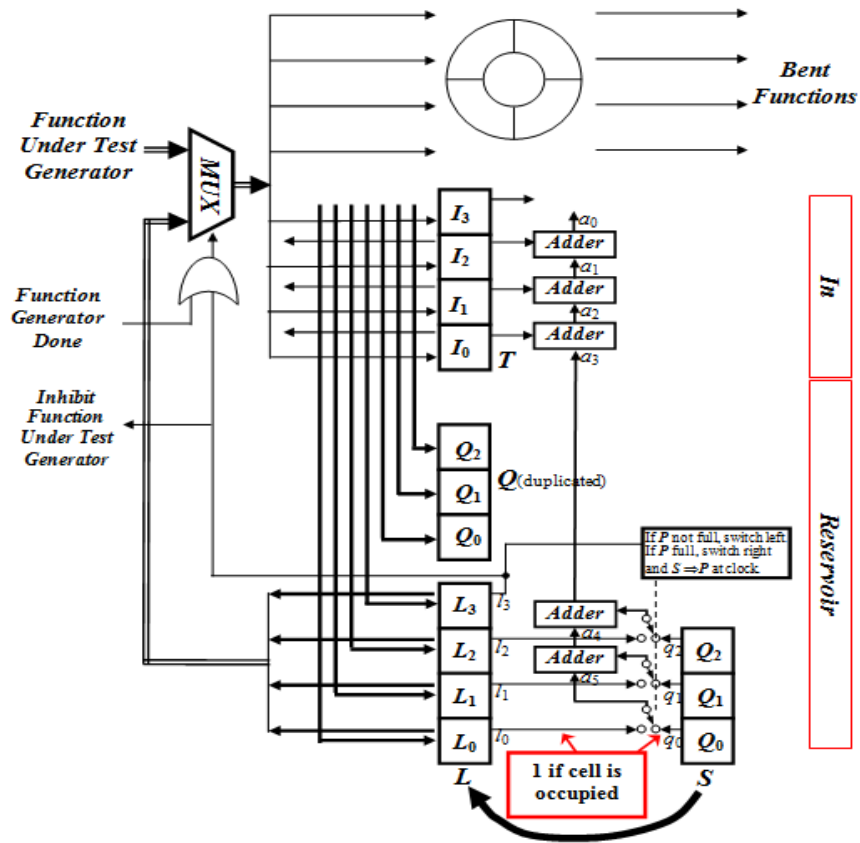


Figure 5. Reservoir Architecture.

The purpose of the reservoir is to store functions rejected by the circular pipeline, so they can be reinserted later. These temporarily stored functions must be queued such that they can be presented to the circular pipeline as a complete set of 2^n functions. A major problem associated with queuing the functions to form a complete set is assuring that no empty registers exist between occupied registers.

The top registers Q_0 through $2^n - 1$ are replicated for the purpose of illustration. It must be known how many empty registers reside below each incoming function I_i (provided by the MUX). Summing the number of L occupied registers with an adder chain is required when the L registers are not all filled. The addition operation needed to sum all occupied L registers is special in that if a stage is found to be occupied, all stages below it are occupied as well. Therefore, a thermometer-type adder, or thermo adder, is used to provide this sum.

Analysis of all possible cases revealed that when the L registers are completely occupied, the same thermo adder simply needs to be applied to the Q registers. This is because the Q registers will slide down to fill the P registers from the bottom up and the incoming functions I will fill in atop these.

The sum produced by the thermo adder is the input to a chain of adders associated with the incoming I functions. A 2^n -bit signal *inToPipe*, from the circular pipeline, is used in the same fashion as the occupied bits are used with the registers. An asserted *inToPipe_i* indicates that the pipeline stage Q_i requires I_i on the next clock; hence, I_i will not be stored in the reservoir. If *inToPipe_i* is low, I_i will be routed into the reservoir. The adder chain accounts for the presence of I_i in the reservoir, which is needed to determine proper routing of other incoming I functions above I_i .

The lowest index I function rejected by the circular pipeline is routed to the lowest indexed available register. The next lowest indexed I function rejected from the circular pipeline is stored in the register directly above where the lowest indexed I function is stored. With this behavior, for each function I to be routed correctly, the number of occupied registers below is needed, to include any other lower indexed I

functions that are being routed to the reservoir on the same clock. The adder chain, applied to the occupied bits of the registers and the *inToPipe* bits of I , provides this number and allows for proper routing.

When the top L register, L_{2^n-1} is filled, a select signal is asserted and the MUX applies the set of $2^n L$ functions from the reservoir. Functions in Q registers slide down to the similarly indexed L register, ensuring the reservoir is filled from the bottom up. When the MUX selects functions from the reservoir, the function generator must be inhibited, which is controlled by the same line used as input to the OR gate that feeds the MUX select. When the function generator has completed generating all functions, a done signal is sent to the reservoir. This signal also feeds the OR gate leading to the MUX select, which routes any remaining functions in the reservoir to the circular pipeline.

Despite being auxiliary, the reservoir is the most complex part of the circular pipeline. An estimate the growth rate of reservoir complexity as a function of n is given in Table 1. The number of connection paths and individual wires required (connections multiplied by bus width) by the reservoir to accompany the circular pipeline for given n are listed in Table 1. The minimum number of transfer paths occurs for I_0 , which has 2^n possible paths. There is no case for which I_0 will be routed to any of the Q registers. I_1 can be routed to any L register or Q_0 . I_2 could be routed to any L register, Q_0 or Q_1 . This pattern continues until reaching I_{n^2} , which could be transferred to any of the registers. This gives a maximum number of transfer paths of $2^{n+1} - 1$.

The total number of transfer paths is given by $\sum_{Min}^{Max} TransferPaths + 2n - 1$. The $2n - 1$ term accounts for the paths for each Q_i register to transfer to its corresponding L_i register. The total number of wires required is found by multiplying the total transfer paths by bus width of f , which is $2n$. Lastly, the growth rate column shows the growth factor of the total number of required wires with respect to the previous row. Bearing in mind that this table omits odd n , we deduce that the complexity of the reservoir grows by approximately $8n$. The circular pipeline is expected to grow at a rate of approximately $2n$, which is the growth rate of the number of stages. This indicates the reservoir

complexity will likely be a limiting factor as n increases and motivated an alternate approach that allows removal of the reservoir. This is discussed in Section C.2.

Table 1. Reservoir Complexity.

n	Stages	Max Transfer Paths	Minimum Transfer Paths	Total Transfer Paths	Bus Width	Total Wires	Growth Rate
2	4	7	4	25	4	100	-
4	16	31	16	391	16	6256	63
6	64	127	64	6175	64	395200	63
8	256	511	256	98431	256	25198336	64
10	1024	2047	1024	1573375	1024	16111136000	64

B. CIRCULAR PIPELINE

Each stage of the circular pipeline is similar to the parallel nonlinearity computers of the conventional sieve architecture. However, additional logic is required to handle the additional complexity of data flow. For each stage, a control unit must determine if a function should be advanced to the next stage or ejected; additionally, whether or not a function is incoming from the preceding stage or a new incoming function should be accepted.

To accomplish this, a 1-bit signal $inToPipe_i$ indicates if the stage Q_i is accepting the incoming function I_i from the MUX. If not, I_i is stored in the reservoir. The 2^n -bit $intToPipe$ vector is used by the reservoir queuing unit to properly route functions to registers in the reservoir.

An n -bit persistence P token accompanies each function throughout its procession in the circular pipeline. A test must be performed to detect when $P \geq 2^n$, at which time the FUT is determined to be bent, removed from the pipeline, and stored.

1. Data Flow and Control Logic Complexity Comparison

The additional complexity required (which translates directly to logic (LUTs on the SRC-6) required for design realization) is best understood by comparing data flow through a traditional linear pipeline to the flow through a circular pipeline. Figure 6 is a graphical depiction of the basic flow of information through a linear pipeline. For bent function searches, this 4-stage pipeline applies to $n = 2$ and each stage is testing f against a distinct linear function for a bent weight. If the function passes through all stages, never failing a test, it is declared bent. Each stage has one input and one output and completes its calculation in one clock. The architecture to control information flow is simple, and throughput is fixed to one function per clock.

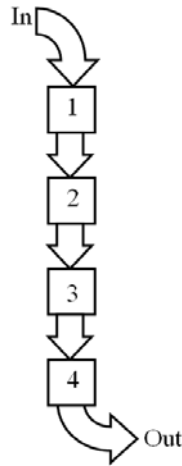


Figure 6. Linear Pipeline Information Flow.

Figure 7 is a depiction of the flow of information through a circular pipeline. Figure 7a is the initial adaptation of the linear architecture and Figure 7b is a modified version of 7a with the output of stage four wrapped around to be the input of stage one. From this illustration, it is immediately clear that greater complexity is required to control the flow of functions through the pipeline. Each stage now has a choice between two inputs and two outputs, which requires controlling logic. An increase in throughput T is the expected payoff.

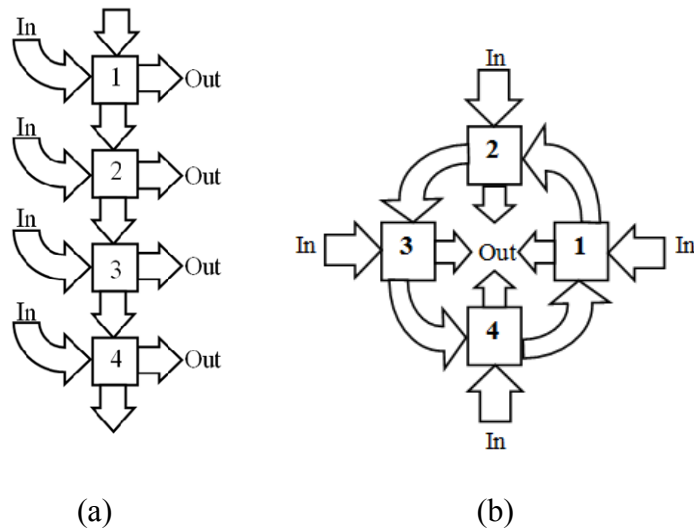


Figure 7. Circular Pipeline Information Flow.

The design for optimal T by enabling every stage to output a result is depicted in Figure 7. With the application we are applying to the circular pipeline, we choose to simplify the design by allowing only one stage to output functions that are determined to be bent, as illustrated in Figure 8.

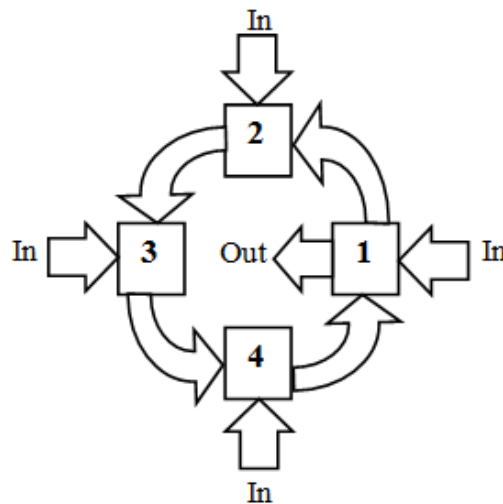


Figure 8. Circular Pipeline Data with One Stage Output.

This simplifies the output interface by disallowing the case that 2^n functions are found to be bent and sent as output on the same clock. If such a case were allowed, as in Figure 7, the output bus would have to be 2^{2^n} bits wide in order to simultaneously transfer 2^n words of 2^n bits each. The SRC-6 can support at least 16 output streams of 320 bits each [6]. Therefore, there is no restriction on output stages through at least $n = 4$. Nonetheless, the simpler design of a single output stage comes with the associated benefits of simpler logic. With the simplification, illustrated by Figure 8, the output bus is 2^n bits wide and the instances of logic required to check the value of P is reduced from 2^n to 1. With this design, every stage has two inputs from which to choose and only one output (to the following stage), save for the one special stage that has an additional output for functions determined to be bent. Additional ideas regarding this issue are presented in Chapter VI Further Research.

C. FUNCTION GENERATOR

1. With Reservoir

The circular pipeline with reservoir architecture requires a function generator that provides 2^n functions on each clock and can be inhibited. This is an extension of the simple counter in the conventional architecture that provided one function and always incremented on each clock. In the conventional architecture, a simple counter used as the function generator was produced with C-style statements implemented on the field programmable gate array (FPGA). This is discussed in greater detail in the sections on Verilog and SRC-6 implementation.

The function generator is also a simple counter when the circular pipeline is used with a reservoir. On each clock, the function generator produces 2^n functions, one for each stage of the pipeline. The most significant n bits of each function f_i are hardwired to i (in binary). A $2^{2^n - n}$ bit counter is concatenated onto the least significant bits. In this way, 2^n distinct truth tables of functions, each 2^n bits long, are formed by the function generator on each clock. The counter is inhibited on any clock that the reservoir's L registers are completely filled because in this case the reservoir provides the functions.

The counter holds its value until the next clock for which the L registers are not completely filled (most likely the very next clock), then resumes incrementing.

A done signal accompanies the FPGA-based function generator. After all possible functions have been cycled through, a done bit signals function generator completion. This signal also asserts the select bit on the input MUX, causing any functions in the reservoir to be routed for insertion to the pipeline. Additionally, the counter done signal initiates termination counter.

The final countdown is $5 \times 2^n - 1$ clocks. This number of clocks is the worst-case for how long it could take to flush the reservoir and circular pipeline. It occurs when all functions in the circular pipeline (i.e. when the function generator signals done and the reservoir is full with $2^{n+1} - 1$ bent functions). If this were to happen, it would take 2^n clocks before the pipeline would accept any functions from the reservoir. After these 2^n clocks, one function per clock would be inserted to the pipeline, and each would persist 2^n clocks. The last function from the reservoir is inserted after $2^{n+2} - 1$ clocks and is determined bent after 2^n clocks, for a total of $5 \times 2^n - 1$ clocks. When this number of clocks is reached, following the function generator signaling completion, the exhaustive test is declared complete and a done signal is asserted.

Using a final countdown rather than testing for and generating signals to indicate the absence of FUTs in the pipeline is a tradeoff between circuit complexity and speed. The final countdown requires the test to continue running for the entire duration of the worst-case scenario, which is unlikely. Additional logic could terminate the test as soon as all functions are removed from the pipeline saving many of the $5 \times 2^n - 1$ clocks. But, this is a very small percentage of the total number of clocks required for the test. Simplifying the circuit and adding a small number of clocks to the test operation was the favored choice.

2. Without Reservoir

Due to the complexity of the reservoir, an alternative design was constructed. In this design, individual function generators exist for each stage. The single function generator used in the conventional and circular pipeline with reservoir architectures is

replaced by an array of 2^n independent function generators (IFGs). Both designs continuously produce 2^n truth tables of functions. Each IFG_{*i*} has its n uppermost bits hardwired to its index (in binary), which range from 0 to 2^n . The remaining lower order bits of each IFG are an independent simple counter. The counter is inhibited any time its associated stage receives a function passed from the preceding stage. If a FUT in a preceding stage fails, no function is passed, a function from IFG_{*i*} is inserted into its corresponding stage S_i , and then IFG_{*i*} is incremented.

A disadvantage with this approach is the inefficiency resulting when IFGs complete their cycle and then remain idle until the last IFG completes. Any S_i is underutilized from the time IFG_{*i*} completes until the last IFG completes. This is because there is no function available for insertion when the S_i is open; S_i continues only to test functions passed from the preceding stage.

The circular pipeline with reservoir does not have this inefficiency because functions are redistributed equitably to all stages until no functions remain. It was postulated that the delta between IFGs' completion times would not be significant, especially as n increases. Due to the nature of bent functions, all stages are expected to have an equal probability of passing or rejecting a function selected at random.

In this configuration, each IFG signals completion and its input to the stage is invalidated. All 2^n function generator's done signals are AND'd with the 2^n *inToPipe* signals, one from each stage. Each asserted *inToPipe_{*i*}* signal indicates the FUT in stage_{*i-1*} was found not to have a bent weight. The output of this 2^{n+1} -input AND function is thereby asserted when all function generators have completed and there is no function remaining in the circular pipeline with a bent weight. This signals completion of the exhaustive test.

D. PERSISTENCE

Throughput is directly related to the average persistence, with the upper bound of 2^n if all functions were to persist for only one clock period, and a lower bound of 1 if all functions persist the 2^n cycles required to determine a function is bent (theoretically, throughput could be a small fraction less than 1, which is explained below).

A function persists in the circular pipeline as long as the bitwise XOR with each linear function returns a bent weight of $2^{n-1} \pm 2^{n/2-1}$. The exact persistence of each function will depend on where in the circular pipeline it is inserted and the order with which the linear functions are placed amongst the stages. Having no insight into advantages with any particular ordering of linear functions within stages, we give no attention to this issue. We expect that the average persistence will depend on the percentage of bent weights contained within all possible functions. A development of this fraction of bent weights is provided in [4]:

For each value of n , there are 2^{2^n} n -variable functions, each of which has a distance value to 2^n linear functions for a total of 2^{2^n+n} instances of a weight. There are 2^n linear functions, each of which is a distance $2^{n-1} \pm 2^{n/2-1}$ from $\binom{2^n}{2^{n-1} \pm 2^{n/2-1}}$ other functions, for a total of $2^n \binom{2^n}{2^{n-1} \pm 2^{n/2-1}}$ instances of a weight of $2^{n-1} \pm 2^{n/2-1}$. Thus, the fraction of instances of weight that are $2^{n-1} \pm 2^{n/2-1}$ is

$$A_n = \frac{2^n \binom{2^n}{2^{n-1} \pm 2^{n/2-1}}}{2^{2^n+n}} = \frac{\binom{2^n}{2^{n-1} \pm 2^{n/2-1}}}{2^{2^n}}. \quad (1)$$

The results of the algorithm for even n , $2 \leq n \leq 8$, are included in Table 2. B_n and N_n are the expected number of bent and non-bent weights for the given A_n . The sum of B_n and N_n is 2^n . In practice, we cannot have fractional values. So, for this development of an estimation of throughput and average persistence, we round B_n and N_n to the nearest integer, notated as $[B_n]$ and $[N_n]$.

Table 2. Throughput and Average Persistence. From [4]

n	A_n	Expected B_n	Expected N_n	2^n	$[B_n]$	$[N_n]$	Calc.. P_{avg}	Calc. T_n	T_n Upper	Exp. P_{avg}	E x
2	0.500	2.0	2.0	4	2	2	1.40	2.86	4	2.50	1
4	0.244	3.9	12.1	16	4	12	1.31	12.2	16	1.65	9
6	0.121	7.8	56.2	64	8	56	1.14	56.1	64	-	-
8	0.060	15.5	240.5	256	16	240	1.07	240.	256	-	-

To calculate P_{avg} for $n = 4$, we proceed as follows. There are five possible sequences of weights for a function to encounter upon insertion to the circular pipeline. These are illustrated in Table 3.

Table 3. Example Computation of Throughput for $n = 4$. From [4]

Sequence of Weights B and N x is either B or N, such that there are 4 B's and 12 N's.	Time in Pipeline (clocks)	Number of Combi- nations
Nxxx xxxx xxxx xxxx	1	$\binom{15}{4}$
BNxx xxxx xxxx xxxx	2	$\binom{14}{3}$
BBNx xxxx xxxx xxxx	3	$\binom{13}{2}$
BBBN xxxx xxxx xxxx	4	$\binom{12}{1}$
BBBB NNNN NNNN NNNN	5	$\binom{11}{0}$

In Table 3, an 'x' represents either a bent weight B or non-bent weight N , the exact placement of each is unimportant, but must total the $[B_n]$ and $[N_n]$ values given in Table 2. The first entry of Table 3 means that f is inserted into a stage for which it does not have a bent weight. It is ejected from the pipeline, and its total time in the pipeline is one clock. In the circular pipeline architecture, functions are always ejected immediately upon failing to test for a bent weight. Of the 15 x's following the initial N , four are bent weights and 11 are non-bent weights, which totals $B_n = 4$ and $N_n = 12$. The number of combinations for four bent weights to occur amongst 11 non-bent weights is given by $\binom{15}{4}$, as shown in the *Number of Combinations* column of Table 3.

The second entry of Table 3 illustrates the scenario that a bent weight is found in the first stage and is advanced to a second stage. In the second stage, a non-bent weight is found and f is ejected from the pipeline. For this case, f spends 2 clocks in the pipeline and there are $\binom{14}{3}$ combinations for which this can occur.

The fifth and final row of Table 3 illustrates the scenario for which f tests for four consecutive bent weights in the first four stages it encounters. Since only four bent weights reside within any 16 tests, the final 12 stages find non-bent weights. There is $\binom{11}{0}$, which is simply one. With this data we can compute the average number of clocks a function will persist in the pipeline for $n = 4$ as

$$P_{avg} = \frac{1\binom{15}{4} + 2\binom{14}{3} + 3\binom{13}{2} + 4\binom{12}{1} + 5\binom{11}{0}}{\binom{15}{4} + \binom{14}{3} + \binom{13}{2} + \binom{12}{1} + \binom{11}{0}} = 1.31 \quad (2)$$

It follows that throughput will be

$$T = \frac{2^4}{P_{avg}} = \frac{16}{1.31} = 12.2. \quad (3)$$

Hence, in a 16-stage pipeline used to sieve for 4-variable bent weights, approximately 12.2 functions can be processed each clock. Repeating the process for larger n , we note from Table 3 that T approaches the upper bound of throughput as n increases. This is due to bent weights becoming increasingly rare as n increases.

Butler [4] also ran a MATLAB simulation for $n = 2$ and $n = 4$ to find experimental values for P_{avg} and T_n . These experimental results give lower T . A goal of this thesis is to provide actual values of T , through $n = 4$, for the circular pipeline sieve run of the SRC-6.

It is to be noted that the calculations and experimentally produced values developed in this section have assumed a bent function is removed from the pipeline upon reaching a persistence of 2^n , $P_{bent} = 2^n$. However, the architecture implemented in this thesis is simplified by allowing bent functions to be extracted at only one stage. Therefore, a bent function can persist longer than 2^n , depending on where it is inserted to the pipeline relative to the location of the bent-function extraction stage. The persistence of a bent function $f_{i,bent}$ in this architecture is $2^n \leq P_{i,bent} \leq 2^{n+1} - 1$. Due to the random nature of function insertion location into the pipeline, the average of bent functions is

$$P_{bent} = \frac{2^n + 2^{n+1} - 1}{2}. \quad (4)$$

The rare nature of bent functions minimizes the impact this additional persistence will have on the average T , especially as n increases, and is ignored in the development of Table 2.

1. **Worst-Case Scenarios**

For the circular pipeline applied as the bent function sieve, these worst-case scenarios are impossible. However, they are included for completeness, as they should be considered in alternative applications of the circular pipeline.

a. With Reservoir

The worst-case scenario, which would cause the T to fall below 1, occurs when the pipeline processes only bent functions for the entire duration of the test. For the first $2^n - 1$ clocks, all functions persist in the pipeline. From clocks 2^n to $2^{n+1} - 1$, the initial 2^n functions are removed and stored as bent functions. The average persistence of this group of 2^n functions given by Equation (4). Following this initial group, T remains 1 because all remaining functions are inserted into stage one and persist exactly 2^n . Therefore, if the number of functions inserted into the circular pipeline is 2^{2^n} , the

average persistence of this worst-case scenario is $\frac{\sum_{i=1}^{2^n} i + (2^{2^n} - 2^n)}{2^{2^n}}$.

b. Without Reservoir

Without a reservoir, we have an IFG associated with each stage. The worst-case scenario begins the same as it does with a reservoir, with each stage receiving a bent function on the first clock. After 2^n clocks, new functions are inserted into stage one, also similar behavior to the with reservoir design, and persist for exactly 2^n clocks, giving a persistence of 1. However, IFG₁ will complete at which time IFG₂ will begin inserting its functions; it was previously blocked from inserting functions because S_1 was passing a function on every clock. The P of all functions produced by IFG₂ will be the worst case of $2^{n+1} - 1$. This pattern continues around the circular pipeline; IFG₃'s

functions persist $2^{n+1} - 2$ clocks, IFG₄'s functions persist $2^{n+1} - 3$ clocks, and so forth. Therefore, the average persistence of this worst-case scenario is equal to P_{bent} , given in Equation (4).

E. SUMMARY

In this chapter, the circular pipeline design concept was outlined; associated data flow and conceptual issues were addressed. The next chapter covers implementation of the circular pipeline concept in hardware.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

The circular pipeline and all associated components, such as the reservoir, were constructed in Verilog hardware description language and run on the SRC-6. The process of accomplishing this is the topic of this chapter.

A. VERILOG IMPLEMENTATION

The circular pipeline architecture Verilog code is fully scalable to any n by modification of a single parameter. Behavioral Verilog augmented with a handful of structural statements is the coding style used. Most of the implementation of the design described in Chapter III into Verilog was straightforward and is not described in further detail. An overview of the Verilog design's components and highlights of some specific issues are discussed in this section. The full Verilog code is in the Appendix.

1. Reservoir

The reservoir is the most complex component in the circular pipeline design, including the circular pipeline itself. The three main components of the reservoir are priority encoders, adders, and registers.

a. Priority Encoders

$2^{n+1} - 2$ priority encoders are generated for the reservoir, one for each register except for the topmost Q_{2^n-2} register. The priority encoders for the $2^n L$ registers each have 2^n inputs, one for each of the T functions applied by the input MUX. The number of inputs to the priority encoders for each Q register tapers off as $2^n - i$.

Starting with L_0 and working up, each register's priority encoder produces the lowest-indexed function I_i that is being rejected from the circular pipeline and not routed to a lower-indexed register. If there is no function to be routed to a given register, its priority encoder produces all zeros.

b. Adders

Adders are used to produce the number of vacant registers below each I function. This number is the routing information needed to place a rejected function I_i into the proper register, ensuring the reservoir is filled from the bottom up. The assurance that the reservoir is filled from bottom up allows use of a thermo adder to produce the value of vacant registers.

The $2^n - 1$ occupied-bits l associated with the L registers are applied to the thermo adder if the topmost L register L_{2^n-1} is not occupied. If L_{2^n-1} is completely filled, the occupied-bits of the Q registers q are applied to the thermo adder. This is because, when L_{2^n-1} is occupied, all of the L registers are transferred out of the reservoir to the input MUX and, simultaneously, all of the Q registers are transferred index-to-index into the P registers on the next positive clock edge. The number of occupied registers on the next positive clock edge is needed for proper routing of I functions. Therefore, the l bits are applied to the thermo adder when L_{2^n-1} is not occupied, and the q bits are applied when L_{2^n-1} is occupied.

The thermo adder's Verilog code begins by inspecting the most significant occupied bit q or l and proceeding down the indices. Upon finding an asserted occupied bit, it is known that all less significant bits will also be asserted, and a value of $i + 1$ is returned.

The output of the thermo adder is fed into a chain of $2^n - 1$ adders, one for each I function above I_0 . I_0 receives its sum used for routing directly from the thermo adder. Each adder increases the input value by 1 if I_{i-1} is being routed to the reservoir and provides this sum to I_i and the next adder in the chain. The adder chain begins with the sum provided by the thermo adder and continues the running sum by adding the NOT of the bit $inToPipe_i$ that corresponds to its function I_i . This running sum indicates the number of functions that will remain in the reservoir below each I_i on the next positive clock edge.

c. Registers

The $2^{n+1} - 1$ registers required by the reservoir are assigned within an `always@(posedge CLK)` statement. This statement instantiates a register and is used only once within the reservoir code for the purpose of creating the registers. Every register receives its input through a MUX that selects between the output of its priority encoder or the register's current value. Each L_i register has Q_i as an additional input to its MUX for the cases that the Q registers slide down.

2. Circular Pipeline

The circular pipeline is implemented using several modules that carry out the operations described in the previous chapter. A function was created to describe the behavior of a standard stage of the pipeline. This function is called $2^n - 1$ times. A modified version of the standard pipeline stage function that has the additional functionality of removing FUTs it determines to be bent (based on persistence) is instantiated once. This gives a total of 2^n stages. The remainder of the module consists of control signals used to direct the flow of functions through the pipeline.

B. VERILOG DESIGN DEVELOPMENT AND TESTING

Project development was managed with Xilinx ISE 10.1. Synplify Pro D-2009.12 was used for synthesis and ModelSimSE 6.4 was used for simulation. The general process was to build a section of code and synthesize. The synthesis report was then used to correct any errors or warnings. Then synthesis would be run again. This process was iterated until synthesis produced an error- and warning-free circuit that appeared reasonable in the register transfer level (RTL) view. Figure 9, 10, and 11 are examples of RTL schematics of a single circular pipeline stage for $n = 4$.

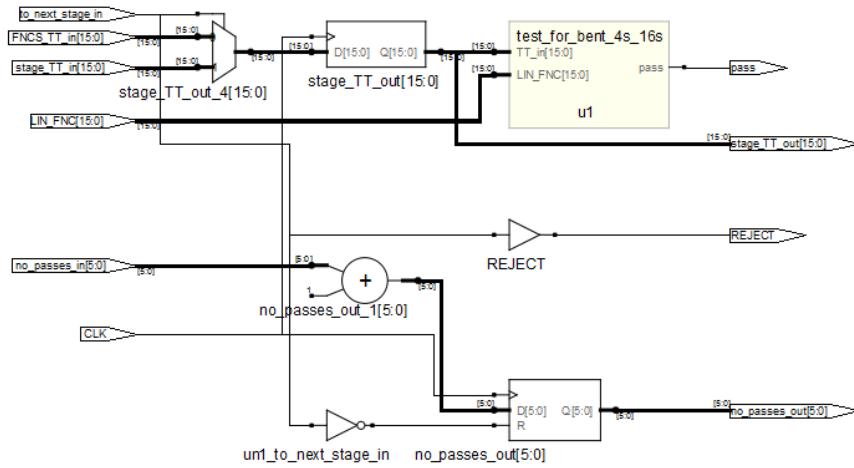


Figure 9. Synplify Pro RTL View of a Circular Pipeline Stage. $n = 4$.

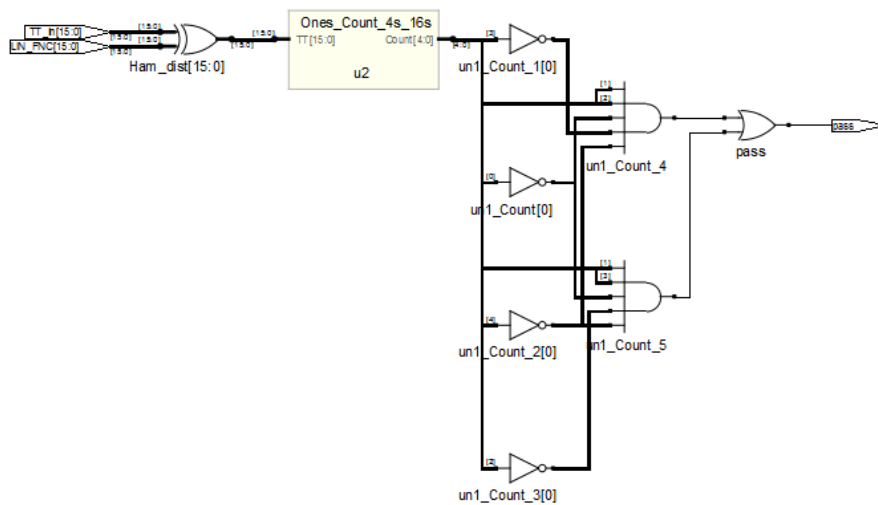


Figure 10. Synplify Pro RTL View of the Bent Weight Tester Within a Stage. $n = 4$.

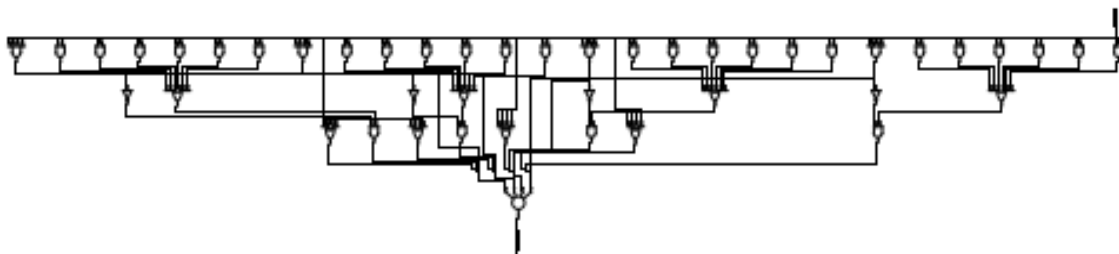


Figure 11. Synplify Pro RTL View of a One's Counter Within a Bent Weight Tester. $n = 4$.

1. Macro Characteristics

The input/output requirements of the Verilog coded circular pipeline, known as a macro in SRC-6 literature, must be characterized in order to choose an appropriate implementation. The circular pipeline requires no input aside from the system clock. It produces outputs that are held for one clock at unpredictable times throughout macro execution. This is a marked difference from the conventional macro design, which was called, returned a value, and terminated on each clock (the function generator was located outside of the macro). This highly regular behavior allowed for the use of the simplest of macro implementation—pure functional.

With the characteristic that the macro returns values while continuing its run, vice returning a value at run termination, an external macro was also unfit for the circular pipeline implementation. A stateful macro remained the only possibility among the known types, but uncertainty remained on its suitability. Finally, on the advice of an SRC engineer, a streaming external macro was explored and found fit to the circular pipeline's characteristics [7].

2. Streaming Output

Streaming output allows for data to be returned from the circular pipeline and stored in On Board Memory (OBM) on any clock throughout the duration of the sieving process. With the implemented circular pipeline returning a maximum of one function per clock, no bottleneck will occur so long as $n < 7$. For $n \geq 7$, the function width is greater than 64 bits, and so a bottleneck could occur over the 64-bit bus used to transfer data from the macro to OBM.

While this was not a concern, in implementations installed for this thesis due to other limiting factors preventing $n \geq 7$, the stream construct can handle such a case. The SRC-6 stream construct includes a buffer that can be configured to handle a backlog of data outflow and stall the circular pipeline until the backlog is processed (e.g. transferred out).

3. CPU

Top-level control is maintained by the CPU by the main.c file. The main.c file allocates memory, calls a subroutine that leads to the macro, and prints results.

4. Subroutine and Macro Call

The subroutine is an interface between the main.c file and the macro. It is written in C-style code, but implemented on the FPGA. The subroutine sets up data types, calls the macro in a way that supports streaming, and passes data from OBM to the CPU. In addition to the subroutine, the files info and blk.v configure the interface between the CPU and the macro. They declare the input/output data types and sizes.

5. Timing

For $n \leq 5$, all timing conditions are met with the circular pipeline, as describe to this point. For $n = 6$, the mapper and place and route application are unable to meet the timing constraint along the critical path. The SRC-6 uses a fixed clock of 100 MHz, which means delay along every path must be equal to or less than 10ns.

The place and route application was unable to meet the 10ns timing constraint along all paths for $n = 6$. However, the circular pipeline behaved as expected at runtime for the sample set of function used. Thus, the critical paths identified by the place and route application are probably not the true critical paths of the circular pipeline. Rather, they are theoretical worst-case paths that the place and route application was unable to eliminate as possibilities.

6. FPGA Resources

For $n < 7$, the resources of a single Xilinx Virtex2 XC2V6000 FPGA are sufficient to realize the circular pipeline. For larger n , moderate changes to the SRC-6 implementation strategy must be adapted. Further details are included in Chapter VI. Exact resource usage data for $n < 7$ is included in Chapter V.

D. SUMMARY

In this chapter, the development process for circular pipeline implementation onto the SRC-6 was covered. The next chapter provides a results from the implemented circular pipeline.

V. RESULTS

A. SPEEDUP

Speedup results of the circular pipeline with IFC are summarized in Table 4. The clocks columns give the total number of clocks that the implemented design required to complete an exhaustive test. T_n is throughput, Upper Bound is the maximum possible, and Realized is what was achieved at runtime. This data is from the implemented architecture running on the SRC-6, so it includes latency and overhead associated with SRC-6 process control. For small n , this overhead is a large percentage of the clocks needed for test completion. This is why the speedup for $n \leq 3$ does not closely match the realized T_n . For $n > 3$, the overhead is a very small percentage of total number of clocks required to complete the exhaustive test. While the conventional design maintains a $T_{n>3}$ of nearly unity, the increased $T_{n>3}$ becomes the speedup realized, rendering $T_{n>3}$ equivalent to the speedup.

Due to excessive computational time requirements, on the order of decades, complete results for $n = 6$ are impossible. However a test set of 3.2×10^{14} ($1.7 \times 10^{-3}\%$ of all 2^{2^6} functions required for an exhaustive test) were run and the results are prorated to give a value for the complete enumeration. Asterisks denote these values.

T is calculated by dividing the number of functions processed by the number of clocks.

$$T_n = \frac{2^{2^n}}{\text{Clocks}} \quad (5)$$

For example,

$$T_4 = \frac{2^{2^4}}{7,840} = 8.36$$

Speedup is calculated by dividing the circular pipeline's clocks by the conventional design's clocks.

Table 4. Realized Speedup.

n	Circular Pipeline T_n		Conventional T_n		Clocks		Speedup
	Upper Bound	Realized	Upper Bound	Realized	Conventional	Circular	
2	4	0.296	1	0.078	205	54	3.8
3	8	2.15	1	0.573	446	119	3.7
4	16	8.36	1	0.997	65,727	7,840	8.4
5	32	21.7	1	1	42.9×10^8	1.98×10^8	21.7
6	64	55*	1	1	184×10^{17} *	3.33×10^{17} *	55*

*Estimate based on small sample size (number of functions tested $\ll 2^{2^n}$)

From Table 4, it is noted that a 55 times speedup over the conventional sieve design is achieved by the circular pipeline. More importantly, there is a trend of increasing speedup as n increases. Figure 13 is a graph of this trend juxtaposed with the upper bound of 2^n ; it is concluded that the speedup achieved by the circular pipeline is on the order of 2^n .

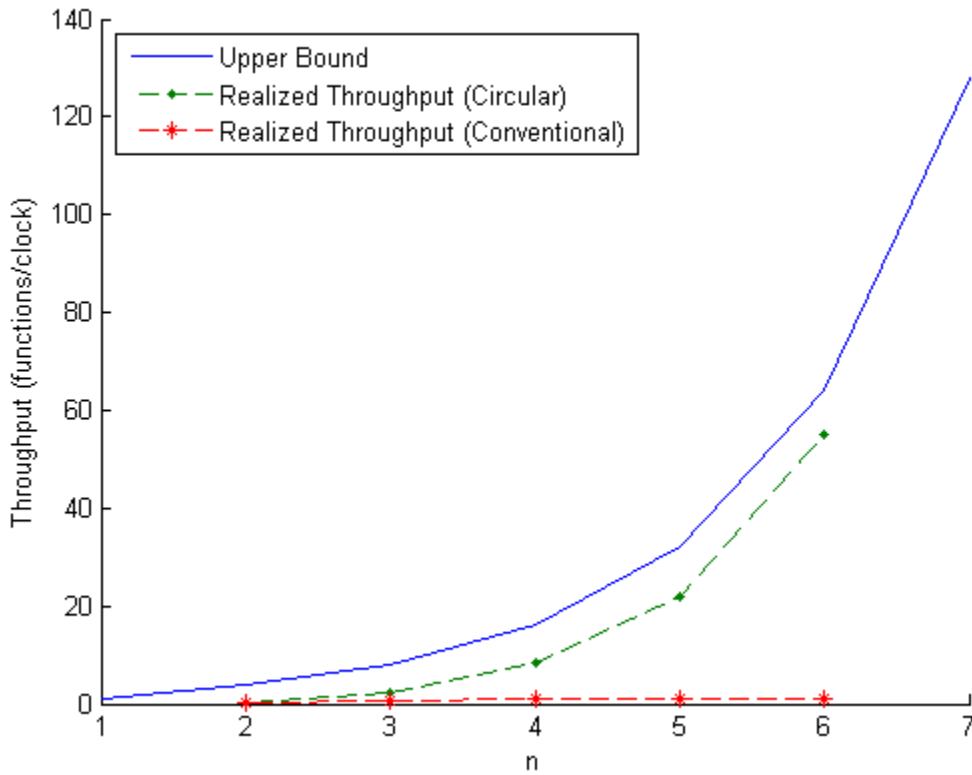


Figure 13. Realized Throughput.

The throughput plotted in Figure 13 does not simply follow the upper bound at a reduced fraction, but approaches the upper bound as n increases. This conclusion is best illustrated in Figure 14, which is normalized to 2^n .

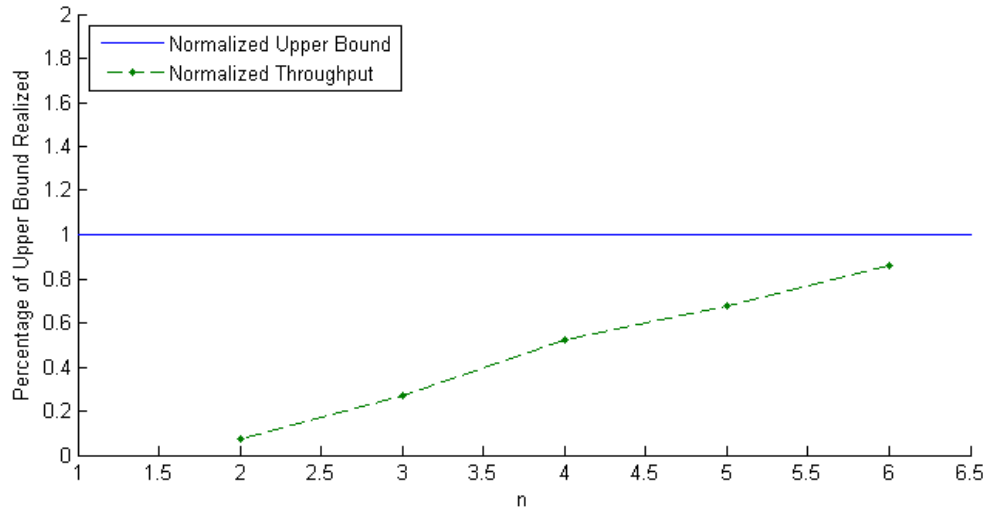


Figure 14. Throughput Normalized to 2^n .

B. RESOURCES

A comparison of resources consumed between the circular pipeline and conventional design is provided in Table 5. The three resource categories are given as percentages of the resources available on the Xilinx Virtex-II FPGA. A slice is the basic building block of the FPGA. Each of the 44,096 slices contain two D flip-flop registers and two 4-input Lookup Tables (LUTs), for a total of 88,192 each. From Table 5, we conclude that LUTs are the limiting factor, as they are consumed at a higher rate than registers as n increases. Therefore, the column Circular Pipeline Resource Multiple is the fraction given by the 4-input LUTs percentage consumed by the conventional design divided by the percentage consumed by the circular design.

For $n \leq 4$ the circular pipeline consumes fewer resources than the conventional design, as shown in Table 5. This is an unexpected and not well understood result. For $n \leq 7$, the additional resources consumed are less than a multiple of three over the conventional design. The additional resource consumption of the circular pipeline is attributed to its control logic.

Table 5. Resources Consumed Summary.

n	Design	Registers (%)	Occupied Slices (%)	4-input LUTs (%)	Circular Pipeline Resource Multiple
2	Conventional	4	3	3	1
	Circular	1	1	3	
3	Conventional	4	6	3	1
	Circular	1	2	3	
4	Conventional	5	7	4	0.75
	Circular	3	5	3	
5	Conventional	5	9	6	1.17
	Circular	5	10	7	
6	Conventional	7	17	13	2.31
	Circular	23	25	30	
7	Conventional	9	42	38	2.47
	Circular	50	113	94	

C. RESERVOIR TRADEOFF

The use of a reservoir to queue and equitably distribute generated function among the stages provides the fastest computation. However, the large demand on logic resources and associated delay rendered its implementation unrealizable for $n > 3$. For $n \geq 4$, the worst-case path delay renders a maximum frequency of less than 30 MHz. Attempts to pipeline the reservoir for the purpose of decreasing delay such that the 100 MHz fixed clock of the SRC-6 could be used were successful.

A comparison between the circular pipeline (without reservoir) and the circular pipeline with reservoir is provided in Table 6. The number of clocks given for $n \geq 4$ in Figure 7 are simulation results, not runtime data from the SRC-6 like all other numbers. Circuits for $n \geq 4$ are unrealizable, so simulation results are required to make speedup comparison. In practice, if the circular pipeline with queue architecture is to be implemented, it would require more registers than what was reported for the unrealizable circuit that was synthesized. However, even with double the registers, LUTs would still

be the limiting factor. The number of LUTs is expected to remain constant, so the LUT comparison for $n \geq 4$, which is data taken from the map report, is valid. From Table 6 and the maximum frequency for $n \geq 4$ being less than 30 MHz, it is clear that the resource and timing demands of the reservoir cannot be met for large n and the simpler design is better suited for the task.

Table 6. Circular Pipeline With and Without Reservoir (Res) Comparison.

n	Clocks		Speedup	LUTs		Resource Multiple
	Res	w/o Res		Res	w/o Res	
2	45	54	1.20	3	3	1
3	111	119	1.07	3	3	1
4	7,259	7,815	1.08	13	3	4.33
5				70	7	10

The speedup produced by the reservoir is limited by the delta between completion times of the IFG. From Figure 15, we conclude that the trend responsible for a significant portion of the maximum delta in completion times is due to using only one stage to remove bent functions. An effect of using just one output stage is that a bent function will persist $2^n \leq P_{bent} \leq 2^{n+1} - 1$, depending into which stage it is inserted. The stage are numbered from 1 to 16 in Figure 15, beginning with the stage that results in optimal P_{bent} and ending with worst case stage. As n increases, this effect will be reduced as bent function become rarer. Figure 15 is a plot of additional clocks required by each IFG_i after the first IFG completed. This value is given as a percentage of the total clocks required for the complete computation. IFG_{16} terminates 1667 clocks after IFG_1 , which is 21.3% of the total clocks consumed.

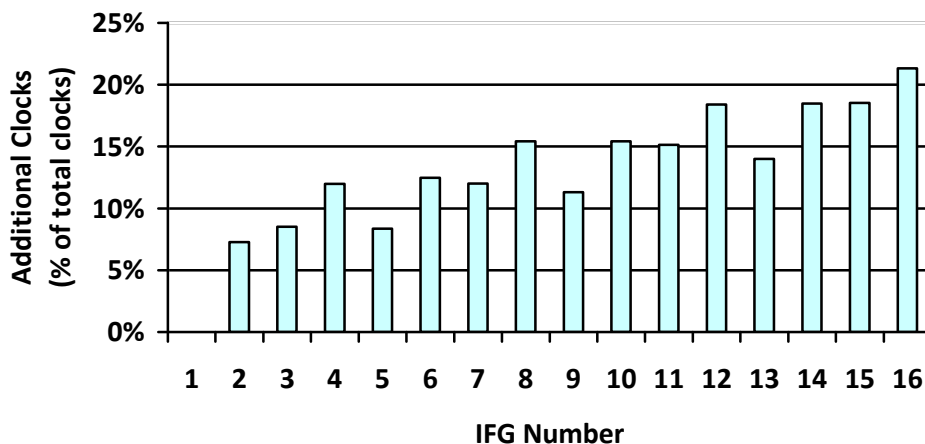


Figure 15. Relative Completion Times of the IFG.

D. SUMMARY

The circular pipeline results in a speedup on the order of 2^n over the conventional architecture used to exhaustively sieve for n -variable bent functions. This speedup is achieved with a small fraction of logic resources compared to what is required to achieve a similar speedup with the conventional architecture.

For $n = 6$, a speedup of 55 times is realized with a resources increase of 2.3 times. With the conventional design, a similar speedup would require a logic resources increase of 55 times. This is because the only way to increase speedup with the fixed throughput of the conventional design is the duplicate the circuit and distribute functions to be tested equally between the duplicated circuits. Speedup gained in this way is utilizing parallelism; doubling the instances of the circuit doubles the throughput. This method of gaining speedup is amenable to the circular pipeline as well. However, for $n = 6$, allocating triple the logic resources of a conventional design and replacing it with the circular pipeline will achieve a speedup of 55 times, vice three times.

In this chapter, the throughput and resource consumption of implemented circular pipelines were presented and analyzed. The next chapter concludes this thesis with recommendation for further research.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSION

The circular pipeline architecture was implemented on the SRC-6 and demonstrated speedup on the order of 2^n . This speedup is realized with a logic resources increase of less than threefold for $n < 7$. For $n = 6$, the ratio of speedup to logic resources increase over conventional architecture is 55:2.3. Previous speedup gains were limited to increases in parallelism, which yield a 1:1 ratio of speedup to logic resources consumption increase. The circular pipeline is an efficient means of increasing throughput in sieving applications.

The reservoir developed for this thesis provides for the most efficient use of the circular pipeline by redistributing functions equitably. However, the delta of run time between the IFGs is minor. Therefore, the cost in complexity of the reservoir is not worth the speedup gained. Yet, the reservoir could be essential if the circular pipeline is applied to other applications without same characteristics of the bent-function sieve providing for an even distribution of passed and rejected functions among the stages.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

1. Multiple Output Stages

The design presented in this thesis was assuming a hard limitation of a single 64-bit output bus. This motivated the design to restrict output from a single stage. In order to run the circular pipeline on the SRC-6, techniques new to the Naval Postgraduate School were implemented. Namely, the use of output streams was critical for the circular pipeline's behavior. While learning the use of output streams, it was realized that up to 16 1024-bit wide output streams can be used. The streams have a programmable buffer mechanism to take care of any bottleneck problems over the 64-bit output bus. Using all

16 of these output streams (for $n \geq 4$) should be a fairly simple improvement to implement. This will result in more LUTs required for the additional stages tasked with examining the persistence token, but will improve throughput.

2. Pipelined Reservoir

As noted in Chapter IV, pipelining attempts with the circular pipeline with reservoir design failed. However, it may be possible. If the circular pipeline is to be applied to other applications, the reservoir will likely be more important, so pipelining it to reduce the worst-case path delay could be important.

3. Multiple FPGAs

For $n \geq 7$, the circular pipeline design does not fit on a single Virtex-II FPGA. Multiple FPGAs must be used for these cases. This is a nontrivial SRC-6 implementation issue that will also require modification to the Verilog code. Solving this issue will likely have the most impact on the continuing bent-function research at the Naval Postgraduate School.

4. Function Generators

While this thesis focuses on speedup via hardware design, the most important speedups moving forward will be gained by reducing the number of functions that require testing. This is the current focus of the continuing bent functions research at the Naval Postgraduate School. Understanding special characteristics of bent functions and using this understanding to eliminate many of the functions included in an exhaustive test is the first step. Building a function generator to produce only these functions is the second step. For the circular pipeline produced in this thesis, it is important that the 2^n IFG produce, on average, functions with the same total number of bent weights.

This area of research requires in-depth mathematical understanding of bent functions as well as ingenuity with Verilog hardware design. In return, it will likely produce the most significant results.

APPENDIX. PROGRAMMING CODE

A. VERILOG

1. Circular Pipeline With Independent Function Generators

```
//-----  
// MY_CIRC_Pipe.v - An interface between the circular pipeline code that sets up streaming  
//                  with the SRC-6. Based on the SRC example user_one_stream.  
//  
// Created:        August 7, 2010  
// Last Modified:  September 3, 2010  
// Author:         Chris Johnson  
//  
// Notes:         DATA_OUT bus width is not parameterized; must be manually edited for n>5.  
//                  modDATA_OUT must be edited for n>6.  
//  
// Sub-module calls: CircPipe.v  
//-----  
  
module MY_CIRC_PIPE (  
    input          START,  
    input          CLK,  
    input          CLR,  
    output reg     DONE,  
    output reg [31:0] DATA_OUT,  
    output reg     VALID_OUT,  
    input          STALL_IN,  
    output reg     TERM_OUT  
);
```

```

//parameter names for the states
localparam IDLE      = 0;
localparam ACTIVE    = 1;
localparam STALLED   = 2;
localparam FINISHING = 3;

reg  [1:0]      state;

//wire connections from module call
wire          modDONE;
wire [63:0]    modDATA_OUT;
wire          modVALID_OUT;
wire          modTERM_OUT;

always @*
    if (CLR) begin
        DATA_OUT <= 0;
        DONE      <= 0;
        VALID_OUT <= 0;
        TERM_OUT  <= 0;
        state     <= 0;
    end
    else
        case (state)
            IDLE:      if (START) begin
                            DATA_OUT <= 0;
                            VALID_OUT <= 1;
                            state     <= ACTIVE;
                        end

            ACTIVE:    begin
                            DATA_OUT <= modDATA_OUT;
                            DONE <= modDONE;
                            VALID_OUT <= modVALID_OUT;
                            TERM_OUT <= modTERM_OUT;
                            state <= ACTIVE;
                            if (STALL_IN) begin

```

```

                VALID_OUT <= 0;
                state      <= STALLED;
                end
            end

    STALLED:      if (~STALL_IN) begin
                  VALID_OUT <= 1;
                  state      <= ACTIVE;
                end

    FINISHING:    begin
                  state <= IDLE;
                end

                default;;
    endcase

CircPipe u1(START,CLK,CLR,modDONE,modDATA_OUT,modVALID_OUT,STALL_IN);

endmodule

```

```

//-----
// CircPipe.v - The circular pipeline with independent function generators top level module.
//
// Created:      December 22, 2009
// Author:       Jon T. Butler
// Last Modified: September 3, 2010
// Modified by:  Chris Johnson
//
// Notes:        Set parameter 'n' in this file.  It is passed to all sub-modules.
//
// Called by:     MY_CIRC_PIPE.v
//
// Sub-module calls:  countersMod.v
//                  Stage_TT.v
//

```

```

// This implements the circular pipeline. For n-variable functions, there are N = 2**n stages,
// one for each linear function (we need only compare against the linear functions, since a
// function that has a bent distance from all linear function, has a bent distance away from
// all affine functions. In this realization, only one stage has a bent function output - to
// simplify the circuit. In this way, the circular pipeline serves as a buffer. In this
// case a bent function will go through from N to 2N-1 stages.
//
//-----
//
module CircPipe #(parameter n=6, parameter N=2**n) //n is number of variables. N is # of bits in func's TT.
(
    input          START,
    input          CLK,
    input          CLR,
    output reg     done,          //Asserted when all counters are done & pipe empty.
    output [63:0]  BENT,
    output         valid_out,    // Indicates a valid bent function is at BENT.
    input         STALL_IN
);

wire [N-1:0]      countDone;          // Set when counter has completed one cycle
wire [N-1:0]      LIN_FNC [N-1:0];
wire [N-1:0]      REJECT;            // 0 bit indicates FNCS word not accepted.
reg              temp;
wire [N-1:0]      FNCS [N-1:0];      // Each of the N words in counter FNCS has N bits.
wire [n-1:0]      FNCSheb [N-1:0];  // High order bits for the counter
wire [N-n-1:0]    counter [N-1:0];  // N simple counters, extra bit to signal counter is done
wire [N-1:0]      to_stage;
wire [N-1:0]      stage_TT [N-1:0];
wire [n+1:0]      no_passes [N-1:0];

genvar g;

////////////////////////////////////

////CREATE INDEPENDENT FUNCTION GENERATORS (IFG)////////////////////////////////
////Instantiate independent counters for function gens////////////////////////////////

```

```

generate
for (g=0; g<N; g=g+1)
    begin: CountersGen
        countersMod #(.n(n)) u4(START,CLK,CLR,STALL_IN,REJECT[g],counter[g],countDone[g]);
    end
endgenerate

generate
//Generate high order bits
for (g=0; g<N; g=g+1)
    begin: CounterHOB
        assign FNCSshob[g] = g;
    end
endgenerate
//Generate counters
generate
for (g=0; g<N; g=g+1)
    begin: CounterConcat
        assign FNCS[g] = countDone[g] ? {N{1'b0}} : {FNCSshob[g],counter[g]};
    end
endgenerate
////CREATE INDEPENDENT FUNCTION GENERATORS (IFG)////////////////////////////////////

////TERMINATION SIGNAL////////////////////////////////////
always@*
    if(countDone[N-1:0] == {N{1'b1}} && to_stage[N-1:0] == {N{1'b0}})
        done <= 1'b1;
    else
        done <= 1'b0;
////TERMINATION SIGNAL////////////////////////////////////

////LINEAR FUNCTIONS////////////////////////////////////
generate
for (g=0; g<N; g=g+1)
    begin: LinearGen
        assign LIN_FNC[g] = Linear(g);
    end
end

```

```

endgenerate

function [N-1:0] Linear(input [n-1:0] Y);
    integer j;
    integer k;
    reg [n-1:0] X;
    begin
        for (j=0; j<N; j=j+1)
            begin
                X = j;
                temp=0;
                for (k=0; k<n; k=k+1)
                    begin
                        temp = temp ^ (X[k] & Y[k]);
                    end
                Linear[N-1-X] = temp;
            end
        end
    endfunction
//////LINEAR FUNCTIONS//////////////////////////////////////

//////INSTANTIATE STAGES//////////////////////////////////////
generate
for (g=0; g<N; g=g+1)
    begin: Stages
        if(g != 0) begin
            stage #(.n(n)) u2(CLK, FNCS[g], REJECT[g], to_stage[g-1], to_stage[g], stage_TT[g-1],
LIN_FNC[g], stage_TT[g], no_passes[g-1], no_passes[g], countDone[g]);
            end
        if(g == 0) begin
            stage1 #(.n(n)) u3(CLK, FNCS[g], REJECT[0], to_stage[N-1], to_stage[0], stage_TT[N-1],
LIN_FNC[0], stage_TT[0], no_passes[N-1], no_passes[0], countDone[g], BENT, valid_out);
            end
        end
    endgenerate
//////INSTANTIATE STAGES//////////////////////////////////////

```

```
endmodule
```

```
//-----  
// countersMOD.v - Instantiates an inhabitable counter.  
//  
// Created:      August 11, 2010  
// Author:      Chris Johnson  
// Last Modified: September 3, 2010  
//  
// Notes:      This counter is the lower N-n-1 bits of the function gen in CountersMod.v.  
//  
// Called by:      CountersMod.v  
//  
// Sub-module calls:  None  
//  
//-----  
//  
module countersMod #(parameter n = 6, parameter N=2**n)  
    (    input    START,  
      input    CLK,  
      input    CLR,  
      input    STALL_IN,  
      input    REJECT,  
      output   reg [N-n-1:0] counter,  
      output   reg    countDone  
    );  
reg [1:0] state = 0;  
  
always@(posedge CLK, posedge CLR)  
    if(CLR) begin  
        countDone <= 0;  
        counter <= 0;  
        state <= 0;  
    end  
    else  
        case(state)
```

```

0: if (START) begin
    counter <= 0;
    state <= 1;
    countDone <= 0;
    end
1: begin //counter active
    if(!REJECT && !STALL_IN)
        counter <= counter + 1;
    if(counter == 2**(N-n)-1)
    begin
        state <= 2;
    end
    end
2: begin //counter complete
    countDone <= 1'b1;
    counter <= {N{1'b0}};
    state <= 0;
    end
default;;
endcase
endmodule

```

```

//-----
// stage.v - One (simple) stage only.
//
// Created:      December 22, 2009
// Author:       Jon T. Butler
// Last Modified: September 3, 2010
// Modified by:  Chris Johnson
//
// Notes:        This does NOT put out a bent function.
//
// Called by:    CountersMod.v
//
// Sub-module calls:  test_for_bent.v
//

```

```

//-----
//
module stage #(parameter n=6, parameter N=2*n) //n is number of variables. N is # of bits in func's TT.
(
    input                CLK,
    input                [N-1:0] FNCS_TT_in,
    output reg          REJECT,
    input                to_next_stage_in,
    output               pass,
    input                [N-1:0] stage_TT_in,
    input                [N-1:0] LIN_FNC,
    output reg          [N-1:0] stage_TT_out,
    input                [n+1:0] no_passes_in,
    output reg          [n+1:0] no_passes_out,
    input                countDone
);

test_for_bent #(.n(n)) ul(stage_TT_out,LIN_FNC,passU1);
and stgs(pass,passU1,valid); //output pass signal if input is valid and TT passes

always@* //Can prune this signal and just use to_next_stage_in
    if(to_next_stage_in==1)
        REJECT <= 1;
    else
        REJECT <= 0;

always@(posedge CLK)
    if(to_next_stage_in==1) //Data to this stage comes from previous stage.
        begin
            stage_TT_out <= stage_TT_in;
            valid <= 1;
            no_passes_out <= no_passes_in + 1;
        end
    else //Data to this stage comes in from input buffer.
        begin
            stage_TT_out <= FNCS_TT_in;
            valid <= !countDone; //valid iff counter is not yet done
        end
endmodule

```

```

        no_passes_out <= 0;
    end
endmodule
//////////////////////////////////// RESULTS //////////////////////////////////////
// n =           2           4           6           8           10           12
// Freq.         181.8       144.8       73.0       53.4       42.9       35.9
// #LUTs (%)     16(0%)     67(0%)   304(0%)  1251(1%)  5384(7%)  22179(32%)
// Reg.Bits not i/o 4(0%)   23(0%)   77(0%)   283(0%)  1037(1%)  4352(6%)

////////////////////////////////////

//-----
// stagel.v - One stage only.
//
// Created:      December 22, 2009
// Author:       Jon T. Butler
// Last Modified: September 3, 2010
// Modified by:  Chris Johnson
//
// Notes:       This does put out a bent function.
//
// Called by:    CountersMod.v
//
// Sub-module calls: test_for_bent.v
//-----
//
module stagel #(parameter n=6, parameter N=2**n) //n is number of variables. N is # of bits in func's TT.
(
    input          CLK,
    input          [N-1:0] FNCS_TT_in,
    output reg     REJECT,
    input          to_next_stage_in,
    output reg     to_next_stage_out,
    input          [N-1:0] stage_TT_in,

```

```

        input      [N-1:0]    LIN_FNC,
        output reg [N-1:0]    stage_TT_out,
        input      [n+1:0]    no_passes_in,
        output reg [n+1:0]    no_passes_out,
        input      countDone,
        output reg [N-1:0]    BENT,
        output reg valid_out
    );

    wire passU1;
    reg  valid;

    test_for_bent #(.n(n)) u1(stage_TT_out,LIN_FNC,passU1);
    and stgs1(pass,passU1,valid); //output pass signal if input is valid and TT passes

    always@* to_next_stage_out <= (pass && (no_passes_out < N));

    always@*
        if(to_next_stage_in==1)
            REJECT <= 1;
        else
            REJECT <= 0;

    always@(posedge CLK)
        if(no_passes_out >= N)
            begin
                BENT <= stage_TT_out;
                valid_out <= 1;
            end
        else
            begin
                BENT <= {N{1'b0}};
                valid_out <= 0;
            end
    always@(posedge CLK)
        if(to_next_stage_in==1) //Data to this stage came from previous stage.
            begin

```

```

        stage_TT_out <= stage_TT_in;
        no_passes_out <= no_passes_in + 1;
        valid <= 1;
    end
else
    begin
        stage_TT_out <= FNCS_TT_in;
        no_passes_out <= 0;
        valid <= !countDone; //valid iff counter is not done
    end
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//-----
// test_for_bent.v - Compares nonlinearity with the two possible bent weights for n.
//
// Created:          December 22, 2009
// Author:           Jon T. Butler
// Last Modified:   September 3, 2010
// Modified by:     Chris Johnson
//
// Notes:           Nonlinearity is returned from Ones_Count.v
//
// Called by:       stage.v
//                  stage1.v
//
// Sub-module calls:  Ones_Count.v
//-----
//
module test_for_bent #(parameter n=6, parameter N=2**n) //n is number of variables. N is # of bits in TT.
    (
        input      [N-1:0]    TT_in,
        input      [N-1:0]    LIN_FNC,
        output reg                pass
    );
//

```

```

parameter n = 6;                // n = number of variables
localparam N = 2**n;           // N = number of bits in truth table of an n-variable function.
//
reg    [N-1:0] Ham_dist;
wire   [n:0]   Count;

always @*
    begin
        Ham_dist = TT_in ^ LIN_FNC;
        if(Count == 2**(n-1) - 2**(n/2-1) || Count == 2**(n-1) + 2**(n/2-1))
            pass = 1;
        else
            pass = 0;
    end

//
Ones_Count u2 (Ham_dist, Count);
defparam u2.n = n;
//
endmodule

//////////////////////////////////// RESULTS //////////////////////////////////////

// n =                2            4            6            8            10
// Freq.              140.8        94.1        55.5        44.0        35.5
// #LUTs (%)          5(0%)       46(0%)    219(0%)   949(1%)   3421(3%)

////////////////////////////////////

////////////////////////////////////
module Ones_Count(TT, Count);
//-----
// Ones_Count.v - A program to count the number of 1's in HD (Hamming Distance), producing that
//                  count at Count. This version of Ones_Count.v uses functions.
//
// Created:         August 18, 2007
// Last Modified:   December 26, 2009

```

```

// Author:      Jon T. Butler
//
// Inputs:      TT
// Outputs:     Count
//
// Notes:       1. For n=2, this circuit builds a 4-input 3-output 1s count circuit that is intended to
//               make efficient use of the 4-input LUTs in the SRC's FPGA.
//
//-----

parameter n = 10; // At n=6, freq = 79.9 MHz. and it does not compile at n=7.
localparam N = 2**n;
output [n:0] Count;
input [N-1:0] TT;
reg [n:0] Count; // If Count is wire, ModelSim complains of "illegal reference to net
                 // Count" below. I believe it is because Count should be declared a
                 // reg, per discussion on p. 178 of Palnitkar. Unfortunately, this
                 // is not a combinational logic circuit. Using 'task' does not seem
                 // to help. Both input and output variables must be reg.

always @(TT)
  begin: CHECK_n
    case(n)
      2: Count <= Count2(TT);
      3: Count <= Count3(TT);
      4: Count <= Count4(TT);
      5: Count <= Count5(TT);
      6: Count <= Count6(TT);
      7: Count <= Count7(TT);
      8: Count <= Count8(TT);
      9: Count <= Count9(TT);
     10: Count <= Count10(TT);
     11: Count <= Count11(TT);
     12: Count <= Count12(TT);
    endcase
  end

```

```

//*****\
//***** The 1's count function - Count10 for 12-variable functions *****\
function [12:0] Count12;
    input [4095:0] TT;

    begin: f12
        Count12 = Count11(TT[4095:2048]) + Count11(TT[2047:0]);
    end
endfunction

//***** The 1's count function - Count12 for 12-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count11 for 11-variable functions *****\
function [11:0] Count11;
    input [2047:0] TT;

    begin: f11
        Count11 = Count10(TT[2047:1024]) + Count10(TT[1023:0]);
    end
endfunction

//***** The 1's count function - Count9 for 11-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count10 for 10-variable functions *****\
function [10:0] Count10;
    input [1023:0] TT;

    begin: f10
        Count10 = Count9(TT[1023:512]) + Count9(TT[511:0]);
    end
endfunction

//***** The 1's count function - Count10 for 10-variable functions *****\
//*****\
//*****\

```

```

//***** The 1's count function - Count9 for 9-variable functions *****\
function [9:0] Count9;
    input [511:0] TT;

    begin: f9
        Count9 = Count8(TT[511:256]) + Count8(TT[255:0]);
    end
endfunction

//***** The 1's count function - Count9 for 9-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count7 for 7-variable functions *****\
function [8:0] Count8;
    input [255:0] TT;

    begin: f8
        Count8 = Count7(TT[255:128]) + Count7(TT[127:0]);
    end
endfunction

//***** The 1's count function - Count7 for 7-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count7 for 7-variable functions *****\
function [7:0] Count7;
    input [127:0] TT;

    begin: f7
        Count7 = Count6(TT[127:64]) + Count6(TT[63:0]);
    end
endfunction

//***** The 1's count function - Count7 for 7-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count6 for 6-variable functions *****\

```

```

function [6:0] Count6;
    input [63:0] TT;

    begin: f6
        Count6 = Count5(TT[63:32]) + Count5(TT[31:0]);
    end
endfunction

//***** The 1's count function - Count6 for 6-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count5 for 5-variable functions *****\
function [5:0] Count5;
    input [31:0] TT;

    begin: f5
        Count5 = Count4(TT[31:16]) + Count4(TT[15:0]);
    end
endfunction

//***** The 1's count function - Count5 for 5-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count4 for 4-variable functions *****\
function [4:0] Count4;
    input [15:0] TT;

    begin: f4
        Count4 = Count3(TT[15:8]) + Count3(TT[7:0]);
    end
endfunction

//***** The 1's count function - Count4 for 4-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count3 for 3-variable functions *****\
function [3:0] Count3;

```

```

input [7:0] TT;

begin: f3
  Count3 = Count2(TT[7:4]) + Count2(TT[3:0]);
end
endfunction

//***** The 1's count function - Count3 for 3-variable functions *****\
//*****\
//*****\
//***** The 1's count function - Count2 for 2-variable functions *****\
function [2:0] Count2;
  input [3:0] TT;

  begin: f2
    Count2[0]=TT[3]^TT[2]^TT[1]^TT[0];

Count2[1]=(TT[3]&TT[2]|TT[3]&TT[1]|TT[3]&TT[0]|TT[2]&TT[1]|TT[2]&TT[0]|TT[1]&TT[0])&~(TT[3]&TT[2]&TT[1]&TT[0]);
    Count2[2]=TT[3]&TT[2]&TT[1]&TT[0];
  end
endfunction

//***** The 1's count function - Count2 for 2-variable functions *****\
//*****\

//////////////////////////////////// RESULTS //////////////////////////////////////

// n =          2          4          6          8          10
// Freq.         149.9      96.7      73.7      47.6      38.7
// #LUTs (%)     3(0%)     32(0%)   71(0%)  595(0%)  2296(3%)
endmodule
////////////////////////////////////
////////
////////////////////////////////////
////////

```

```

////////////////////////////////////
////////
////////////////////////////////////
////////

```

//////////////////////////////////// RESULTS //////////////////////////////////////

// n =	2	3	4	5	6		
//							
//nonlinearity	-	over	all functions/rot.	sym. func./symmetric	func.		
// 0	8/4/4	16/4/4	32/ 4/ 4	64/ 4/ 4	?	4/	4
// 1	8/4/4	128/8/8	512/ 8/ 8	2048/ 8/ 8	?	8/	8
// 2	0/0/0	112/4/4	3840/ 8/ 4	31744/ 4/ 4	?	8/	4
// 3	0/0/0	0/0/0	17920/ 8/ 0	317440/ 0/ 0	?	16/	0
// 4	0/0/0	0/0/0	28000/12/ 4	2301440/ 0/ 0	?	20/	0
// 5	0/0/0	0/0/0	14336/16/ 8	12888064/24/ 8	?	16/	0
// 6	0/0/0	0/0/0	896/ 8/ 4	57996288/48/ 16	?	56/	8
// 7	0/0/0	0/0/0	0/ 0/ 0	215414784/24/ 8	?	88/	16
// 8	0/0/0	0/0/0	0/ 0/ 0	647666880/ 0/ 0	?	80/	8
// 9	0/0/0	0/0/0	0/ 0/ 0	1362452480/ 0/ 0	?	152/	0
// 10	0/0/0	0/0/0	0/ 0/ 0	1412100096/36/ 4	?	184/	0
// 11	0/0/0	0/0/0	0/ 0/ 0	556408832/72/ 8	?	144/	0
// 12	0/0/0	0/0/0	0/ 0/ 0	27387136/36/ 4	?	324/	4
// 13	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	432/	8
// 14	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	360/	4
// 15	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	648/	8
// 16	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	832/	8
// 17	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	768/	0
// 18	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1076/	0
// 19	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1304/	0
// 20	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1232/	0
// 21	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1536/	16
// 22	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1924/	16
// 23	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	2232/	0
// 24	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	1612/	0
// 25	0/0/0	0/0/0	0/ 0/ 0	0/ 0/ 0	?	752/	0

```
// 26      0/0/0    0/0/0    0/ 0/ 0      0/ 0/ 0    ?/ 432/ 4
// 27      0/0/0    0/0/0    0/ 0/ 0      0/ 0/ 0    ?/ 96/ 8
// 28      0/0/0    0/0/0    0/ 0/ 0      0/ 0/ 0    ?/ 48/ 4
// 29      0/0/0    0/0/0    0/ 0/ 0      0/ 0/ 0    ?/ 0/ 0
// 30      0/0/0    0/0/0    0/ 0/ 0      0/ 0/ 0    ?/ 0/ 0
```

// Notes:

- // 1. Values for ALL functions for n = 6 were not obtained, since this computation takes more than 5000 years at 100 MHz..
- // 2. Values for ROT. SYM. functions for n = 7 were not obtained because, after 15 hours of compilation time, Synplify Pro issued an "Out-of-Memory" error message.
- // 3. Values for SYMMETRIC functions for n = 7 were not obtained because, after 15 hours of compilation time, Synplify Pro issued an "Out-of-Memory" error message

2. Circular Pipeline With Reservoir

Modules identical to those in the circular pipeline with IFGs (code in section 1) are not replicated in this section.

```
//-----  
// MY_CIRC_Pipe.v - An interface between the circular pipeline w/reservoir code that sets up  
//                  streaming with the SRC-6. Based on the SRC example user_one_stream.  
//  
// Created:      August 20, 2010  
// Last Modified: September 3, 2010  
// Author:       Chris Johnson  
//  
// Notes:       DATA_OUT bus width is not parameterized; must be manually edited for n>5.  
//                  modDATA_OUT must be edited for n>6.  
//  
// Sub-module calls: CircPipe.v  
//-----  
  
module MY_STREAM_TEST (  
    CNT,  
    START,  
    CLK,  
    CLR,  
    DONE,  
    DATA_OUT,  
    VALID_OUT,  
    STALL_IN,  
    TERM_OUT  
);  
input  [31:0] CNT;  
input  START;  
input  CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */;  
input  CLR;  
output DONE;
```

```

output [31:0] DATA_OUT;
output VALID_OUT;
input  STALL_IN;
output TERM_OUT;
    // output [N-n-1:0] COUNTER;

reg [31:0] DATA_OUT;
reg VALID_OUT;
reg TERM_OUT;
reg DONE;
reg [1:0] state;

parameter IDLE      = 0;
parameter ACTIVE   = 1;
parameter STALLED  = 2;
parameter FINISHING = 3;

wire      modDONE;
wire [63:0] modDATA_OUT;
wire      modVALID_OUT;
wire      modTERM_OUT;

always @*// (posedge CLK or posedge CLR)
    if (CLR) begin
        DATA_OUT  <= 0;
        DONE       <= 0;
        VALID_OUT  <= 0;
        TERM_OUT   <= 0;
        state      <= 0;
                //COUNTER    <= 0;
    end
    else
        case (state)
            IDLE:      if (START) begin
                DATA_OUT  <= 0;
                VALID_OUT  <= 1;
                // COUTNER <= 0;

```

```

        state    <= ACTIVE;
    end

ACTIVE:    begin
            DATA_OUT <= modDATA_OUT;
            DONE <= modDONE;
            VALID_OUT <= modVALID_OUT;
            TERM_OUT <= modDONE;

            if (STALL_IN) begin
                VALID_OUT <= 0;
                state    <= STALLED;
            end
        end

STALLED:   if (~STALL_IN) begin
            VALID_OUT <= 1;
            state    <= ACTIVE;
        end

FINISHING: begin
            //DONE <= 0;
            state <= IDLE;
        end
    default;;
endcase

CircPipe u2(START,CLK,CLR,modDONE,modDATA_OUT,modVALID_OUT,STALL_IN,modTERM_OUT);

endmodule

```

```

//-----
// CircPipe.v - The circular pipeline with independent function generators top level module.
//

```

```

// Created:      December 22, 2009
// Author:      Jon T. Butler
// Last Modified: September 3, 2010
// Modified by: Chris Johnson
//
// Notes:       Set parameter 'n' in this file.  It is passed to all sub-modules.
//
// Called by:   MY_CIRC_PIPE.v
//
// Sub-module calls:  countersMod.v
//                  Stage_TT.v
//
// This implements the circular pipeline.  For n-variable functions, there are N = 2**n stages,
// one for each linear function (we need only compare against the linear functions, since a
// function that has a bent distance from all linear function, has a bent distance away from
// all affine functions.  In this realization, only one stage has a bent function output - to
// simplify the circuit.  In this way, the circular pipeline serves as a buffer.  In this
// case a bent function will go through from N to 2N-1 stages.
//
//-----
//
module CircPipe #(parameter n=5, parameter N=2**n)
    (
        input          START,
        input          CLK,
        input          CLR,
        output         done,
        output [63:0]  BENT,
        output         valid_out,
        input          STALL_IN,
        output reg     term_out
    );

wire  [N-1:0]  LIN_FNC [N-1:0];
wire  [N-1:0]  REJECT; // 0 bit indicates FNCS word not accepted.
reg     temp;
wire  INHIBIT; // signal from the queue to pause counters
wire  [N-1:0]  FNCS [N-1:0]; // Each of the N words in counter FNCS has N bits.

```

```

wire [N*N-1:0]  FNCS_1d;           //for connection to queue module
wire [N*N-1:0]  QUEUE;           //output of reservoir queue
wire [n-1:0]    FNCSShob [N-1:0]; //high order bits for the counter
wire [N-n-1:0] counter;         //simple counter, extra bit to signal counter is done
wire [N-1:0]    to_stage;
wire [N-1:0]    stage_TT [N-1:0];
wire [n+1:0]    no_passes [N-1:0];

genvar g;

////////////////////////////////////

////////////////////////////////////FUNCTION GENERATOR////////////////////////////////////
//instantiate a single counter
countersMod #(.n(n)) u4(START,CLK,CLR,STALL_IN,INHIBIT,counter,done);

generate
//Generate high order bits
for (g=0; g<N; g=g+1)
    begin: CounterHOB
        assign FNCSShob[g] = g;
    end
endgenerate
//Generate function generators
generate
for (g=0; g<N; g=g+1)
    begin: CounterConcat
        assign FNCS[g] = {FNCSShob[g],counter[N-n-1:0]};
    end
endgenerate
//Create 1-d version of function generators for i/o interface
generate
for (g=0; g<N; g=g+1)
    begin: FNCS1d
        assign FNCS_1d[g*N+N-1:g*N] = FNCS[g];
    end
endgenerate

```

```

////////////////////////////////FUNCTION GENERATOR////////////////////////////////
////////////////////////////////LINEAR FUNCTIONS////////////////////////////////
generate
for (g=0; g<N; g=g+1)
    begin: LinearGen
        assign LIN_FNC[g] = Linear(g);
    end
endgenerate

function [N-1:0] Linear(input [n-1:0] Y);
    integer j;
    integer k;
    reg [n-1:0] X;
    begin
        for (j=0; j<N; j=j+1)
            begin
                X = j;
                temp=0;
                for (k=0; k<n; k=k+1)
                    begin
                        temp = temp ^ (X[k] & Y[k]);
                    end
                Linear[N-1-X] = temp;
            end
        end
    end
endfunction
////////////////////////////////LINEAR FUNCTIONS////////////////////////////////

////////////////////////////////RESERVOR/QUEUE////////////////////////////////
CircPipeQue #(.n(n)) QueModule(CLK, FNCS_1d, REJECT, INHIBIT, QUEUE);
////////////////////////////////RESERVOR/QUEUE////////////////////////////////

////////////////////////////////STAGES////////////////////////////////
generate
for (g = 0; g<N; g=g+1)

```

```

begin: Stages
  if(g != 0) begin
    stage #(.n(n)) u2(CLK, QUEUE[g*N+N-1:g*N], /*VALID_IN[g],*/ REJECT[g], to_stage[g-1],
to_stage[g], stage_TT[g-1], LIN_FNC[g], stage_TT[g], no_passes[g-1], no_passes[g]);
    end
    if(g == 0) begin
      stage1 #(.n(n)) u3(CLK, QUEUE[N-1:0], /*VALID_IN[0],*/ REJECT[0], to_stage[N-1],
to_stage[0], stage_TT[N-1], LIN_FNC[0], stage_TT[0], no_passes[N-1], no_passes[0], BENT, valid_out);
      end
    end
endgenerate
//////////STAGES//////////

endmodule

```

```

//-----
// CircPipeQue.v - Reservoir and queue for circular pipeline.
//
// Created:      March 30, 2010
// Author:      Chris Johnson
// Last Modified: September 3, 2010
//
// Notes:      None
//
// Called by:      CircPipe.v
//
// Sub-module calls:  pri_enc.v
//                  thermo_adder.v
//-----
//
module CircPipeQue      #(parameter n=3, parameter N=2**n)
  (
    input      CLK,
    input      [N*N-1:0]  gen_1,
    input      [N-1:0]    reject,
    output     inFromRes, //stall function generator
  )

```

```

        output reg [N*N-1:0] queue
    );

localparam SHAMT_WIDTH = n+1; //number of bits for shamt. n is enough to hold the max transfer distance

wire [N-1:0]          inToPipe;
reg [N-1:0]          in [N-1:0]; //Output of MUX that selects candidates for pipeline
wire [N*N-1:0]       in_1; // 1-d version of in
reg [N-1:0]          res [2*N-2:0]; //extra reg for pipelining
wire [N*(2*N-1)-1:0] reswire;
reg [SHAMT_WIDTH:0]  shamt [N:0]; //shift amount using to route TT's into "res"
wire [3*2**(2*n-1)-2**(n-1)-2:0] shamt_sel; //translate shamt into sel lines for use in pri_enc
//vector width is equivalent to sum(2^n,2^(n+1)-1)

wire [N-1:0]          out [2*N-2:0];
reg [N-1:0]          gen [N-1:0]; //2-D version of Func Gen inputs
reg [2*N-2:0]         occ; //occupied marker bits, one for each reservoir and "in" function
wire [n-1:0]         thermoSum;
reg [N-2:0]          thermo_occ; //occupied bits routed to thermoSum (either middle or lower 3 occ bits)

genvar i, j;

//Transform Func Gen's TT's to 2-D arrays
generate
for(i=0; i<N; i=i+1)
    begin: multidim
        always@* //(posedge CLK)//Pipeline function generator
            begin
                gen[i] <= gen_1[N*i+N-1:N*i];
            end
    end
endgenerate

always@* //MUX to select which source of functions to provide to CircPipe
    if(inFromRes)
        queue <= reswire[N*N-1:0];
    else
        queue <= gen_1;

```

```

//to output to the testbench
generate
for(i=0; i < 2*N-1; i=i+1)
    begin: ReswireOutput
        assign reswire[i*N+N-1:N*i] = res[i];
    end
endgenerate

//*****
//Create select lines from shamt
generate
for(i=0; i<N; i=i+1)
    begin: in1D
        assign in_1[i*N+N-1:i*N] = in[i];
    end
endgenerate

generate
for(i=0; i<2*N-2; i=i+1)
    begin: shamt_sel_gen
        if(i<N)
            begin
                for(j=0; j<N; j=j+1)
                    begin: sham_sel_gen_inner1
                        assign shamt_sel[i*N+j] = (shamt[N-j]==i && !inToPipe[j]) ? 1'b1 : 1'b0;
                    end
                end
            end

            else// if(i<2*N-2)
                begin
                    for(j=0; j<2*N-1-i; j=j+1)
                        begin: shamt_sel_gen_inner2
                            assign shamt_sel[shamt_idx(i)+j] = (shamt[2*N-i-1-j]==i && !inToPipe[j+i-N+1]) ?
1'b1 : 1'b0;
                        end
                    end
                end
            end
end

```

```

        end
    endgenerate
//*****
//SECTION ONE: INPUT CONTROL AND RESERVOIR
////////////////////////////////////
assign inFromRes = occ[N-1];

//Select input from either func gen or reservoir
generate
for(i=0; i<N; i=i+1)
    begin: incoming
        always@*//(inFromRes, res[i], gen[i])
        begin: A
            in[i] <= inFromRes ? res[i] : gen[i];
        end
    end
endgenerate

//Calculate shamt from reservoir
always@* thermo_occ <= inFromRes ? occ[2*N-2:N] : occ[N-2:0];
thermo_adder #(n) thermo(thermo_occ,thermoSum);
always@* shamt[N] <= thermoSum;

//Calculate shamt for each incoming function T from the MUX
generate
for(i=0; i<N; i=i+1)
    begin:shiftCalc
        always@*//(shamt[i+1], inToPipe[N-1-i])
        begin: shamt_setup
            shamt[i] = shamt[i+1] + !inToPipe[N-1-i];
        end
    end
endgenerate

//set occ bits based on res contents
generate
for(i=0; i<2*N-1; i=i+1)

```

```

begin: occ_connect
    always@*
        if(res[i]) occ[i] <= 1'b1;
        else occ[i] <= 1'b0;
    end
endgenerate

//Assign to resTemp (wires to the reservoir registers) the proper input, based on xfer table & inToPipe
// Accomplished through use of priority encoders
generate
    for(i=0; i<2*N-2; i=i+1)
        begin: Cases
            if(i<N)
                begin
                    pri_enc #(.n(n),.s(N)) pi_1(in_1,shamt_sel[i*N+N-2:i*N],inToPipe[N-1:0],out[i]); //for i
                    >= N, pri_enc doesn't need entire 'in_1', so pruning will occur, shamt's are each 5 bits
                end
            else//(i<2*N-2)
                begin
                    pri_enc #(.n(n),.s(2*N-1-i)) pi_2(in_1[N*N-1:(i-N+1)*N],shamt_sel[shamt_idx(i+1)-
                    1:shamt_idx(i)],inToPipe[N-1:i+1-N],out[i]); //parring should occur
                end
            end
        end
    endgenerate

//Constant function to generate indicies of shamt_1 in the generate elseif(i<2*N-2) section of pri_enc calls
function integer shamt_idx(input integer index);
    integer k;
    integer j;
    integer test; //added for XST
    begin
        k=1;
        shamt_idx=N*N;
        for(j=index; N<j; j=j-1)
            begin
                shamt_idx = shamt_idx + N - k;
                k=k+1;
            end
        end
endfunction

```

```

                end
            end
endfunction
/*
generate
    for(i=0; i<2*N-1; i=i+1)
        begin: Pipelres
            always@(posedge CLK)
                res[i] <= res_0p[i];
        end
    endgenerate
*/

generate
    for(i=0; i<2*N-1; i=i+1)
        begin: reservoir
            if(i<N-1) begin
                always@(posedge CLK) res[i]/*res_0p[i]*/ = low_res(inFromRes,shamt_sel[i*N+N-
1:i*N],shamt[i],out[i],res[N+i],res[i]);
            end
            else if (i==N-1) begin
                always@(posedge CLK) res[i]/*res_0p[i]*/ = low_res(inFromRes,shamt_sel[i*N+N-
1:i*N],shamt[i],out[i],{N{1'b0}},res[i]);
            end
            else if (i<2*N-2)begin //(N-1 < i < 2*N-2)
                always@(posedge CLK) res[i]/*res_0p[i]*/ = mid_res(inFromRes,{i-
N+1{shamt_sel[shamt_idx(i+1)-1:shamt_idx(i)]}},out[i],res[i]);
            end
            else begin//i==2*N-2
                always@(posedge CLK) res[i]/*res_0p[2*N-2]*/ = if_func_Nbit(in[N-1],inToPipe[N-
1],inFromRes,shamt[1],res[2*N-2]);//probably don't need this, just control occ bit and always assign in[N-1]
to reswire [N-1]
            end
        end
    endgenerate

function [N-1:0] low_res (    input inFromRes,

```

```

        input [N-1:0] sel,
        input [N-1:0] shamt_i, //may not be needed if out is already zeros
        input [N-1:0] out,
        input [N-1:0] mid_res,
        input [N-1:0] res
    );
begin
    if(inFromRes && mid_res) begin //slide middle registers down
        low_res = mid_res;
    end
    else if(sel && out) begin //if sel and outwite are not zero
        low_res = out;
    end
    else low_res = inFromRes ? {N{1'b0}} : res;
end
endfunction

function [N-1:0] mid_res (    input inFromRes,
                            input [N-1:0] sel, //couldn't figure out how to taper this width
                            //input [N-1:0] shamt_i,
                            input [N-1:0] out,
                            input [N-1:0] res);

begin
    if(sel && out) begin //if sel and outwite are not zero
        mid_res = out;
    end
    else mid_res = {N{1'b0}};
end
endfunction

//This is a NOT-IF
function [N-1:0] if_func_Nbit(    input [N-1:0] in,
                                input inToPipe,
                                input inFromRes,
                                input [SHAMT_WIDTH:0] shamt_i,
                                input prior_value
                                );

```

```

begin
    if((3>=shamt_i) && inFromRes)
    begin
        if_func_Nbit = {N{1'b0}};
    end
    else if((shamt_i==2*N-2) && !inToPipe)
    begin
        if_func_Nbit = in;
    end
    else
        if_func_Nbit = prior_value;
    end
endfunction
endmodule

```

```

module pri_enc #(parameter n=2,s=4) (in, sel, inToPipe, out);
//-----
// pri_enc - Verilog code to implement a priority encoder depending on a parameters, n and m.
//
//
// Created:      March 15, 2010
// Last Modified: July 21, 2010
// Author:      Chris Johnson
//              Adapted from J.T. Butler's 1-bit priority encoder, modified for
//              for busses and select lines in the Circular Pipeline Reservoir.
//
// Notes:       None.
//
// Called by:   CircPipeQue.v
//
// Sub-module calls:  sel_module.v
//                iff.v
//-----
parameter N = 2**n;
//s is number of TT's being input (all MUX's get one TT, except the last one generated gets 2)

```

```

localparam SHAMT_WIDTH = n+1; //number of bits for shamt. n is large enough to hold the max transfer
distance

input  [s*N-1:0]  in;          // in has up to N*N bits; all the applicable incoming functions
input  [s-2:0]    sel;        // sel determines which OUT. Up to N-1 bits.
input  [s-1:0]    inToPipe;  // signal indicating slot in circ pipe is vacant
output [N-1:0]    out ;       // OUT is main output of circuit.

wire  [s*N-1:0]  inC;
wire  [(INNER_S(s)-3)*N+N-1:0]      inner;      // inner is a line interconnecting

genvar i;

//Constant function to provide INNER_S index
function integer INNER_S(input integer s);
    begin
        if(s>2)
            INNER_S = s;
        else
            INNER_S = 3;
        end
    endfunction

//Bring TT in if it's rejected from the circular pipeline, else don't bring it in.
generate
    for(i=0; i<s; i=i+1)
        begin: ifinToPipe
            iff #(.N(N)) u5 (in[i*N+N-1:i*N],inToPipe[i],inC[i*N+N-1:i*N]);
        end
endgenerate

//    Within the generate for loop below, if statements handle (3) special interconnection
//    requirements, beginning, end, and middle.
generate
    for (i=0; i<s-1; i=i+1)
        begin:stage

```

```

        if (s == 2)
            assign inner[N-1:0] = inC[s*N-1:s*N-N];
            if (i == 0)
                sel_module #(.N(N)) u1 (inner[N-1:0],          inC[N-1:0],
sel[i],    out);
            else if (i == (s-2))
                sel_module #(.N(N)) u2 (inC[s*N-1:s*N-N],      inC[s*N-N-1:s*N-2*N],  sel[s-2],
inner[(i-1)*N+N-1:(i-1)*N]); //in case of s=2, input 2 (inC) is repeated from MUX_0
            else
                sel_module #(.N(N)) u3 (inner[i*N+N-1:i*N],    inC[i*N+N-1:i*N],      sel[i],
inner[(i-1)*N+N-1:(i-1)*N]);
        end
    endgenerate

endmodule

```

```

//-----
// sel_module - Selector module. Basically, a MUX.
//
//
// Created:      March 30, 2010
// Last Modified: July 21, 2010
// Author:       Chris Johnson
//
//
// Notes:        None.
//
// Called by:    pri_enc.v
//
// Sub-module calls:  None.
//-----
//
module sel_module #(parameter N=4) (sel_0, sel_1, sel, out);

input    [N-1:0] sel_0;

```

```
input    [N-1:0]  sel_1;
input    sel;
output   [N-1:0]  out;
reg      [N-1:0]  out;
```

```
always @*
    begin
        if (sel == 1) out <= sel_1;
        else out <= sel_0;
    end
endmodule
```

```
//-----
// iff - Simply and if statement, used for calls within a generate statement
//
// Created:      March 30, 2010
// Last Modified: July 21, 2010
// Author:       Chris Johnson
//
//
// Notes:        None.
//
// Called by:    iff.v
//
// Sub-module calls:  None.
//
//-----
//
```

```
module iff #(parameter N=4) (in,inToPipe,out);
```

```
input    [N-1:0]    in;
input    inToPipe;
output   [N-1:0]    out;

reg      [N-1:0]    out;
```

```
always@*
```

```

begin
    if(!inToPipe)
        out <= in;
    else
        out <= {N{1'b0}};
end

endmodule

```

```

module thermo_adder #(parameter n = 2) (occup, sum);
//-----
// thermo_adder - Verilog code to compute the sum of a 2^n bit input, occupp.
//               occupp is the set of bits from the stages in the reservoir
//               that indicate whether the stage is occuppied (1) or not (0).
//               The bits from occupp is a thermometer. So, if occupp(i) = 1,
//               then occupp(j) = 1 for all j < i. This results in a simpler
//               circuit.
//
// Created:      January 31, 2010
// Last Modified: 21 July 2010
// Author:      Jon T. Butler
// Modified:    Chris Johnson
//
// Called by:    CircPipeQue.v
//
// Sub-module calls:  None.
//-----
//
localparam N = 2**n;

input  [N-2:0]      occup;          // occupp has 2^n bits.
output reg [n-1:0] sum;            // sum is an n-bit number indicating how many input bits are 1.

```

```

integer index, g;

always @*
  if(occup[N-2] == 1'b1)
    sum[n-1:0] = {{n{1'b1}}};
  else
    begin
      sum[n-1] = 1'b0;
      index = 2**(n-1)-1;
      for (g=n-1; g>=0; g = g-1)
        begin
          if(occup[index] == 1'b1)
            begin
              sum[g] = 1;
              index = index + 2**(g-1);
            end
          else
            begin
              sum[g] = 0;
              index = index - 2**(g-1);
            end
        end
      end
    end
end

endmodule

```

B. SRC-6 IMPLEMENTATION FILES

1. main.c

```
////////////////////////////////////
/*                                                                    */
/* main.c - C program to test an SRC-6E implementation of min.v      */
/*                                                                    */
/* Author: Chris Johnson                                             */
/* Created: August 1, 2010                                           */
/* Last modified: September 3, 2010                                  */
/*                                                                    */
/* Description: This program searches for bent functions using the   */
/*              circular pipeline with IFGs                          */
/*                                                                    */
/*                                                                    */
/*****/

#include <map.h>
#include <stdlib.h>
#include <string.h>

void subr (int64_t*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*, int8_t*, int64_t*, int);

int main () {

    int i,j,mapnum=0;
    int64_t time_clk, r1, r2, cmin[32], invalc;
    int64_t *in0, *in1, *in2, *in3, *BENT, *REJECT, *STAGE_TT_out;
    int8_t *valid_out;

    /* Allocate array of x values, in, and array of function values, out */
    in0 = (int64_t *) malloc (4096* sizeof (int64_t));
    in1 = (int64_t *) malloc (4096* sizeof (int64_t));
```

```

in2 = (int64_t *) malloc (4096* sizeof (int64_t));
in3 = (int64_t *) malloc (4096* sizeof (int64_t));
BENT = (int64_t *) malloc (4096* sizeof (int64_t));
STAGE_TT_out = (int64_t *) malloc (4096* sizeof (int64_t));

    for (i = 0; i < 4096; i++){
        in0[i] = 12816;//3210
        in1[i] = 30292;//7654
        in2[i] = 47768;//AB98
        in3[i] = 65244;//FEDC
        out[i] = 0;
    }

map_allocate (1);

// Call subroutine subr.mc on the MAP.
subr (in0, in1, in2, in3, &time_clk, REJECT, BENT, valid_out, STAGE_TT_out, mapnum);

/* Print out the number of clocks.                                     */
printf ("%lld clocks\n", time_clk);

/* Print out the output.                                           */
    for (i=0; i<4096; i++){
        printf("BENT: %x \n",BENT[i]);
        if(out[i])
            printf("PartialStageTT: %x \n",out[i]);
    }

map_free (1);

exit(0);

}

```

2. subr.mc

```

/*****
/*
/* subr.mc - MAP C subroutine to cue TT's for circular pipeline.
/*
/* Author: Chris Johnson
/* Created: June 14, 2010
/* Last modified: September 3, 2010
/*
/* Description: This program calls an SRC-6 macro that seives
/* functions through a circular pipeline.
/*
/*
/*
/*
/*****

#include <libmap.h>

void subr (int64_t in0[], int64_t in1[], int64_t in2[], int64_t in3[], int64_t *time, int64_t reject[],
int64_t bent[], int8_t valid_out, int64_t tt[], int mapnum) {

// Declare one OBM banks in SRC-6 to store...
    OBM_BANK_A (IN0, int64_t, 1024)
    OBM_BANK_B (BENT_o, int64_t, 4096)
    OBM_BANK_C (IN1, int64_t, 1024)
    OBM_BANK_D (IN2, int64_t, 1024)
    OBM_BANK_E (IN3, int64_t, 1024)
    OBM_BANK_F (TT_o, int64_t, 4096)

    int64_t my64bit_in0, my64bit_in1, my64bit_in2, my64bit_in3, REJECT, BENT, stage_TT_out, t0, t1;
    int8_t VALID_OUT; //only need 1 bit
    int i;

// Get values by DMAing FROM the CPU
    DMA_CPU (CM2OBM, IN0, MAP_OBM_stripe(1,"A"), in0, 1, 1024*sizeof(int64_t), 0);
    DMA_CPU (CM2OBM, IN1, MAP_OBM_stripe(1,"C"), in1, 1, 1024*sizeof(int64_t), 0);

```

```

DMA_CPU (CM2OBM, IN2, MAP_OBM_stripe(1,"D"), in2, 1, 1024*sizeof(int64_t), 0);
DMA_CPU (CM2OBM, IN3, MAP_OBM_stripe(1,"E"), in3, 1, 1024*sizeof(int64_t), 0);
wait_DMA (0);

read_timer(&t0);

for (i = 0; i < 1024; i++){
// The my_operator macro call has 2 inputs, IN and INTOPIPE, and one output, OUT
my64bit_in0 = IN0[i];
my64bit_in1 = IN1[i];
my64bit_in2 = IN2[i];
my64bit_in3 = IN3[i];
my_operator (my64bit_in0, my64bit_in1, my64bit_in2, my64bit_in3, REJECT, BENT, VALID_OUT,
stage_TT_out);
BENT_o[i] = BENT;
TT_o[i] = stage_TT_out;
}

read_timer(&t1);

*time = (t1 - t0);

// Return values by DMAing TO the CPU
DMA_CPU (OBM2CM, BENT_o, MAP_OBM_stripe(1,"B"), bent, 1, 4096*sizeof(int64_t), 0);
DMA_CPU (OBM2CM, TT_o, MAP_OBM_stripe(1,"F"), tt, 1, 4096*sizeof(int64_t), 0);
wait_DMA (0);
}

```

3. makefile

```

# $Id: Makefile.template,v 1.13 2005/04/12 19:18:30 jls Exp $
#
# Copyright 2003 SRC Computers, Inc. All Rights Reserved.

```

```

#
#       Manufactured in the United States of America.
#
# SRC Computers, Inc.
# 4240 N Nevada Avenue
# Colorado Springs, CO 80907
# (v) (719) 262-0213
# (f) (719) 262-0223
#
# No permission has been granted to distribute this software
# without the express permission of SRC Computers, Inc.
#
# This program is distributed WITHOUT ANY WARRANTY OF ANY KIND.
#
# -----
# -----
# User defines FILES, MAPFILES, and BIN here
# -----
FILES           = main.c

MAPFILES       = subr.mc

BIN            = main

# -----
# Multi chip info provided here
# (Leave commented out if not used)
# -----
#PRIMARY       = <primary file 1>  <primary file 2>

#SECONDARY     = <secondary file 1> <secondary file 2>

#CHIP2        = <file to compile to user chip 2>

#-----
# User defined directory of code routines

```

```

# that are to be inlined
#-----

#INLINEDIR      =

# -----
# User defined macros info supplied here
#
# (Leave commented out if not used)
# -----
MACROS          = my_macro/CircPipe.v
MY_BLKBOX       = my_macro/blk.v
MY_NGO_DIR      = my_macro
MY_INFO         = my_macro/info
# -----
# Floating point macros selection
# -----

#FPMODE         = SRC_IEEE_V1 # Default SRC version IEEE
#FPMODE         = SRC_IEEE_V2 # Size reduced SRC IEEE with
#               # special rounding mode

# -----
# User supplied MCC and MFTN flags
# -----

MCCFLAGS        = -v
MFTNFLAGS       = -v

# -----
# User supplied flags for C & Fortran compilers
# -----

CC              = gcc      # gcc      for Intel cc for Gnu
FC              = ifort    # ifort    for Intel f77 for Gnu
#LD            = ifort    -nofor_main # for mixed C and Fortran, main in C
#LD            = ifort    # for Fortran or C/Fortran mixed, main in Fortran
LD              = gcc      # for C codes

```

```

MY_CFLAGS      =
MY_FFLAGS      =
MY_LDFLAGS     =          # Flags to include libs if needed
# -----
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# -----

#USEVCS        = yes      # YES or yes to use vcs instead of vcsi
#VCSDUMP       = yes      # YES or yes to generate vcd+ trace dump
# -----
# MODELSIM simulation settings
# (Set as needed, otherwise just leave commented out)
# -----

#USEMDL        = yes      # YES or yes to use modelsim instead of vcs/vcsi
#USEMDLGUI     = yes      # YES or yes to use modelsim GUI interface
#MDLDUMP       = yes      # YES or yes to generate vcd trace dump
# -----
# No modifications are required below
# -----
MAKIN    ?= $(MC_ROOT)/opt/src/cci/comp/lib/AppRules.make
include $(MAKIN)

```

4. info

```

/*****/
/**
/**  info - info file to specify the input and output of macro CircPipeCue */
/**
/**   Author:      Chris Johnson      */
/**   Created:     August 2, 2010     */
/**   Last modified: September 3, 2010 */
/**

```

```

/**
/*****
BEGIN_DEF "my_operator"          //Name used in .mc file to call macro.
    MACRO = "CircPipe";         //Macro name.
    STATEFUL = NO;
    EXTERNAL = NO;
    PIPELINED = YES;
    LATENCY = 0;

    INPUTS = 4:
        I0 = INT 64 BITS        (FNCS0[64:0])
        I1 = INT 64 BITS        (FNCS1[64:0])
        I2 = INT 64 BITS        (FNCS2[64:0])
        I3 = INT 64 BITS        (FNCS3[64:0])
        ;

    OUTPUTS = 4:
        O0 = INT 64 BITS        (REJECT[63:0])
        O1 = INT 64 BITS        (BENT[63:0])
        O2 = INT 8  BITS        (valid_out[7:0]) //only need 1 bit
        O3 = INT 64 BITS        (STAGE_TT_out[63:0])
        ;

    IN_SIGNAL: 1 BITS "CLK" = "CLOCK";
END_DEF

```

5. blk.v

```

/*****
/**
/** blk.v - black-box file that specifies input and output
/**

```

```

/*      Author:      Chris Johnson      */
/*      Created:     August 1, 2010     */
/*      Last modified: September 3, 2010      */
/*      */
/*****/

module CircPipe (CLK, FNCS0, FNCS1, FNCS2, FNCS3, REJECT, BENT, valid_out, STAGE_TT_out);
    input CLK;
    input [63:0] FNCS0;
    input [63:0] FNCS1;
    input [63:0] FNCS2;
    input [63:0] FNCS3;
    output [63:0] REJECT;
    output [7:0] valid_out;
    output [63:0] stage_TT_out;
    output [63:0] BENT;
endmodule

```

LIST OF REFERENCES

- [1] M. Matsui, “Linear cryptanalysis method for DES cipher,” in *Advances in Cryptology – EUROCRYPT*, pp. 386–397, 1993.
- [2] J. L. Shafer, S. W. Schneider, J. T. Butler, and P. Stanica, “Enumeration of bent Boolean functions by reconfigurable computer,” in *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 265–272, 2010.
- [3] T. W. Cusick and Pantelimon Stanica, *Cryptographic Boolean Functions and Applications*. San Diego: Elsevier, 2009.
- [4] J. T. Butler, “On the speedup of sieves in the circular pipeline,” October 2009 Preprint.
- [5] “Introduction to EC3820 and its Laboratory,” class notes for EC3820, Department of Electrical and Computer Engineering, Naval Postgraduate School, Fall 2010.
- [6] SRC Computers, Inc., “SRC Carte™ C Programming Environment v3.2 Guide,” SRC–007–20, Colorado Springs, Colorado, November 2009.
- [7] J. Hammes (private communication), August 2010.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Dr. Clark Robertson
Naval Postgraduate School
Monterey, California
4. Dr. John G. Harkins
National Security Agency
Fort Meade, Maryland
5. Dr. David R. Podany
National Security Agency
Fort Meade, Maryland
6. Mr. David Caliga
SRC Computers
Colorado Springs, Colorado
7. Mr. Jon Huppenthal
SRC Computers
Colorado Springs, Colorado
8. Dr. Jeff Hammes
SRC Computers
Colorado Springs, Colorado
9. Dr. Jon T. Butler
Naval Postgraduate School
Monterey, California
10. Dr. Pantelimon Stanica
Naval Postgraduate School
Monterey, California

11. Dr. Robert L. Herklotz
Program Manager, Information Operations and Security
Air Force Office of Scientific Research (AFOSR/RSL)
Arlington, Virginia
12. J. L. Shafer
US Naval Academy
Annapolis, Maryland
13. S. W. Schneider
Naval Postgraduate School
Monterey, California
14. N. B. Schafer
Naval Postgraduate School
Monterey, California