THE UNIVERSITY OF CALGARY

Self-Timed Bit-Serial Architectures for Digital Signal Processing

by

Jianchuan Li

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

CALGARY, ALBERTA

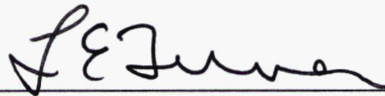
September, 2005

© Jianchuan Li 2005

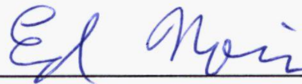
THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

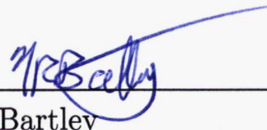
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Self-Timed Bit-Serial Architectures for Digital Signal Processing" submitted by Jianchuan Li in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.



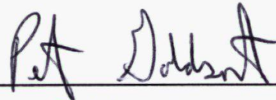
Supervisor, Dr. Laurence Turner
Department of Electrical and Computer Engineering



Dr. Ed Nowicki
Department of Electrical and Computer Engineering



Mr. Norm Bartley
Department of Electrical and Computer Engineering



Dr. Peter Goldsmith
Department of Mechanical and Manufacturing Engineering

September 14, 2005

Date

Abstract

Bit-serial systems can be attractive alternatives to bit-parallel systems in the signal processing domain, due to the inherent small size, local connectivity, and constant routing density. However, static-timed bit-serial systems are perceived to be difficult to design, as detailed knowledge of bit-serial arithmetic and timing is required, and static delays needs to be inserted to synchronize the data bit streams. This thesis explores a new concept in bit-serial systems, called self-timed bit-serial. Instead of using static delays to adjust timing, a self-timed bit-serial system adapts to its system structure dynamically during run-time. Advantages of such a system include shortened design time, variable word length, and the possibility of run-time reconfiguration. Two applications (floating point addition and speech synthesis) have been implemented and tested on a Xilinx VirtexE FPGA using a self-timed bit-serial architecture.

Acknowledgments

I would like to take this opportunity to thank all those who made this possible for me. First of all, I thank my parents for their support over these long, long, years for me to get to this level of education. Without their emotional support and insightful advices on life I would have never made it this far. I also thank Dr. Turner for his help in my academic pursuit over the past two years. Last but not least, I would like to thank all my friends who have been with me for the past seven years and helped me to retain my sanity.

Table of Contents

Approval Page ii

Abstract iii

Acknowledgments iv

Table of Contents v

List of Tables viii

List of Figures x

List of Symbols and Acronyms xi

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Static-Timing vs Self-Timing . . . 2
- 1.2 Objective . . . 3
- 1.3 Thesis Outline . . . 3

2 Bit-Serial Systems Background 6
- 2.1 Static-Timed Bit-Serial Architecture . . . 7
  - 2.1.1 Data Frame Signal . . . 7
  - 2.1.2 Two's Complement Number Representation . . . 8
  - 2.1.3 Latency and Scheduling . . . 10
- 2.2 Basic Processing Elements and Their Structures . . . 13
  - 2.2.1 Adder/Subtractor . . . 13
  - 2.2.2 Shift Left Operator . . . 15
  - 2.2.3 Shift Right Operator . . . 16
  - 2.2.4 Multipliers . . . 17
  - 2.2.5 Delay Element . . . 21
- 2.3 Temporal Spatial Trade-Offs With Digit-Serial . . . 22

3 Self-Timed Bit-Serial Architectures 24
- 3.1 Benefits of a Self-Timed Bit-Serial Architecture . . . 25
- 3.2 General Requirements for Self-Timing . . . 26
- 3.3 Architecture One: Controller Regulated Architecture . . . 28
  - 3.3.1 Data Format and Control Signals . . . 28

3.3.2 Architecture Overview 28
3.3.3 Adder/Subtractor 30
3.3.4 Serial by Parallel Multiplier with Parallel Load 33
3.3.5 Delay Element 34
3.3.6 Fork Element 37
3.3.7 Summary of the Controller Regulated Architecture 39
3.4 Architecture Two: Full Word Buffer Architecture 40
3.4.1 Data Format and Control Signals 40
3.4.2 Architecture Overview 40
3.4.3 Adder/Subtractor 42
3.4.4 Multiplier 43
3.4.5 Delay Element 45
3.4.6 Fork Element 46
3.4.7 Summary of the Full Word Buffer Architecture 47
3.5 Architecture Three: 2-Bit Buffer Architecture 49
3.5.1 Data Format and Control Signals 49
3.5.2 Architecture Overview 50
3.5.3 Two-Input Elements 51
3.5.4 Multiplexer and 3-Input Elements 56
3.5.5 Fork/Delay Elements 59
3.5.6 Summary of the 2-Bit Buffer Architecture 59

4 Floating Point Application **62**
4.1 Floating Point Background 64
4.1.1 Floating Point Number Format and Addition Algorithm . . . 64
4.1.2 Bit-Parallel Floating-Point Units 66
4.2 Self-timed Bit-serial Floating Point Adder 68
4.2.1 Right Shift Operator 68
4.2.2 Adder Structure 69
4.3 Adder Performance Comparison 71
4.4 Self-Timed Bit-Serial Floating Point Unit Summary 73

5 Speech Synthesizing Vocoder Application **75**
5.1 Rsynth Text to Speech Vocoder 75
5.1.1 Advantages in Bit-Serial Implementation 76
5.1.2 Vocoder Structure 76
5.2 Self-Timed Bit-Serial Implementation 78
5.2.1 Finite Precision Implementation 79
5.2.2 Vocoder Implementation 80
5.2.3 Coefficient Storage 81

5.2.4 Audio Codec 83
5.3 Self-Timed Bit-Serial Vocoder Summary 85

6 Conclusion and Future Work **87**
6.1 Thesis Summary 87
6.2 Future Work 89
6.2.1 Architectural Improvements 89
6.2.2 Run-Time Reconfiguration 89
6.2.3 Self-Timed Bit-Serial Compiler 90

Bibliography **91**

List of Tables

3.1 Size and Speed Estimation of Processing Elements in Architecture One. 39
3.2 Size and Speed Estimation of Processing Elements in Architecture Two. 48
3.3 Size and Speed Estimation of Processing Elements in Architecture Three. 61

4.1 Comparison of Single Precision Floating Point Adders. 73

5.1 Self-Timed Bit-Serial Vocoder Summary. 86

List of Figures

2.1 Data Frame Signal for Static-Timed Bit-serial Architecture 8
2.2 Multiple Input Channels in a Static-Timed Bit-serial Architecture . . 12
2.3 First Order IIR Filter Implementation in a Static-Timed Bit-serial Architecture 12
2.4 Bit-Serial Adder Structure 14
2.5 Bit-Serial Subtractor Structure 14
2.6 Bit-Serial Adder Timing Diagram 15
2.7 Bit-serial Shift Left Operator Structure 16
2.8 Bit-serial Shift Left Operator Timing Diagram 16
2.9 Bit-serial Shift Right Operator Structure 17
2.10 Bit-serial Shift Right Operator Timing Diagram 17
2.11 Carry Ripple Multiplication Table 18
2.12 Bit-serial Constant Coefficient Serial by Parallel Multiplier Structure 19
2.13 Constant Word Length Serial by Parallel Multiplier Structure 21
2.14 Bit-serial Delay Structure 21
2.15 2-Bit Digit-Serial Adder Structure 23

3.1 Architecture One Data Format and Control Signals 29
3.2 Single Bit Input Buffer for Architecture One 30
3.3 Two-Input Element Structure for Architecture One 31
3.4 Adder/Subtractor State Machine Controller Structure for Architecture One 32
3.5 Multiplier State Machine Controller Structure for Architecture One . 34
3.6 Delay Element Concept for Architecture One 35
3.7 State Machine for *ready_receive* Signal Generation 36
3.8 State Machine Controller for the Delay Element 37
3.9 Fork Element Structure 38
3.10 Two-Input Element Structure for Architecture Two 41
3.11 Two-Input Element Central Controller for Architecture Two 42
3.12 Input Buffer for Architecture Two 43
3.13 Adder Unit for Architecture Two 44
3.14 Processing Unit of the Multiplier for Architecture Two 45
3.15 Delay Element for Architecture Two 46
3.16 Fork Element for Architecture Two 47
3.17 Architecture Three Data Format and Control Signals 50
3.18 DFF Implementation of the 2-Bit Input Buffer 51
3.19 SRL16 Implementation of the 2-Bit Input Buffer 52

3.20 General Structure of a Two-Input Processing Element in Architecture Three. 53
3.21 State Machine Controller for Two-Input Processing Element in Architecture Three. 54
3.22 Serial by Serial Constant Word Length Multiplier Structure. 55
3.23 Control Structure Example. 56
3.24 Structure of the State Machine Controller for 3-Input Processing Elements in Architecture Three. 57
3.25 Structure of the Multiplexer in Architecture Three. 58
3.26 Structure of the Fork/Delay Element in Architecture Three. 60

4.1 Typical Floating-Point Number Format 64
4.2 Structure of a Shift Right Operator 69
4.3 Structure of the Mantissa Alignment Shift. 70
4.4 Self-Timed Bit-Serial Floating Point Adder Structure and Latency. . 72

5.1 Structure of the Vocoder. 77
5.2 Structure of the Vocoder System. 78
5.3 Coefficient Quantization Example. 81
5.4 Resonator Structure. 82
5.5 Anti-Resonator Structure. 82
5.6 1st Order Differentiator Structure. 82
5.7 Coefficient Memory Interface to the Vocoder. 84
5.8 Audio Codec Interface. 85

List of Symbols and Acronyms

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
IO	Input Output
LSB	Least Significant Bit
LUT	Look Up Table
MSB	Most Significant Bit
SWL	System Word Length

Chapter 1

Introduction

1.1 Motivation

With recent advancements in the density and speed of integrated circuits and digital hardware, digital signal processing is fast becoming an important field. Different signal processing architectures have evolved since the 1960s for digital signal processing implementations [1]. At the system level, different architectures balance size and speed by the level of parallelism incorporated in the design. On one end of the spectrum, data is processed sequentially by time-sharing components to save hardware resources, such as the case in a general purpose digital signal processor. On the other end of the spectrum, data is processed in parallel to increase throughput, and requires a full implementation of the system on an Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA). The same compromise between size and speed can also be applied at the bit level. Systems where each bit of a word is processed sequentially are referred to as bit-serial systems, as opposed to bit-parallel systems, where every bit of a word is processed at the same time.

By the very nature of a bit-serial system, its throughput is System Word Length (SWL) times slower when compared to a fully pipelined bit-parallel system operating at the same clock frequency. However, the hardware resources required for implementation is significantly less, because only enough hardware to process one bit at a time is needed. A pipelined bit-parallel system with an *n-bit* system word length would be

at least n times larger than a bit-serial implementation, plus the cost in registers for pipelining. This small size characteristic makes a bit-serial architecture attractive for applications where the available technology is faster than the required throughput and the designer is looking to save hardware resources. However, bit-serial systems may be perceived as being difficult to design, because synchronization between different serial bit streams is required, and timing control can be complex when the system is large. Other restrictions such as constant word length and the inability to re-adjust timing during run-time also limits the usefulness of the static-timed bit-serial architecture (defined in Section 1.1.1). In a bit-parallel system, variable word length can be achieved because word length can be manipulated in space and timing is not an issue. Changing the word length in a bit-serial system would require manipulation of the data signal in time, and because the timing is static, this is not possible.

1.1.1 Static-Timing vs Self-Timing

A static-timed bit-serial system solves the data stream synchronization problems by inserting static delays on data channels until the correct timing is achieved. A new concept in bit-serial systems, called self-timed bit-serial, solves the synchronization problem differently. Instead of using static delays, which are fixed at design time, the timing is self adjusted during run-time using handshaking signals between each processing element. This idea is similar in concept to asynchronous systems where data and timing information are sent together. Asynchronous systems are built with the complete absence of a global clock. However, because a truly asynchronous system cannot be implemented on any commercial FPGA available at this time, the

self-timed bit-serial system still requires a global clock to synchronize each data bit.

1.2 Objective

The objective of this research is to design, implement, and test self-timed bit-serial architectures for digital signal processing applications. These architectures should allow flexible design approaches such as variable word length and dynamic timing adaptation during run-time. System implementation should be made easy on these architectures, so that detailed knowledge of bit-serial arithmetic and timing are not required by designers.

The focus of this thesis is to demonstrate that such architectures are possible and beneficial to designers. This is accomplished by designing self-timed bit-serial architectures and evaluating their performance in suitable applications. Specifically, three self-timed bit-serial architectures are developed, and two sample applications, floating point addition and speech vocoder, are used to evaluate their performance. The important performance metrics are: hardware size, throughput, and the ease of system implementation.

The future direction of this research is to explore the possibility of further reducing hardware usage by dynamically changing data paths during run-time and time-sharing system components. The dynamic timing characteristic of these new architectures makes this hardware saving approach possible.

1.3 Thesis Outline

This thesis consists of six chapters. The contents of chapter two to six are as follows.

- Chapter Two presents an introduction to bit-serial arithmetic. The first part of this chapter provides an architectural overview of static-timed bit-serial arithmetic systems. Details such as the number format and arithmetic used and timing considerations are discussed. The second part of this chapter provides an overview of the processing elements used in a bit-serial signal processing system. These are the adder/subtractor, shift left operator, shift right operator, multipliers, and delay elements respectively.

- Chapter Three presents three self-timed bit-serial architectures. The first part of this chapter discusses the benefits of self-timing and outlined the architectural requirements. The first architecture presented uses a state machine controller in each processing element to regulate data flow. It achieves self-timing and allows flexible processing element design, at the expense of a high hardware resource requirement. The second architecture presented achieves self-timing by buffering the entire input word before processing, resulting in significantly smaller systems compared to the first architecture, but with reduced throughput. The third architecture presented makes a compromise between size and speed, by changing the data format and introducing input restrictions. It is possible to achieve self-timing at a significantly reduced hardware cost compared to the first architecture, but with comparable throughput. The implementation detail and size and speed estimates for all three architectures are presented in this chapter.

- Chapter Four demonstrates a potential application of self-timed bit-serial architectures; floating point addition. The first part of this chapter provides a

background to floating number format and algorithms. The motivation for a bit-serial implementation of a floating point adder, namely a lower hardware resource requirement than bit-parallel implementation, is provided. The adder implementation is explained in detail, and a comparison between the self-timed bit-serial floating point adder and other academic and commercial implementations is also provided.

- Chapter Five demonstrates the application of a self-timed bit-serial architecture in the speech synthesis domain. This chapter starts by giving background information on the speech vocoder in the Klatt formant synthesizer chosen for implementation. Implementation details for the speech synthesizing system on a Xilinx VertexE FPGA, using the third self-timed bit-serial architecture are discussed, as are the size and performance of the system.

- Chapter Six provides a summary of the results and conclusions of this thesis. Recommended future work and directions for this research is discussed.

Chapter 2

Bit-Serial Systems Background

Bit-serial systems are different from bit-parallel systems in that they process data one bit at a time rather than one word at a time [2] [3]. There are a number of advantages associated with bit-serial systems. First of all, the critical path, the path with the longest combinational logic delay, is short in a pipelined bit-serial system, and can be implemented with minimum logic depth. This is because long logic sequences are broken up into smaller sections with flip flops. Therefore, even though the throughput, the number of words processed per second, is SWL times slower than a non-pipelined bit-parallel implementation of the same system structure at the same clock frequency, the bit-serial implementation can operate at a higher bit-clock frequency [4]. Fully pipelined bit-parallel systems can be clocked just as fast, but the hardware resources required to implement the parallel data processing structure and the pipeline stages are too great for most applications. Secondly, systems designed using bit-serial architecture are smaller, typically $1/n^{th}$ the size compared to an n-bit bit-parallel implementation [5] [6] [7]. Last but not the least, nearly 100% logic utilization can be acheived on FPGAs for bit-serial system implementation [4], due to bit-serial characteristics such as local connectivity, low fan-in and fan-out, and low Input Output (IO) pin count [8]. The routing density also stays constant regardless of the system size, a strong contrast to bit-parallel systems [8].

For the above reasons, the bit-serial approach is an attractive alternative when a bit-parallel implementation proves to be too large, or has the potential to operate at

a faster rate than the throughput required. Applications such as image processing [9], audio processing [10], and digital filters [11] [12] [13] [14] can benefit from the size advantage of a bit-serial architecture. The potential advantage of bit-serial is so great that new FPGA architectures with high-performance pipelined data paths, designed for more efficient bit-serial system implementations have been proposed [4] [15].

In this chapter, the static-timed bit-serial architecture, where the timing characteristic is fixed throughout the system, is described in detail. Processing elements used in a bit-serial signal processing system are also discussed.

2.1 Static-Timed Bit-Serial Architecture

In a bit-serial system, data is sent through a single wire and can be pipelined at the bit-level to increase the maximum clock frequency. Therefore, control structures are needed to create separations between system words and make sure multiple inputs are aligned before they arrive at a processing element. Delay elements also need to be added to a bit-serial system to store previous results for recursive loops. This section will explain architectural characteristics and control structures in a bit-serial system.

2.1.1 Data Frame Signal

Data frame signals are used to separate one system word from another in a bit-serial system, and are necessary because data is sent through a single wire one bit at a time, with no separation between words [2]. The frame signals run in parallel to the

data signal, and are set high on the Least Significant Bit (LSB) of every word. It should be noted that there are systems utilizing Most Significant Bit (MSB) first processing, also referred to as on-line arithmetics, for specialized applications [16] [17] [18]. However, due to the complexity of an on-line system, the majority of bit-serial systems are still designed for LSB first processing. When a processing element, such as an adder, detects a pulse on the frame signal, the internal memory will be reset to prepare for the arrival of a new word. A graphical illustration of the data frame signals for a static-timed bit-serial system is shown in Figure 2.1.

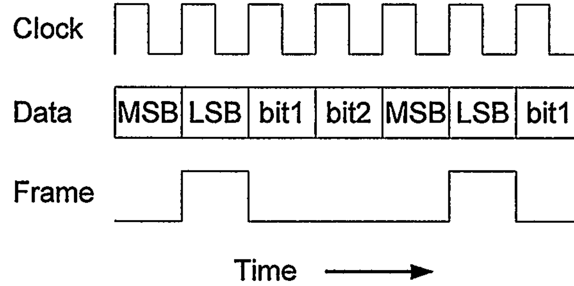


Figure 2.1: Data Frame Signal for Static-Timed Bit-serial Architecture

2.1.2 Two's Complement Number Representation

If signed numbers need to be represented in a bit-serial system, the two's complement number system is used. A two's complement representation of a number can be obtained by first converting its absolute value into binary and then inverting all bits and adding one to the result [19]. The word length must remain constant during this process. To convert it back to base 10 integer, the following formula can by used: $A = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$, where b is the binary two's complement representation of the number and n is the word length. The MSB of a number in a two's complement representation serves as the sign bit; a number is negative if and only if its MSB is 1.

In the two's complement number system, zero is considered positive because its sign bit is 0. Because two's complement has only one representation of zero, there is one extra negative number, -(2^{n-1}), that does not have a positive counterpart, resulting in a range of representable numbers from -(2^{n-1}) to +(2^{n-1}-1).

Other systems exist for signed number representations, such as signed-magnitude representation and ones' complement representation. In signed-magnitude representation, an extra bit is used to represent the sign of a number (0=positive, 1=negative), and there are two possible representations of zero. Even though signed-magnitude representation is easy to understand, it is not suitable for digital hardware implementation, because comparisons of the operands's sign bits need to be made before the operation can be carried out, which translates into increased complexity for the logic circuit. The ones' complement representation is similar to the two's complement representation, except that a weight of -(2^{n-1}-1), rather than -(2^{n-1}), is given to the MSB. The range of representable numbers is -(2^{n-1}-1) to +(2^{n-1}-1), and there are also two representations for zero. The main advantage of the ones' complement system is in it's symmetry and the ease of complementation. However, zero-detecting circuits in a ones' complement system need to check for both representations of zero.

Two's complement arithmetic makes signed calculations easy as basic calculations such as addition, subtraction, and multiplication can be carried out in the same way for both positive and negative numbers, making it the preferred choice over signed-magnitude and ones' complement representations. When performing an arithmetic operation with *n-bit* two's complement numbers, the carry out of the MSB is ignored and only the lower-order *n-bit* of the result is used as only the lower *n-bit* is correct,

unless the operands are properly sign extended to match the desired word length of the result. As long as the result does not overflow and exceed the range of representable numbers for the given word length, it will be correct [19], which fits well with the static-timed bit-serial architecture, as a constant word length must be kept throughout the system.

2.1.3 Latency and Scheduling

The time delay between the arrival of the input to a digital circuit and when the output becomes valid is called signal propagation delay. This includes flip flop switching delay, the time it takes for the output of a flip flop to become valid after a clock edge is detected, logic gate delay, the time it takes for the output of a logic gate to become valid after the input arrives, and interconnect delay, the time it takes for a signal to travel through an interconnect between two elements.

The maximum clock frequency of a system depends on the signal propagation delay of the critical path, the path with the longest combinational logic depth. This is because the outputs on all combinational logic paths must be stable before a clock edge arrives. In order to increase the maximum clock frequency of a bit-serial system, the system can be pipelined at bit-level by placing a flip flop at the output of each processing element. The flip flop re-synchronizes the logic at its output and breaks the combinational logic path into smaller sections, resulting in a shorter critical path. However, bit-level pipelining causes delays between the time of arrival of the LSB of the input at a processing element and the generation of the LSB of the output, a delay known as latency. Basic bit-serial processing elements have 1 to SWL clock cycle latency. However, zero latency elements may be included in a system with no effect

on overall performance, as long as they are not placed on the critical path [20]. In bit-parallel systems, excessive latency will slow down the overall system throughput in recursive loops, where the current output depends on previous outputs, because each clock cycle latency in a bit-parallel system translates to a system word latency. In a recursive system, processing needs to be paused until the result from the current processing cycle becomes valid before the next processing cycle can begin, effectively slowing down the system. This is less of an issue for bit-serial systems, because although their clock cycle latency is high, their system word latency is low, and in a recursive system, results from previous processing cycles are likely to be valid long before they are needed for current calculation [3].

If multiple data streams are merged together in a system, such as the one represented by the equation $F = A + B + C + D + E$, delays must be added to the shorter data channel to ensure that every channel has the same total latency when the outputs arrive at a processing element in the system. Otherwise, the bits will be misaligned, and the computation will be incorrect. Figure 2.2 provides a graphical illustration of this approach. The boxed number attached to the adders represent their latency.

If recursive loops exist in a bit-serial system, and results from previous computation cycles are needed to generate the current output, the previous results will be stored within the pipeline stages throughout the system. If the total latency in a particular loop is not the desired multiple of SWL clock cycles, extra delays will need to be added until the required latency is achieved [3]. For example, to implement a first order Infinite Impulse Response (IIR) filter described by the equation $y(n)=a \bullet y(n\text{-}1)+x(n)$, where n represents the sample interval and $x(n)$ represents the input,

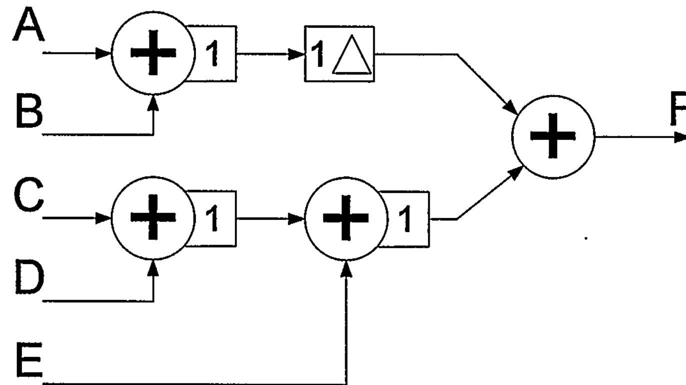


Figure 2.2: Multiple Input Channels in a Static-Timed Bit-serial Architecture

the result from the previous computation cycle, *y(n-1)*, needs to be fed back to the adder. Assuming the adder and the multiplier have a latency of one and five clock cycles respectively and the SWL is 16-bit, an extra 10 clock cycles delay needs to be added to ensure the bits from $a \bullet y(n\text{-}1)$ and *x(n)* are aligned when they arrive at the adder inputs. The minimum system word length for this filter is 6-bit, as that is the minimum latency in the recursive loop. Figure 2.3 provides a graphical illustration of this filter implemented using bit-serial architecture.

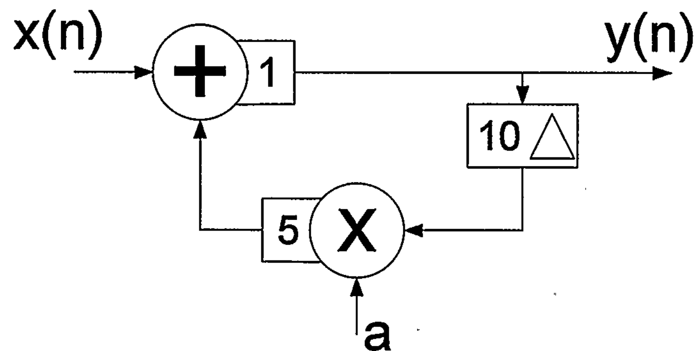


Figure 2.3: First Order IIR Filter Implementation in a Static-Timed Bit-serial Architecture

2.2 Basic Processing Elements and Their Structures

More complex bit-serial systems are formed from basic processing elements. This section describes the functionalities and structures of processing elements used for digital signal processing applications.

2.2.1 Adder/Subtractor

The adder is an important processing element in the digital signal processing domain as it is used in virtually every system and is the building block of more complex processing elements such as the multiplier. In a bit-serial system, a full adder [19] is used to perform addition. A flip flop is placed at the sum output of the adder for pipelining, which gives the adder unit a latency of one bit-clock cycle. The carry output is delayed by one bit-clock cycle so that it can be saved for the carry input in the calculation of the next most significant bit of the sum. The frame signal is used to clear the carry flip flop between words. When the LSB of a word arrives, the frame signal will be set high, by-passing the saved carry and sending a '0' to the carry input.

Subtraction in two's complement arithmetic is equivalent to addition of the two's complement of the subtrahend. Therefore, A-B is equivalent to $A+\overline{B}+1$. The structure of a bit-serial subtractor is similar to the adder, except the B input is inverted, and a carry in of '1' is used when the LSB arrives. Figure 2.4 shows the structure of a bit-serial adder and Figure 2.5 shows the structure of a bit-serial subtractor [3]. A timing diagram showing the bit-level activities of a 4-bit adder with continues inputs of the numbers 2 and 3 is shown in Figure 2.6.

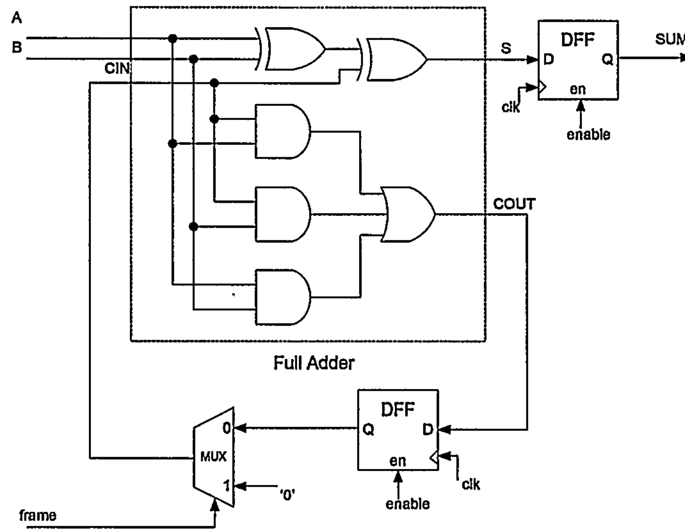


Figure 2.4: Bit-Serial Adder Structure

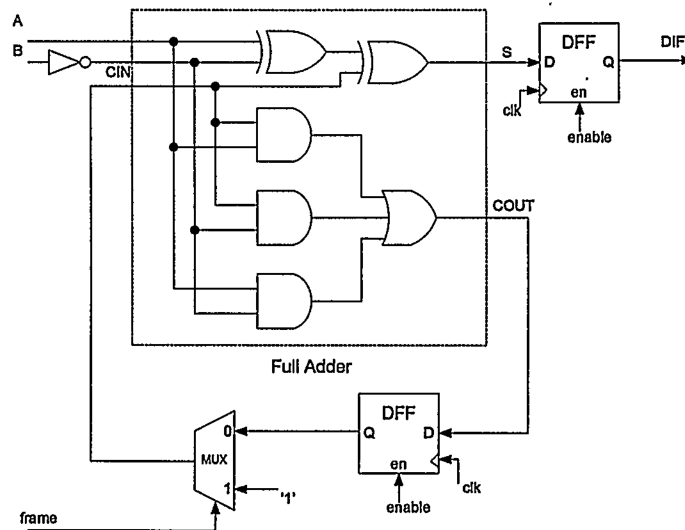


Figure 2.5: Bit-Serial Subtractor Structure

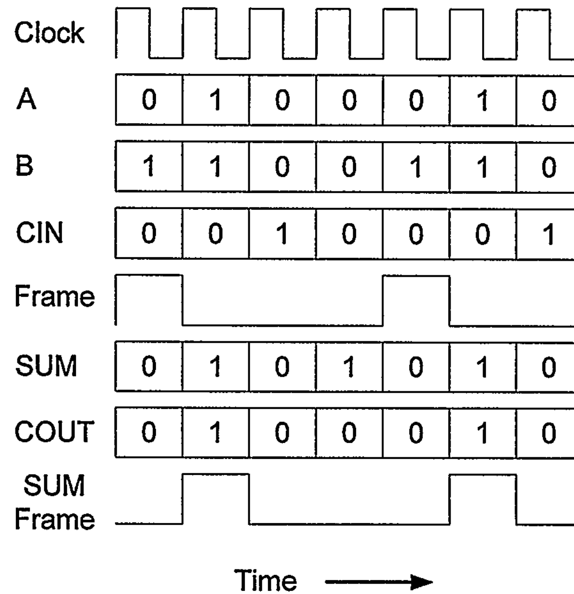


Figure 2.6: Bit-Serial Adder Timing Diagram

2.2.2 Shift Left Operator

The shift left operator [2] (also known as an M-shift operator), performs the same function as multiplication by the powers of two. To shift the input data by n bits to the left, the input is delayed internally by n bit-clock cycles. During these n bit-clock cycles, the output signal is set to zero, effectively truncating n MSBs of the previous word. A flip flop is is added to the output data and frame channels for pipelining, resulting in a one bit-clock cycle latency for this unit. The structure of a *1-bit* shift left operator is shown in Figure 2.7. An *n-bit* shift left operator can be constructed by holding the frame signal for n bit-clock cycles, and cascading n flip flops in front of the AND gate. Overflow will occur if the result of the shift exceeds the range of representable numbers for the system word length and is not handled by the processing element. The designer is responsible to ensure that overflow does not occur during system execution. A timing diagram demonstrating a 1-bit shift

left operation is shown in Figure 2.8.

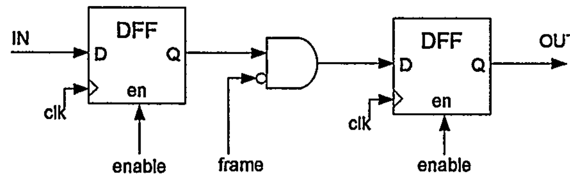


Figure 2.7: Bit-serial Shift Left Operator Structure

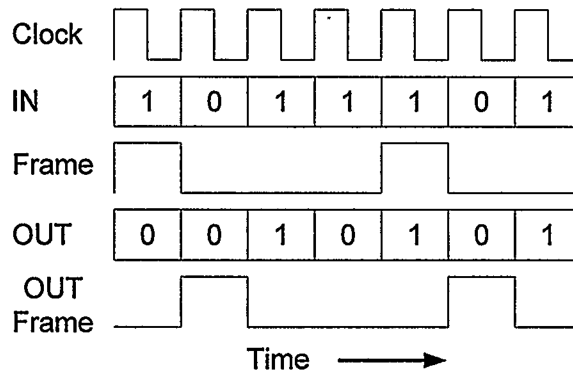


Figure 2.8: Bit-serial Shift Left Operator Timing Diagram

2.2.3 Shift Right Operator

In digital signal processing systems, divisions are simplified to divisions of powers of two because arbitrary division is complex and difficult to implement in systems with limited hardware resources. In a binary system, division of two is a shift right operation, also referred to as a D-shift [2]. In bit-parallel systems, a shift operation is a shift in space, but in bit-serial systems it is a shift in time. To perform a right shift of n bits, the latency will be $n+1$ bit-clock cycles. For n bit-clock cycles after a input frame signal arrives, the processing element will pause and hold the MSB of the current word, sign extending it to the proper word length while truncating the lower n bits of the next word. The input frame signal is also delayed $n+1$ bit-clock

cycles to ensure that the output frame signal is produced at the same time as the LSB of the output data. The structure of a 1-bit shift right operator is shown in Figure 2.9. A timing diagram demonstrating a 1-bit shift right operation is shown in Figure 2.10.

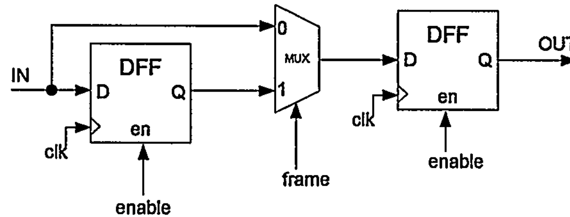


Figure 2.9: Bit-serial Shift Right Operator Structure

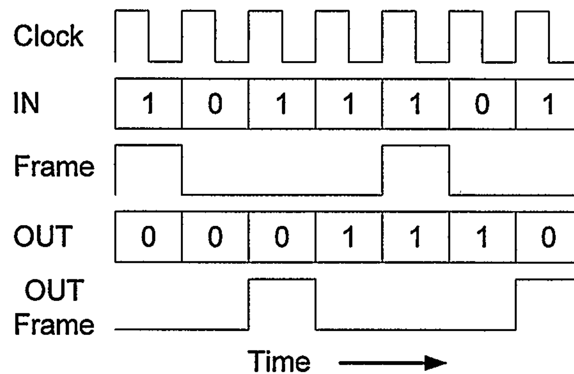


Figure 2.10: Bit-serial Shift Right Operator Timing Diagram

2.2.4 Multipliers

Multiplication is a complex, yet essential operation for DSP systems [21] [22]. In a parallel combinational multiplier, multiplication of two n-bit numbers requires $n \times (n-1)$ adders [3] [19]. In a bit-serial system, that requirement is reduced by a factor of n. Two different multiplier structures for bit-serial systems will be discussed in this section. One generates an *n-bit* output starting from the LSB, for an *n-bit*

multiplicand input, while the other generates a pre-shifted output in which only the most significant *n-bit* are outputted. Both multipliers operate on the principle of shift and add of partial products and require the multiplier coefficients to be provided in parallel. The partial products are generated using bitwise AND operations between the multiplicand and each bit of the coefficient, and are then shifted according to the positions of the coefficient bits before they are added together to generate the product. An example demonstrating the multiplication process for multiplicand A and coefficient B (both of which are represented in two's complement numbers) is shown in Figure 2.11 [3].

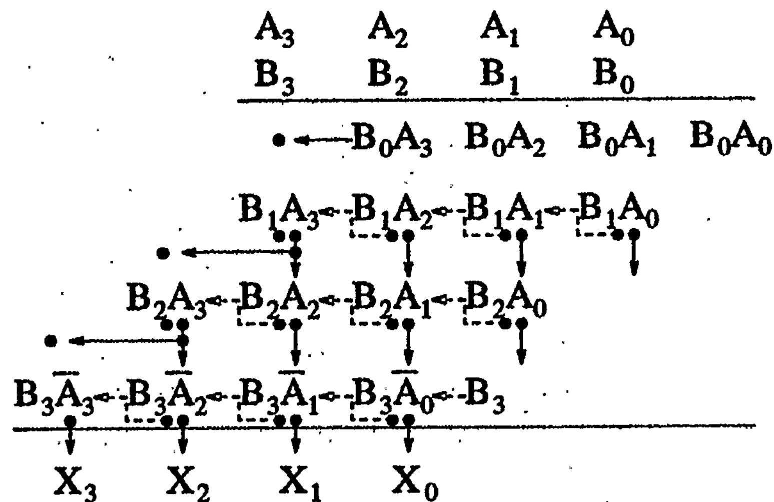


Figure 2.11: Carry Ripple Multiplication Table

Constant Coefficient Serial by Parallel Multiplier

The constant coefficient serial by parallel multiplier produces the product LSB first. With an *n-bit* input, the least significant *n-bit* of the product will be generated [23]. This multiplier operates on the principles of shift and add. Therefore, for a

multiplier with an *n-bit* coefficient, a series of *n-1* adders and one subtractor are required. The subtractor is there to handle negative numbers in two's complement representations. Flip flops are inserted between each internal stage to store and shift partial products, and a flip flop is also added to the product output of the multiplier to provide bit-level pipelining, making its latency one bit-clock cycle.

The multiplicand M is sent, LSB first, to the multiplier in serial, but the coefficient C needs to be loaded in parallel. Even though this multiplier is referred to as constant coefficient, the coefficient only needs to remain constant during one word computation. At the boundary where a new multiplicand input arrives, a new coefficient can also be loaded. At that time, the input frame signal is also sent to every adder/subtractor unit to reset all the carry inputs. The structure of an *n-bit* constant coefficient serial by parallel multiplier is shown in Figure 2.12 [23].

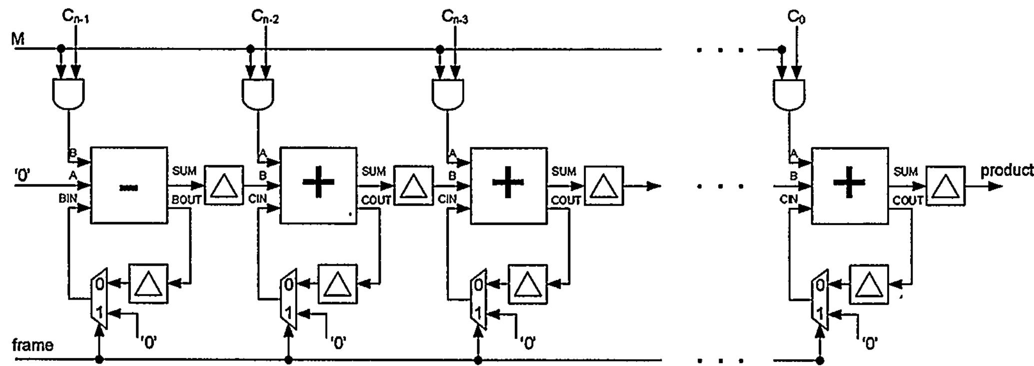


Figure 2.12: Bit-serial Constant Coefficient Serial by Parallel Multiplier Structure

Constant Word Length Serial by Parallel Multiplier

When a *m-bit* word is multiplied by an *n-bit* coefficient, the full length product is *m+n-1* bits long for two's complement numbers. However, in a static-timed bit-serial system, the word length must be kept constant throughout the system. Furthermore,

in digital filter applications, the coefficients C are rational fractions. To perform these multiplications correctly using the constant coefficient multiplier, the input word length needs to be increased to prevent overflow of the results. The coefficients are shifted left by powers of two before they are used for computation and the resulting products are shifted right to approximate a multiplication by a fraction.

The constant word length multiplier solves this problem by pre-shifting the product internally to truncate the lower *n-1* bits, outputting only the most significant m bits [3]. Consider a multiplication, $A \times B$, where both A and B are 4-bit two's complement number. The product X can be described by Equation 2.1.

$$X = -A \times B_3 + [A \times B_2 + [A \times B_1 + A \times B_0 2^{-1}]2^{-1}]2^{-1} \tag{2.1}$$

Equation 2.1 is obtained by the use of Horner's rule, shown in a tabular format in Figure 2.11 [3]. Multiplication of a signal by 2^{-1} is equivalent to a right shift by one bit. This operation performs two tasks simultaneously. First it truncates the LSB of the word, and then it sign extends the MSB. Thus a constant word-length is maintained for all signals. The sign bit of the partial product equals the MSB unless an overflow occurs in the partial product, in which case it equals the complement of the MSB [24]. An overflow can only occur if both inputs to the adder have the same sign bit, and can be detected when the input carry to the MSB addition differs from the output carry from the addition. However, implementing this detection mechanism requires additional hardware resources, and is rarely used. A more efficient way to solve this problem is to increase the word length of the multiplicand by one bit to ensure that overflow of partial products never occurs. In Figure 2.11, note that the MSB of the coefficient is added on the last row to generate

the two's complement of the last partial product. The structure of this multiplier is shown in Figure 2.13 [3].

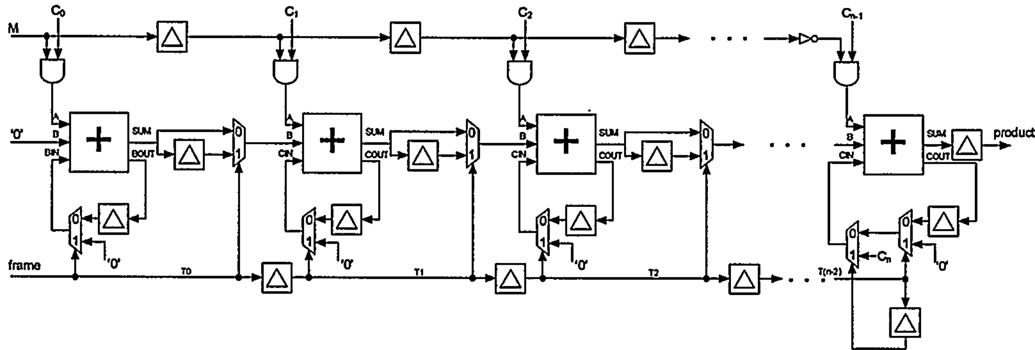


Figure 2.13: Constant Word Length Serial by Parallel Multiplier Structure

2.2.5 Delay Element

In a more complex bit-serial system, data path latencies for different bit streams vary as they arrive at the inputs of a processing element. To ensure that the LSBs of different input bit streams arrive at the same time, it is necessary to delay the shorter path until the latencies matches each other, making delay elements essential in a bit-serial system, as they are responsible for internal data storage and scheduling. The structure of an *n-bit* delay element implemented using flip flops is shown in Figure 2.14.

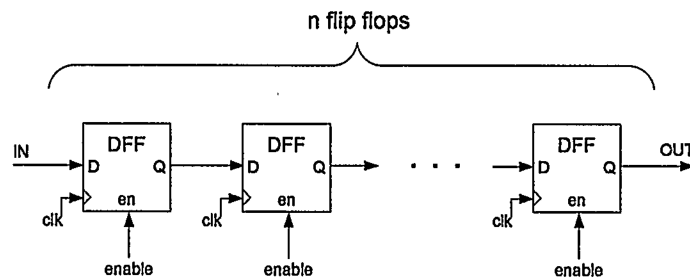


Figure 2.14: Bit-serial Delay Structure

2.3 Temporal Spatial Trade-Offs With Digit-Serial

Bit-serial systems are characterized by their single bit operations. However, if hardware resources are available and faster throughput is required, then the system can be designed to process more than one data bit per clock cycle. The resulting system is called a digit-serial system [3]. Digit serial systems with *n-bit* digits can potentially be n times faster than their bit-serial system counterpart, provided the system clock frequency is the same. However, as the number of bits per digit increases, the propagation delay due to combinational logic may also increase, resulting in a lower maximum clock frequency. The limit for a digit-serial system would be a bit-parallel system, where the digit width is the same as the word length. When designing a system, hardware size and speed requirements should be evaluated in order to choose a digit width for optimal performance. Figure 2.15 illustrats how a 2-bit digit-serial adder can be constructed from two full adders.

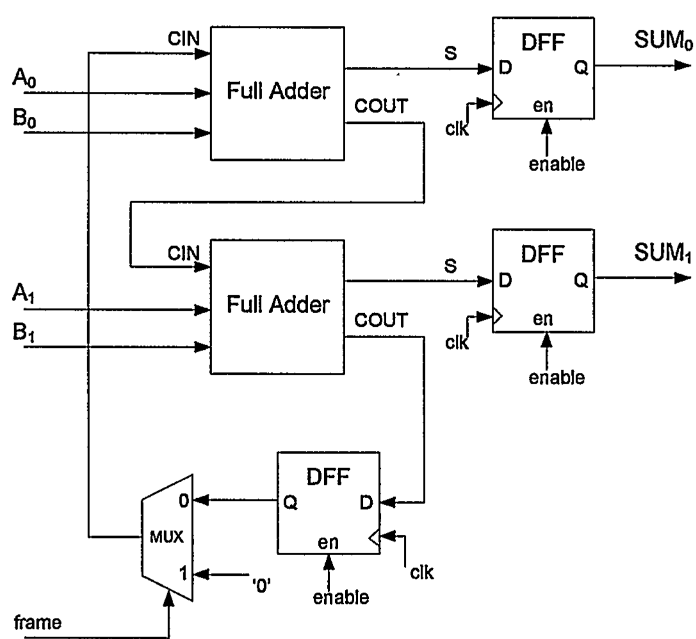


Figure 2.15: 2-Bit Digit-Serial Adder Structure

Chapter 3

Self-Timed Bit-Serial Architectures

Bit-serial systems are different from bit-parallel systems in that data is processed one bit at a time rather than one system word at a time. A frame signal is used to separate one word from another and to provide synchronization between processing elements. These frame signals are generated from a central controller unit in static-timed bit-serial architectures and cannot change during run-time. There are a number of disadvantages associated with a central controller unit. First of all, this kind of control unit generates long control lines which will result in long propagation delays, since signal delays due to the length of the wires are in the range of gate delays in today's chip implementations [25]. Secondly, it is difficult to scale up the system with a central controller unit as the scheduling has to be recomputed and the controller redesigned every time a change is made to the system. The control signals generated by the central controller are static, so dynamic timing changes within the system are not allowed. The goal of this research is to create an architecture where control signals are generated locally at each processing element, eliminating the need for a central controller. Systems based on this architecture will be self-timed, and capable of adjusting their scheduling during run-time.

3.1 Benefits of a Self-Timed Bit-Serial Architecture

There are a number of advantages with a self-timed bit-serial architecture. First and foremost, because scheduling is automatically adjusted during run-time, the system is easier to design, implement, and modify compared to a static-timed bit-serial approach. If the structure of a system is known, implementation is a simple matter of connecting the components, with no timing or scheduling calculation necessary. This simplified implementation process results in shorter design time and allows non-specialists to design bit-serial systems without detailed knowledge of bit-serial timing.

The second advantage of self-timed bit-serial is variable word length. In a static-timed bit-serial architecture, the word length must remain constant throughout the system during run-time, but this is no longer necessary in self-timed bit-serial systems because scheduling is automatically adjusted during run-time. Therefore, a system can reduce its word length during execution to increase throughput when necessary, and increase its word length when high precision or a large dynamic range is required. If two inputs of different word length are sent to one processing element, the input with a shorter word length will be sign extended until it matches the longer one.

Last but not the least, self-timed bit-serial architecture enables the possibility of building run-time reconfigurable systems. Run-time reconfiguration refers to the dynamic change of data paths and processing element functions during execution. This can result in a smaller system size, as hardware resources can be reconfigured to perform different functions. Implementing a run-time reconfigurable system using static-timed bit-serial architectures is difficult because any change to the data path

can potentially change the scheduling of the entire system. Self-timed bit-serial architecture solves this problem by dynamically adjusting the scheduling as the system structure changes.

3.2 General Requirements for Self-Timing

In order to achieve reliable timing at each processing element, a set of robust and non-ambiguous rules are required. Different methods for self-timing have been explored by other researchers. In the 2003 research by Achim Rettberg, a control marker is embedded in the data stream to separate the words, and special units called Routers was used to reconfigure data path during run-time [25]. Other research in 1996 by R.S. Hogg achieved self-timing and built arithmetic logic units using an asynchronous design [26]. In this research, a three-wire handshaking scheme was chosen for better performance and data format flexibility.

In a static-timed bit-serial system, we assume that the bit streams are synchronized when they arrive at the inputs of a processing element, so a common control signal can be shared by all input signals. However, in a self-timed architecture, that assumption is no longer valid, as data for different inputs can arrive at different times. Therefore, a frame signal is needed for every data signal. In the case where one input arrives earlier than other inputs, the processing element must be able to pause that particular input until the rest of the inputs arrive. Pausing is accomplished through a second control signal, named *ready_receive*, generated at every input of a processing element. If multiple processing elements are connected in series, the *ready_receive* signal will propagate from the point of origin until it reaches the first element in the

series. To minimize propagation delay, the *ready_receive* signal is pipelined. Therefore, if the *ready_receive* signal level changes on the n-th element in the series, it will take n-1 clock cycles before the change will be detected by the first element. For this reason, instantaneous response to the *ready_receive* signal is not necessary. When a *ready_receive* signal initiates a pause on an input, the input should be paused on the next clock cycle, and when pausing is released, the input should resume data transmission after one clock cycle. Three different self-timed architectures are presented in this chapter, each with its own advantages and disadvantages. All three architectures use the two control signals (*frame* and *ready_receive*) for every data signal to achieve self-timing.

3.3 Architecture One: Controller Regulated Architecture

The first architecture presented uses a state machine controller for every processing element to generate all control signals, hence the name Controller Regulated Architecture. Each controller will be responsible for detecting input frame signals, generating *ready_receive* signals for all inputs, and generating the output frame signal.

3.3.1 Data Format and Control Signals

Two control signals are associated with each data signal, making a total of three wires per channel. The data signal carries the actual bit stream for each data word, and does not have any control signal embedded. The frame signal separates one word from another and indicates if the bit stream is currently paused. This is accomplished by toggling the frame signal low for the MSB of each word as well as when the bit stream is paused. If the bit stream is shifting normally and the MSB is not yet reached, the frame signal will be kept high. The *ready_receive* signal is kept high for an input if the processing element is ready to receive that input; when *ready_receive* goes low, the input data will be paused on the next clock cycle, and resumes one clock cycle after *ready_receive* goes back to high. A graphical illustration of the data format is shown in Figure 3.1 to provide a better understanding of the process.

3.3.2 Architecture Overview

Each processing element in this architecture is composed of three sections: input buffers, a control unit, and the actually processing unit.

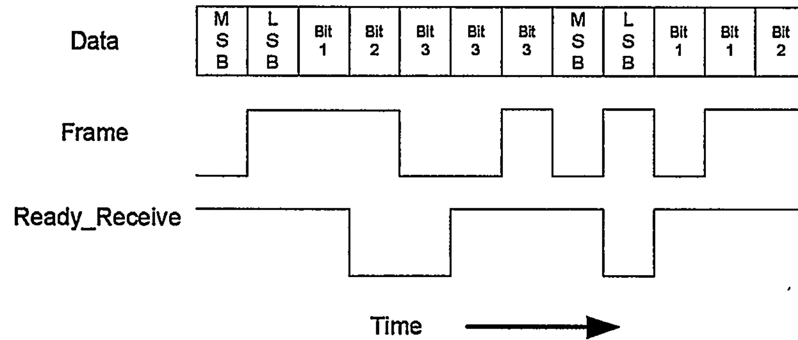


Figure 3.1: Architecture One Data Format and Control Signals

Because the control signals are pipelined, the input signals will not respond instantaneously, and it will take one clock cycle for any change in the control signals to reflect on the data signal. Therefore, a buffer system is needed at the inputs of every processing element to store the input bits for the clock cycle where the processing unit is paused but the inputs are still shifting. The buffer system will also provide inputs to the processing unit for one clock cycle after processing resumes, before the input bit streams become active. Since the buffer is only active for one clock cycle at a time, it is only necessary to make the buffer one bit deep in size, and only one flip flop and a multiplexer to select the input to the processing unit is needed. Both the flip flop enable and the select line of the multiplexer are controlled by the control unit in each processing element. When the controller initiates a pause on one input, the flip flop is disabled and mux select line toggled low. When data transmission resumes, the flip flop is enabled and mux select line toggled high. Processing can not start until all inputs for a multi-input processing element become valid, so the *ready_receive* line for every input will be held low until the LSB for all the inputs arrive. A graphical illustration of the buffer system is shown in Figure 3.2.

As discussed earlier, each processing element is equipped with its own controller.

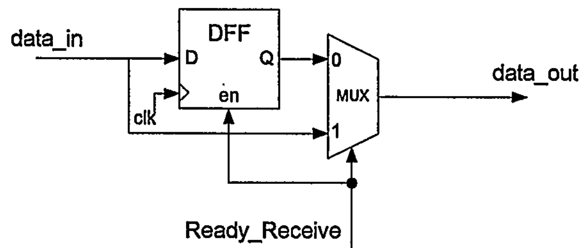


Figure 3.2: Single Bit Input Buffer for Architecture One

This controller is composed of an one hot encoded state machine that is designed to supply all the control signals required for the input buffers and the processing unit, as well as generate the output frame signal. The controller can be different for different processing elements, depending on the control signals required. For example, the controller for processing elements that require parallel inputs will be different from the controller for processing elements that do not, and processing elements that have different number of inputs will also have different control units. For processing elements that have the same control requirements, the same design can be shared for the control unit. However, if the control requirements are different, a custom control unit will need to be designed for each processing element. The general structure of a two-input processing element is shown in Figure 3.3.

A number of common processing elements have been designed and built to test the performance of this architecture. These processing elements are the essential building blocks of a digital filter and are fundamental for signal processing.

3.3.3 Adder/Subtractor

One of the basic and important processing elements in a digital signal processing system is the adder. The structure of the adder and subtractor is similar with the

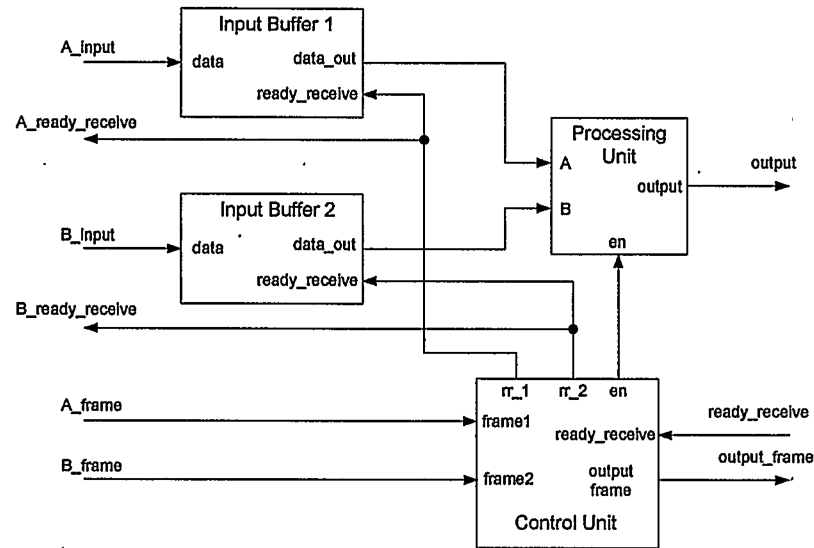


Figure 3.3: Two-Input Element Structure for Architecture One

only difference being the inverter at the subtrahend input of the processing unit. Detailed structures of both the adder and subtractor were discussed in Section 2.2.1 for a static-timed bit-serial architecture, and they are used for the processing unit section for both processing elements. The enable signal for the flip flops and the carry clear signal in both processing units are controlled by the control units to pause and resume processing.

The control unit consists of a five-state one hot encoded state machine. Five states are needed because there are five possible modes that the processing element could be in: initial state to wait for both inputs to be valid; allow both inputs to transmit, allow *input_1* to transmit while pausing *input_2*, allow *input_2* to transmit while pausing *input_1*, and pausing both inputs. The detailed design of this state machine is shown in Figure 3.4.

The inputs to this state machine is the two frame signals, *a_frame* and *b_frame*,

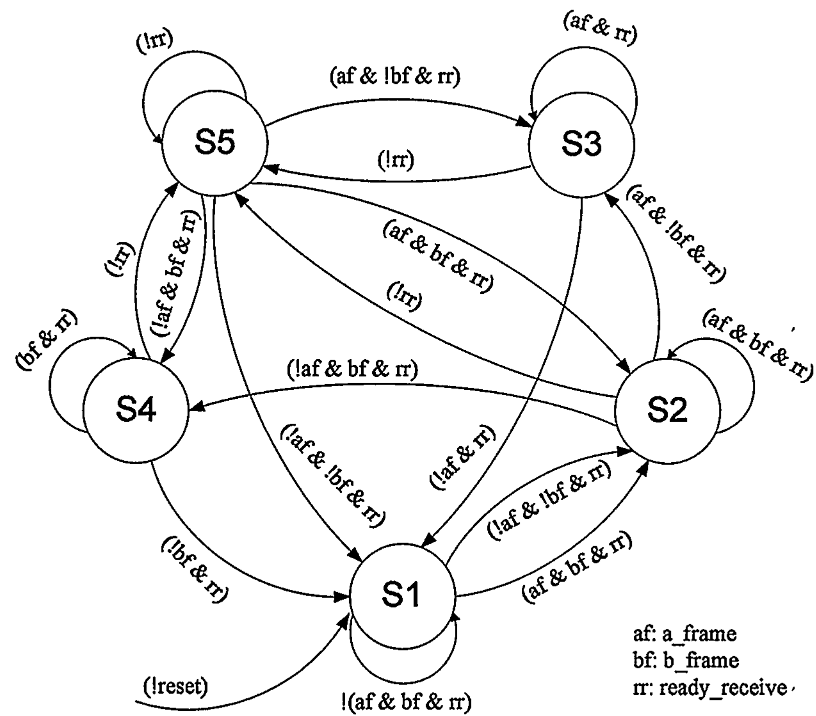


Figure 3.4: Adder/Subtractor State Machine Controller Structure for Architecture One

and the *ready_receive* signal for the processing element output. At the reset state, the system will wait until both inputs to the adder become valid. Once they are valid, the system will start its processing. If the output's *ready_receive* signal goes low during processing, the control unit will pause both inputs and wait until the external pause is lifted.

3.3.4 Serial by Parallel Multiplier with Parallel Load

The multiplier is an essential yet complex unit in a digital signal processing system. Even in bit-parallel systems, multiplication is often done sequentially to reduce hardware cost. Using a bit-serial architecture, the multiplier can be implemented at a fraction of the hardware resource of a bit-parallel implementation for the same word length.

In this self-timed bit-serial architecture, the constant coefficient serial by parallel multiplier shown in Section 2.2.4 is chosen as the basis of the multiplier unit. Because the coefficient has to be sent to the system in parallel, a parallel latch is used to load the coefficient input to the system. The structure of the multiplier unit is shown in Figure 2.12.

The control unit for the multiplier is responsible for regulating the multiplicand input, loading the coefficient at the right time, and generating the output frame signal. The control unit consists of a five-state one hot encoded state machine as shown in Figure 3.5.

Five states are needed for the five possible operating modes of the processing element: initial state waiting for the input multiplicand to be valid, loading the multiplier coefficient, continuous data processing, pausing the input bit stream, and

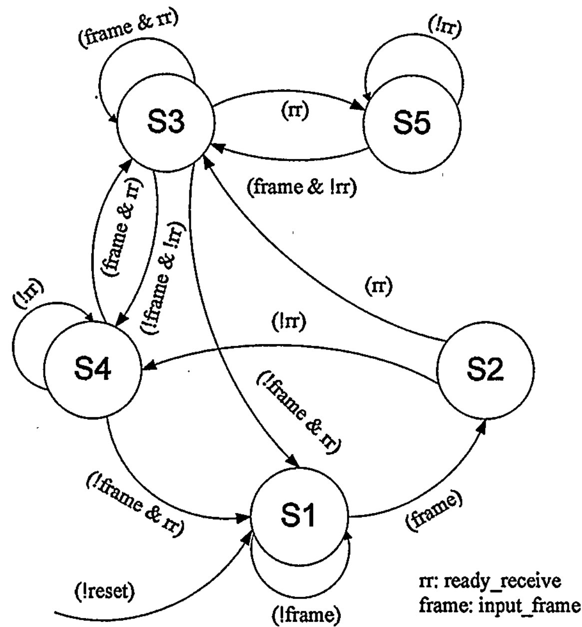


Figure 3.5: Multiplier State Machine Controller Structure for Architecture One

the special case scenario where the output's *ready_recieve* signal goes low at the same time as when the MSB of the input multiplicand arrives.

3.3.5 Delay Element

As discussed in Section 2.1.3, if there are recursive loops in a bit-serial system, the total latency in the loops must be whole number multiples of the system word length. In static-timed bit-serial systems, this is accomplished by adding delays in the system to adjust latency. These delays are static and can not meet the dynamic timing requirements of a self-time bit-serial architecture, so a special delay element is needed. This delay element must be able to adjust its length based on the system word length and the timing characteristics of each system that it is incorporated into.

The delay element is built around a multi-tap shift register, where the output can be read from different points in the shift register. Since the introduction of the Xilinx Vertex FPGA, every Look Up Table (LUT) can be configured as a 16-bit shift register with selectable output, called SRL16. Multiple SRL16 can also be cascaded to form longer structures, allowing shift registers to be built cheaply.

Two pointers are used to keep track of the position of a word within a shift register, one pointing to the position of the output while the other one pointing to the MSB. The two pointers will move closer together as the word is shifted out, and when an overlap of the two pointers is detected, the system will move on to the next word in the shift register. The two pointers are constantly compared with their minimum and maximum values to detect if the shift register is full or empty. A graphical illustration of this process is shown in Figure 3.6.

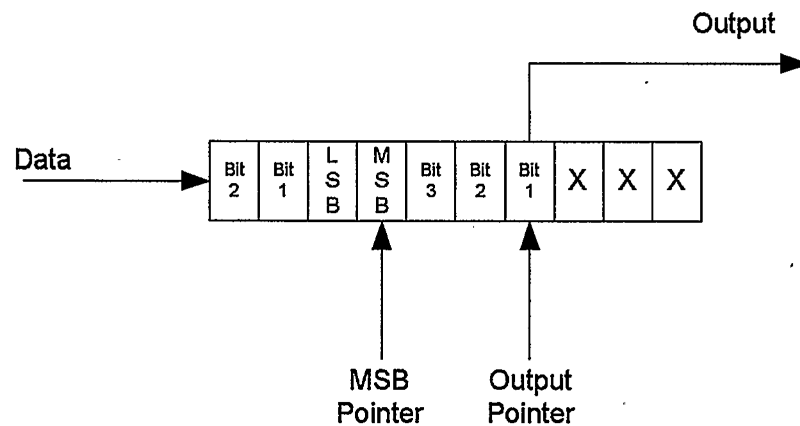


Figure 3.6: Delay Element Concept for Architecture One

Two state machines are used for the controller unit in this processing element, one for generating *ready_receive* control signal for the input, while the other controls the shifting and manipulates the pointers.

The *ready_receive* controller consists of a three-state one hot encoded state machine, which will monitor the position of the MSB and output pointers to detect when the shift register is full. If the shift register is full, the controller will pause the input until the stored bits starts to shift out. As there are only two pointers available for the delay element, it is not possible to store two full words in the element. Therefore, if the MSB of the second word is detected before the first word is shifted out, the controller will also pause the input and wait for the first word to complete its transmission. The state machine controller responsible for the *ready_receive* signal is shown in Figure 3.7.

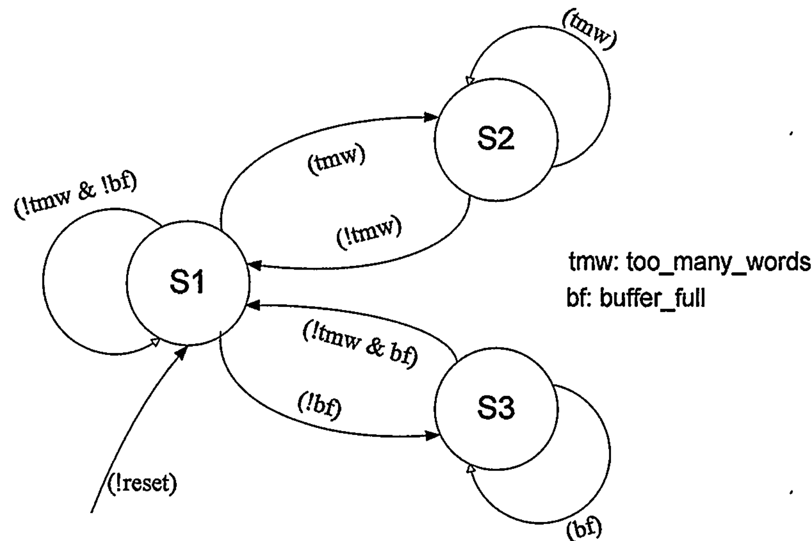


Figure 3.7: State Machine for *ready_receive* Signal Generation

The controller responsible for data shifting is a four-state one hot encoded state machine. The four different states represent the four possible operating modes of the shift register: shift in but not out, shift out but not in, shift in and out at the same time, and do nothing. A graphical illustration of this state machine is shown in Figure 3.8.

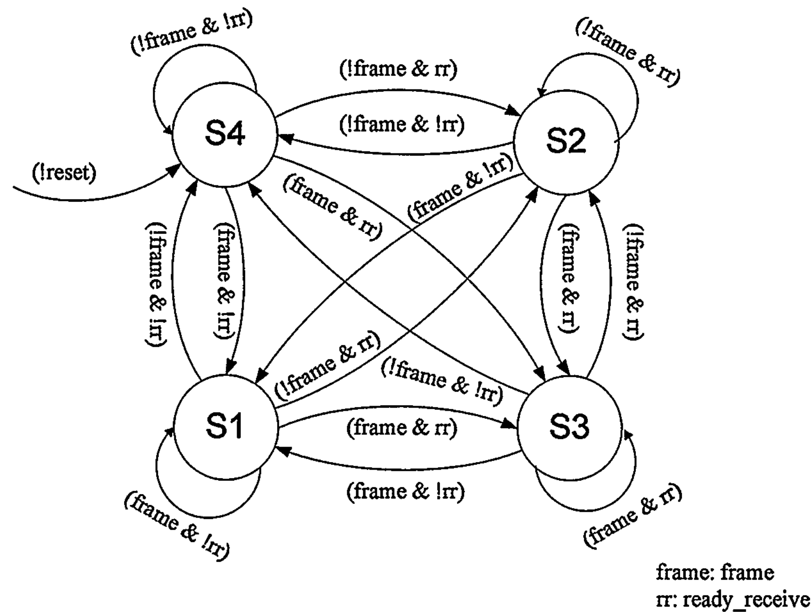


Figure 3.8: State Machine Controller for the Delay Element

3.3.6 Fork Element

In static-timed bit-serial systems, when the output of a particular processing element needs to be sent to two separate inputs, it can be wired directly. If the two branches for the inputs have different timing characteristics, static delays are added to one of the branches until their latencies match each other. However, this practice is not possible for self-timed bit-serial systems because the timing characteristics are not known in advance, and may change during run-time. Therefore, a special unit called the fork element is needed at every branch junction in the system. The fork element will read in the output data from the previous unit and send it to two separate branches, while complying with the self-timed protocol. This process is accomplish by implementing a variable size buffer at each of its outputs. If one output branch needs to be paused, the buffer on that branch will store the input data until the

pausing is released, such that any pausing on the output branches will not affect the input data flow.

The data buffer on each branch of the fork element has a similar structure to the delay element, but the initial conditions for the address pointers are different. In the fork element, both address pointers are initialized to be zero upon reset, making both buffers empty rather than a size of one bit, as in the case for the delay element. The buffers in the fork element are also assumed to be never full, so the fork element will always be ready to receive new data and conflicts between the two buffers will not occur. However, the fork element must be designed to accommodate the largest possible system word length. Another restriction imposed on the fork element is that only one word is allow to be stored in its registers at any given time. This way, the MSB pointers are no longer needed, and can be eliminated to save hardware. The structure of the fork element is shown in Figure 3.9.

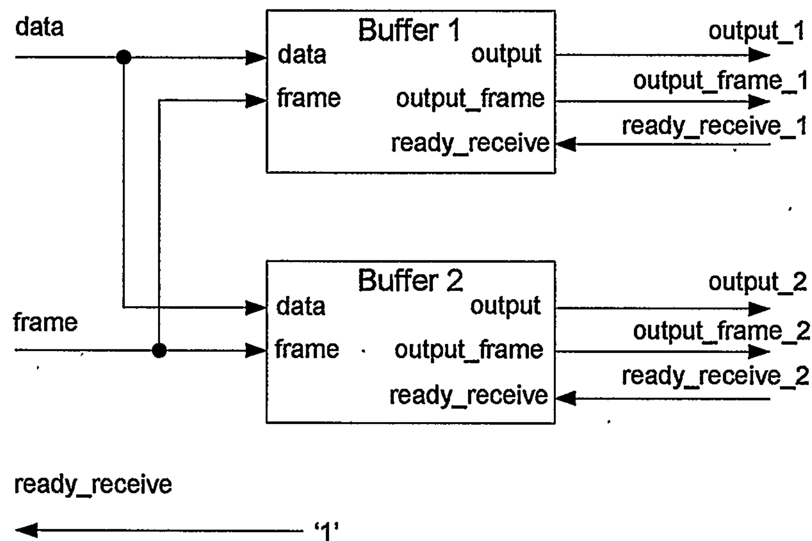


Figure 3.9: Fork Element Structure

3.3.7 Summary of the Controller Regulated Architecture

This architecture achieves self-timing by using a state machine controller in each processing element to regulate its inputs and output in accordance to the self-timed protocol. Special units such as the delay and fork elements are designed to ensure proper operation of the self-timed bit-serial system.

However, the state machine controllers are complex and difficult to design, and the fork and delay elements require address pointers which are expensive to implement in an FPGA. Systems designed using this architecture are complex and large, diminishing the small size advantage of a bit-serial system. The size and speed estimation of each processing element in this architecture is shown in Table 3.1.

	LUTs	Flip Flops	Max Frequency (MHz)
Adder/Subtractor	21	9	78.309
Multiplier (16-bit coefficient)	67	56	65.815
Delay Element (32-bit word length)	44	21	66.578
Fork Element (32-bit word length)	55	23	57.657

Table 3.1: Size and Speed Estimation of Processing Elements in Architecture One.

3.4 Architecture Two: Full Word Buffer Architecture

The previous self-timed bit-serial architecture achieved self-timing using a sophisticated control unit for each processing element. However, a significant drawback for this approach is high hardware resource usage. The complex control units, delay elements, and fork elements required a large number of logic cells to implement, significantly reducing the small size advantage of bit-serial systems. Therefore, a more hardware efficient self-timed bit-serial architecture needs to be designed.

The second architecture was inspired by pipelined bit-parallel systems, where each full word is loaded to the next processing element before processing begins. It is called the Full Word Buffer Architecture. This architecture eliminates the need for a state machine controller and complex fork elements. The simplified control structure also results in a higher maximum clock frequency for the system. This section provides a detailed description of this architecture.

3.4.1 Data Format and Control Signals

The data format and control signals are the same as for the previous architecture. The frame signal stays high for all valid non-MSB bits, and the *ready_receive* signal controls the pausing of data streams. For a graphical illustration of the data format and timing scheme please refer to Figure 3.1.

3.4.2 Architecture Overview

The differences between this architecture and the previous one are the larger input buffers, and the absence of a state machine controller. Because input buffers will

operate relatively independently of each other, a complex state machine is not needed to regulate the inputs. Instead, each input buffer has an simple embedded controller only responsible for the control signals for that particular buffer. Once all inputs are completely stored in the buffers, the processing will then begin. This way, once processing starts for a word, it will not be interrupted until it finishes. To keep the structure of input buffers simple, their size is fixed. Therefore, variable word length during run-time is not possible. The structure of a general two-input processing element is shown in Figure 3.10.

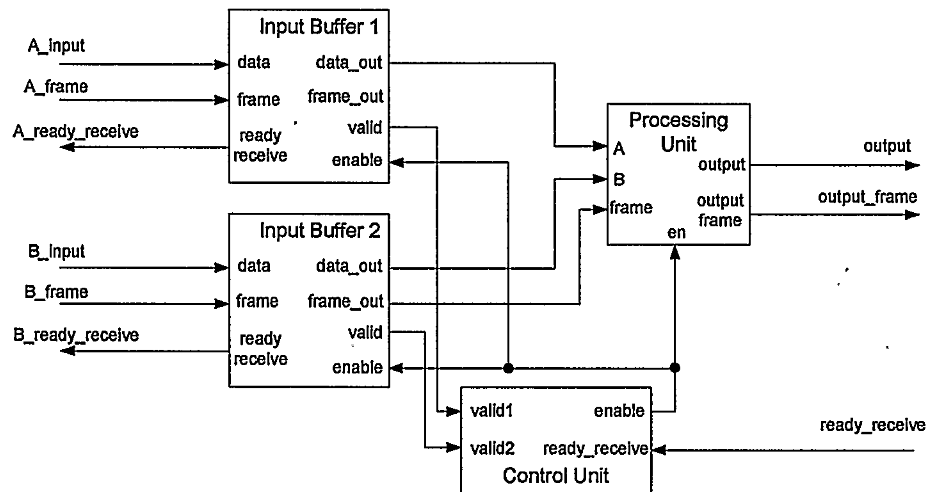


Figure 3.10: Two-Input Element Structure for Architecture Two

Even though a complex central state machine controller is not necessary for this architecture, a simple controller is still needed to synchronize the input buffers and to produce the enable signal for the processing element. This controller consists of a multi-input AND gate and one flip flop as shown in Figure 3.11. The flip flop on the *ready_receive* input is responsible for pipelining the control signals to reduce the maximum combinational logic delay.

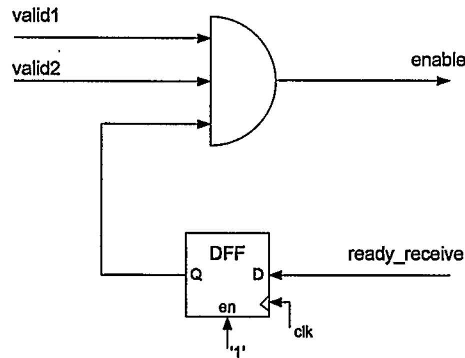


Figure 3.11: Two-Input Element Central Controller for Architecture Two

There are a number of advantages with distributing the controls to each input buffer. First of all, the structure of each control unit is simple and consumes very little hardware resource. Secondly, because the control units are embedded into the buffers and share an identical design, the system is easy to scale up. Creating processing elements with more inputs is a simple matter of connecting a buffer to each extra input, rather than redesigning the control unit, as in the previous architecture. In an ASIC design, creating large buffers would be expensive. However, since this architecture is targeted to the Virtex FPGAs, where the buffers can be implemented with SRL16 shift registers [27], increasing the buffer size only results in a minimal increase in hardware requirement. The structure of the input buffer along with the embedded control logic is shown in Figure 3.12.

3.4.3 Adder/Subtractor

The adder/subtractor design for this architecture is simple. The general two-input processing element structure is used for the basis of these two processing elements, and the processing unit section is replaced by the adder/subtractor unit described in Section 2.2.1. The only difference is in how the carry is reset. In a static-timed

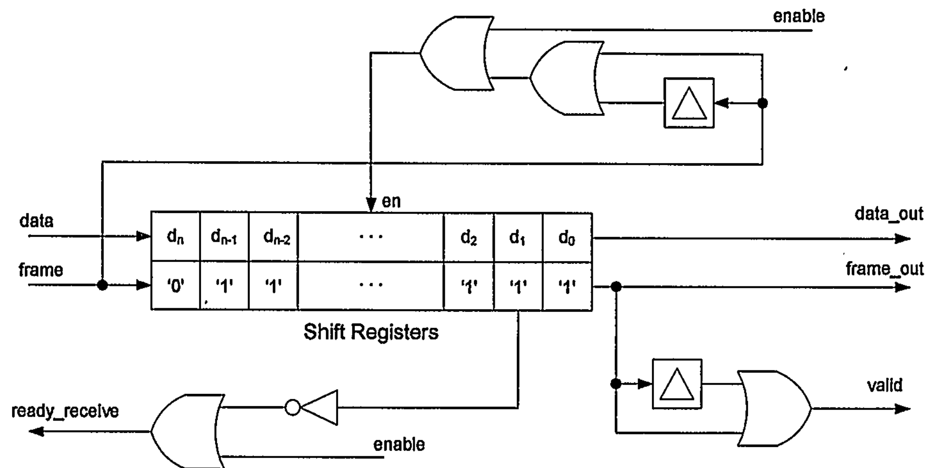


Figure 3.12: Input Buffer for Architecture Two

bit-serial system, the frame signal is a pulse at the LSB of every word. However, for this self-timed architecture, the frame signal stays high for every bit other than the MSB, so a different way to reset the carry signal is needed.

Because data processing will not be interrupted during a word, carry reset can be accomplished by placing the carry select multiplexer before the carry register. When the frame signal is high, the carry signal generated from the inputs is sent to the carry register, but when the frame signal goes low, zero will be sent instead. The structure of this modified bit-serial adder is shown in Figure 3.13. For detailed structural diagrams of the full adder/subtractor unit, please refer to Figure 2.4 and 2.5 in Section 2.2.1.

3.4.4 Multiplier

The constant coefficient serial by parallel multiplier described in Section 2.2.4 is used as the basis for the multiplier unit in this architecture. However, instead of having a parallel input for the multiplier coefficient, the coefficients are sent to the input buffer

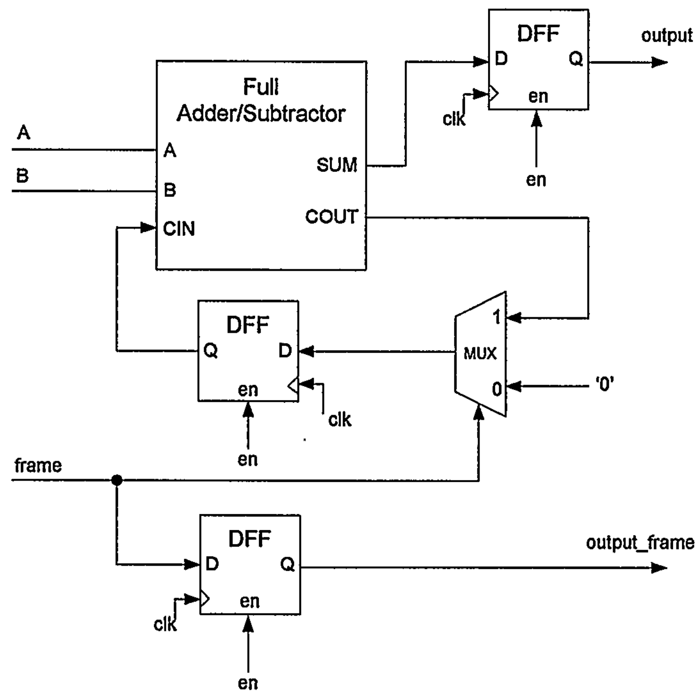


Figure 3.13: Adder Unit for Architecture Two

in serial and then latched into the multiplier in parallel, effectively creating a serial by serial multiplier, and resulting in a unified data format throughout the entire system. The SRL16 component in the Virtex FPGAs only allows single bit output, so the input buffer responsible for buffering the multiplier coefficients cannot use SRL16s for its shift registers, as parallel output of the coefficient is required. Therefore, a custom shift register with parallel outputs is designed by cascading a series of flip flops. As soon as the inputs are completely buffered, the coefficient is latched into the multiplier and the processing begins. The structure of the processing unit section of this multiplier is shown in Figure 3.14. For a detailed structural diagram of the constant coefficient serial by parallel multiplier please refer to Figure 2.13 in Section 2.2.4.

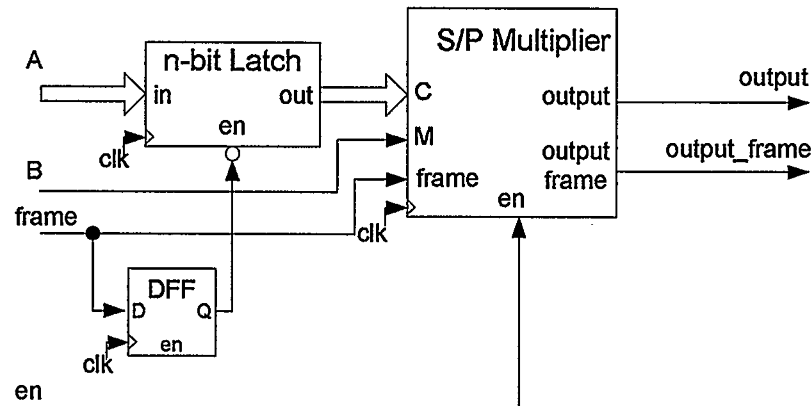


Figure 3.14: Processing Unit of the Multiplier for Architecture Two

3.4.5 Delay Element

The delay element used for a system word length delay in a recursive feedback loop has essentially the same structure as an input buffer in this architecture, with the only difference being the two flip flops at the data and frame output. This delay element is significantly smaller than the one in the previous architecture, primarily because of the fixed system word length. Because the system word length will no longer change during run-time, the delay element size can be fixed, and address pointers for tracking the location of a word is no longer needed in the delay element, resulting in significantly reduced hardware resource requirements. This delay element operates by storing an entire word first before sending it to the next processing element. The structure of the delay element is shown in Figure 3.15. Please refer to Figure 3.12 for the structure of the input buffer.

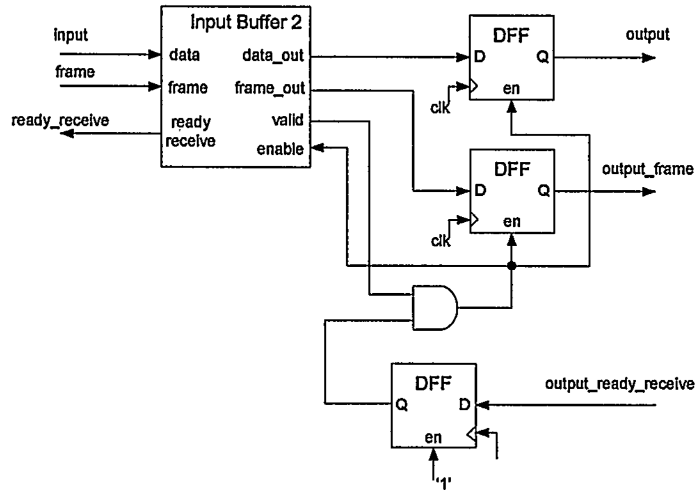


Figure 3.15: Delay Element for Architecture Two

3.4.6 Fork Element

A fork element is still needed in this architecture when connecting an output to two different operations. The fork element described in Section 3.3.6 required a large amount of hardware resources to implement, because it is composed of two buffers, each with an address pointer to handle variable system word length and latency at run-time. In this architecture, each processing element input is equipped with its own full word buffer, eliminating the need for the two buffers in the fork element. Because the system word length is fixed and can not change during run-time, the address pointers are also not needed. The resulting fork element is very simple compared to the one in the previous architecture. In fact, only a single AND gate is needed to generate the *ready_receive* signal for the input, and the rest of the signals can be connected directly. The structure of the fork element is shown in Figure 3.16.

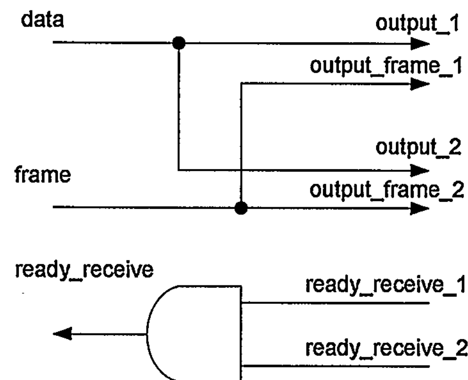


Figure 3.16: Fork Element for Architecture Two

3.4.7 Summary of the Full Word Buffer Architecture

This architecture achieved self-timing at a significantly lower hardware cost compared to the previous architecture, due to a simpler control unit for each processing element, and the smaller delay and fork elements. However, the full word buffer at each input introduced a SWL latency for every processing element and creates a SWL gap between each system word, resulting in a significantly lower throughput compared to systems built using the previous architecture or a static-timed bit-serial architecture. This slowdown is especially noticeable for systems with recursive feedback loops, such as an IIR filter. The size and speed estimation of each processing element in this architecture is shown in Table 3.2.

	LUTs	Flip Flops	Max Frequency (MHz)
Adder/Subtractor	12	10	124.688
Multiplier (16-bit coefficient)	56	73	114.025
Delay Element (32-bit word length)	7	6	158.680
Fork Element (32-bit word length)	1	0	N/A

Table 3.2: Size and Speed Estimation of Processing Elements in Architecture Two.

3.5 Architecture Three: 2-Bit Buffer Architecture

Even though both architectures in Sections 3.3 and 3.4 achieved self-timing, they both have significant drawbacks. In the first architecture, a state machine controller is responsible for generating all of the control signals in a processing element, resulting in complex control units that consumed a significant portion of the hardware resource. The second architecture simplified the control unit by buffering an entire system word before processing, introducing long gaps between words, and slowing down the throughput significantly. Because it is desirable to have a system that is both fast and small, a compromise between structural complexity and speed is the next goal for the self-timed bit-serial architecture.

One way to simplify the control unit is to reduce its responsibilities. In the first architecture, the controller is responsible for generating all control signals. However, certain control signals can be produced through a fixed logic sequences, and do not need to be generated by the controller's state machine. Restrictions can be imposed on the system to further reduce the control requirement. As a result, a new type of self-timed bit-serial architecture, called the 2-Bit Buffer Architecture is introduced, and will be described in detail in this section.

3.5.1 Data Format and Control Signals

The data format and control signals are slightly different in this architecture compared to the previous two. It still uses two control signals for every data signal, but the forward frame signal no longer covers the entire valid word. Instead, the frame signal is only a single pulse at the LSB of every word, much like the frame signal

in static-timed bit-serial systems. Changing the frame signal to this format imposes certain limitations on the system. First of all, the state of the processing element is not known based solely on its output as the frame signal will not show whether the processing element is paused or not. Secondly, the processing element can no longer initiate a pause of its output by itself, due to the fact that if the processing element initiates a pause on its output, other units in the system have no way of detecting the pause, as it is not reflected on the frame signal. In this architecture, all output pauses are initiated externally. Each processing element still has full control over its inputs through *ready_receive* signals. This data format is shown in Figure 3.17.

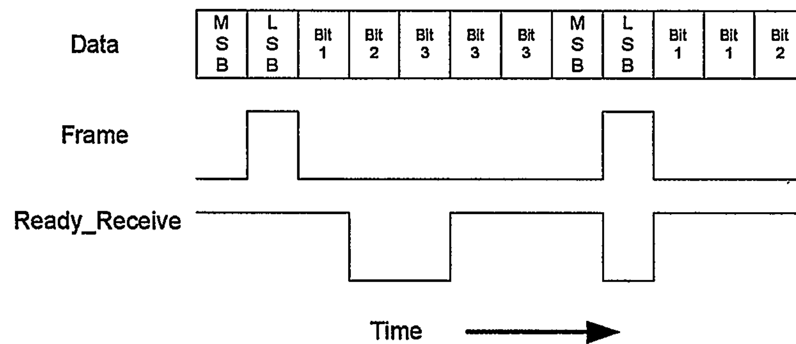


Figure 3.17: Architecture Three Data Format and Control Signals

3.5.2 Architecture Overview

The structure of each processing element in this architecture is similar to the structure in the Controller Regulated Architecture. Each processing element is equipped with a buffer for every input to provide one bit of storage every time the system pauses, a controller to regulate data flow, and a processing unit for the actual processing of data.

For this architecture, the bit-level pipelining is switched from the output of each

processing element to the inputs, resulting in a 2-bit buffer at every input. There are two reasons for this change. First of all, the input frame signal can be detected by the controller one clock cycle before it arrives at the processing unit, giving extra time for the controller to react and making the state machine easier to design. Secondly, by switching to an input pipeline, the two flip flops and one multiplexer of the input buffer can be implemented using just one LUT, by utilizing the SRL16 part. This creates a balance between the LUT and flip flop usage in the system and leads to more efficient usage of the hardware resources in an FPGA. The structure of the two bit input buffer is shown in Figure 3.18, and a SRL16 implementation of the same structure is shown in Figure 3.19.

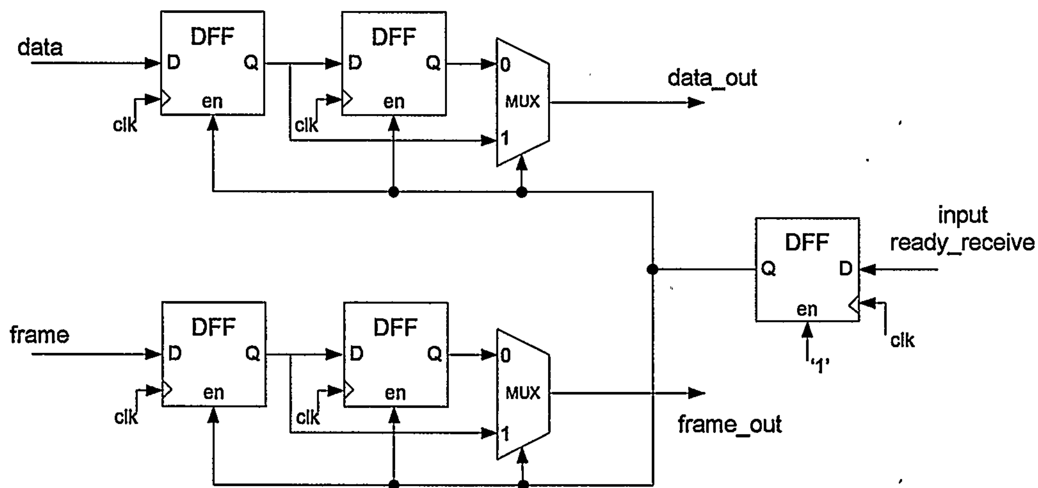


Figure 3.18: DFF Implementation of the 2-Bit Input Buffer

3.5.3 Two-Input Elements

The majority of processing elements in the signal processing domain, such as adders and multipliers, are two-input elements. These two-input elements can be cascaded

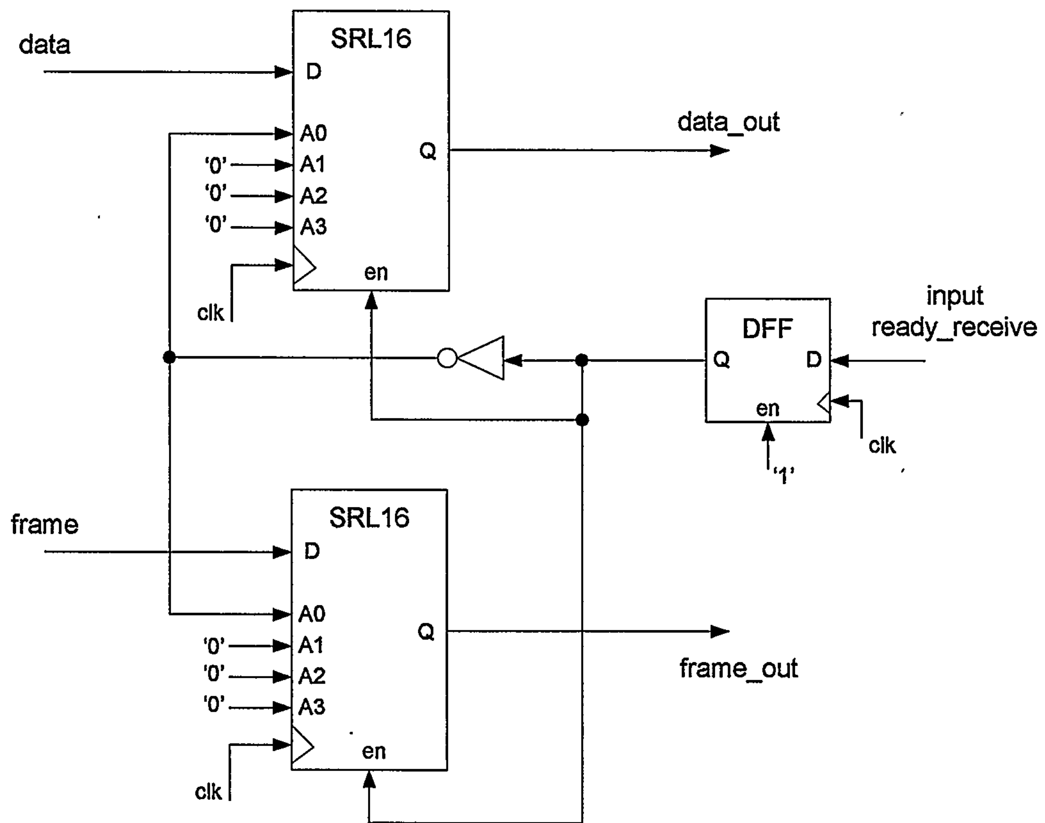


Figure 3.19: SRL16 Implementation of the 2-Bit Input Buffer

to form larger processing elements to implement more inputs. Because timing is regulated at the input by the controller, the frame signals for both inputs will be matched when they arrive at the processing unit, and only one frame signal needs to be passed on to the processing unit. A general structure of a two-input processing element is shown in Figure 3.20.

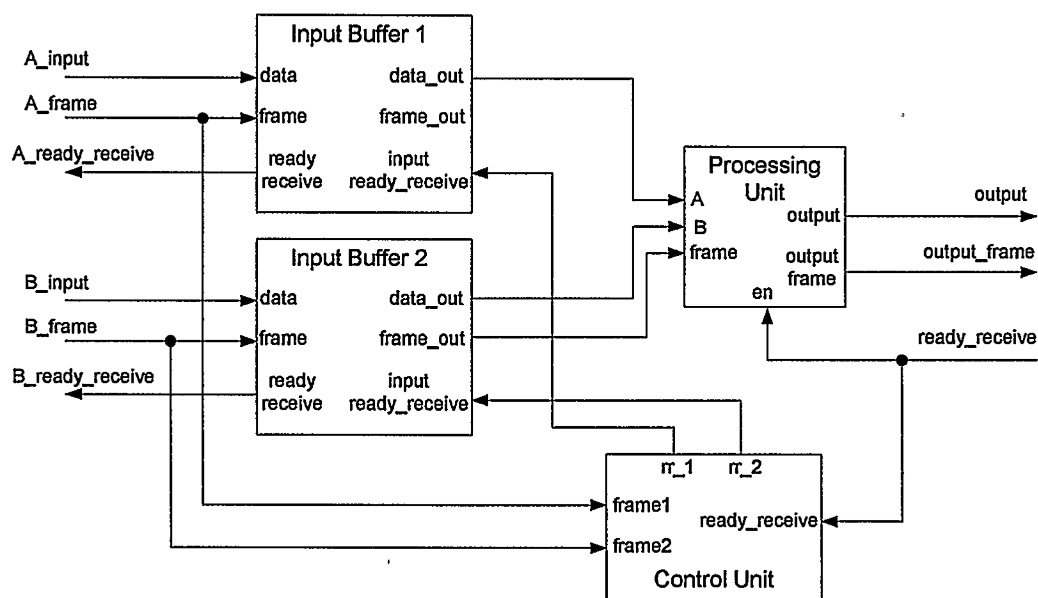


Figure 3.20: General Structure of a Two-Input Processing Element in Architecture Three.

One of the restrictions imposed on this self-timed architecture is that all of the inputs must be serial and no parallel loading of coefficients or data are allowed. This significantly simplifies the control structure, and allows a common control unit design to be shared between processing elements having the same number of inputs. As shown in Figure 3.20, the control unit will only be responsible to regulate the inputs, which further simplifies its structure.

The control unit consists of a three-state one hot encoded state machine. The

three states represent the three modes of operation for the inputs: both channels transmitting, A paused and B transmitting, and B paused and A transmitting. The state where both channels are transmitting is the reset state. If data in any channel arrives earlier than the other, the state machine will jump to a state where that channel is paused, and wait for the arrival of the LSB on the other channel. The condition where both channels are paused is an overwriting condition controlled by the output's *ready_receive* signal and is not handled by the state machine. Figure 3.21 gives a graphical illustration of this state machine controller.

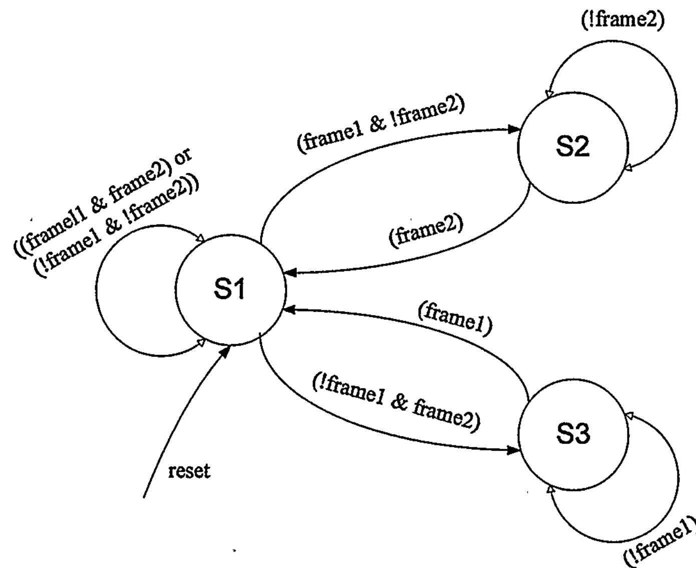


Figure 3.21: State Machine Controller for Two-Input Processing Element in Architecture Three.

Adder/Subtractor

The most common two-input processing element is the adder/subtractor. For this architecture, the processing unit section of the adder/subtractor is fitted with the static-timed bit-serial adder/subtractor shown in Figure 2.4 and 2.5.

Multiplier

The Multiplier is slightly different for this architecture. Because every input must be serial, parallel loading of the multiplier coefficient is not allowed. Therefore, a serial by serial multiplier must be designed. Referring to Figure 2.13 of the constant word length multiplier, to calculate bit X_n of the product, only bits B_0 to B_n of the coefficient are needed, and it is possible to achieve serial by serial multiplication by sequentially loading the coefficient into the multiplier one bit at a time during data processing, which can be achieved by connecting the serial input of the coefficient to a series of latches. The *enable* signal of these latches are controlled by the *frame* signal propagating down a chain of flip flops, loading one bit of the coefficient per clock cycle. The T_0 to T_{n-2} signals for clearing the carry storages in the multiplier structure can be used for this purpose. The structure of this modified serial by serial constant word length multiplier is shown in Figure 3.22.

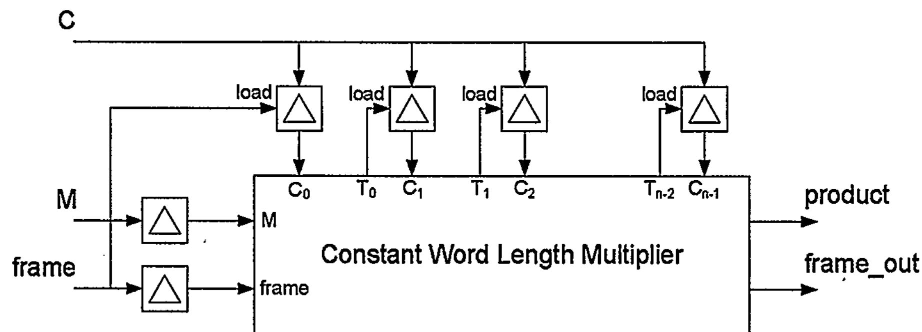


Figure 3.22: Serial by Serial Constant Word Length Multiplier Structure.

As discussed in Section 2.2.4, the latency of the constant word length multiplier is dependent on the width of the coefficient because the product is right shifted to keep only the n MSB bits, where n is the word length of the multiplicand. This is beneficial for digital filter applications, as filter coefficients are rational fractions,

and can be implemented with an integer multiplier followed by a shift right operator. Using this multiplier, the multiply and shift operation are done in one stage, so a separate shift right operator is no longer needed.

3.5.4 Multiplexer and 3-Input Elements

If a conditional control structure exists in a signal processing system, every branch of the control structure is built into hardware, and their outputs are sent to a multiplexing unit controlled by a select signal. Figure 3.23 provides a graphical illustration of this process.

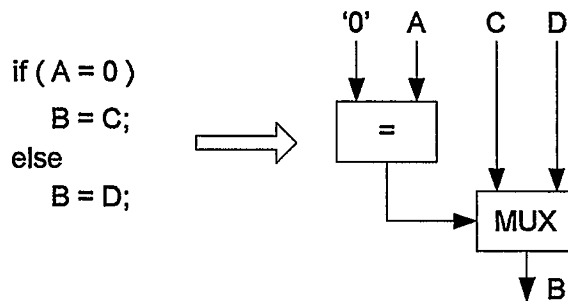


Figure 3.23: Control Structure Example.

For the self-timed architecture, the multiplexing unit must also be self-timed, where all of its inputs must follow the rules of self-timing. Therefore, a control unit must be capable of handling more than two inputs if necessary. For a two-input multiplexer, the control unit must be able to regulate three inputs as the select line is also treated as an input. The complexity of the control unit increases as the number of inputs increases. The state machine controller for a three-input processing element is seven states, compared to only three states for two-input. Fortunately, multiplexers with more than two inputs can be built by cascading multiple two-input

multiplexers, so controllers for four or more inputs are not needed. Figure 3.24 gives a graphical illustration of the state machine controller for three inputs. The frame signals from three different inputs are combined to form a vector to provide a clear representation on the diagram, where X="don't care".

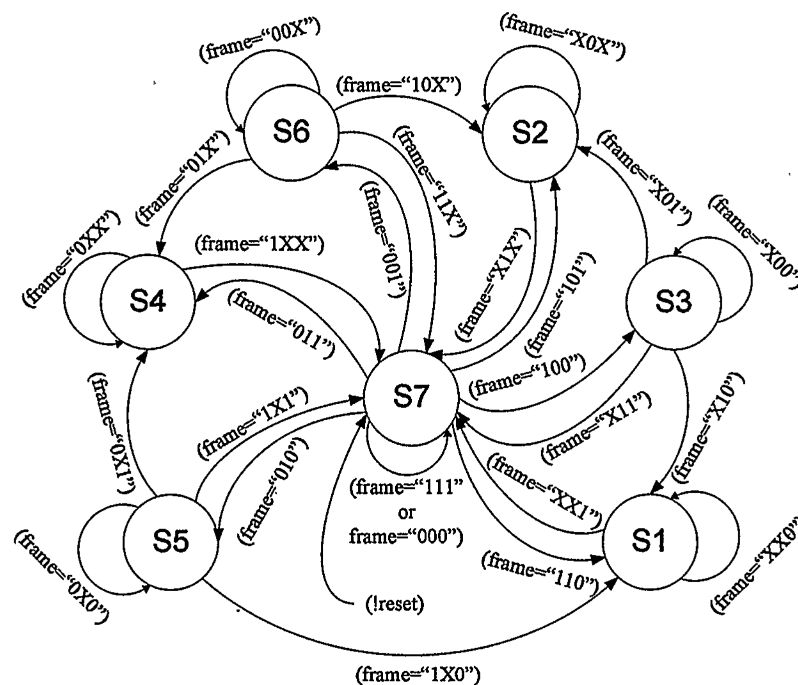


Figure 3.24: Structure of the State Machine Controller for 3-Input Processing Elements in Architecture Three.

The processing unit section of the multiplexer is designed to assume that all of the inputs arrive at the same time. When the frame pulse is detected, it will load the LSB of the *select* input, and use it to control which of the inputs will be sent to the output. The structure of the processing unit is shown in Figure 3.25.

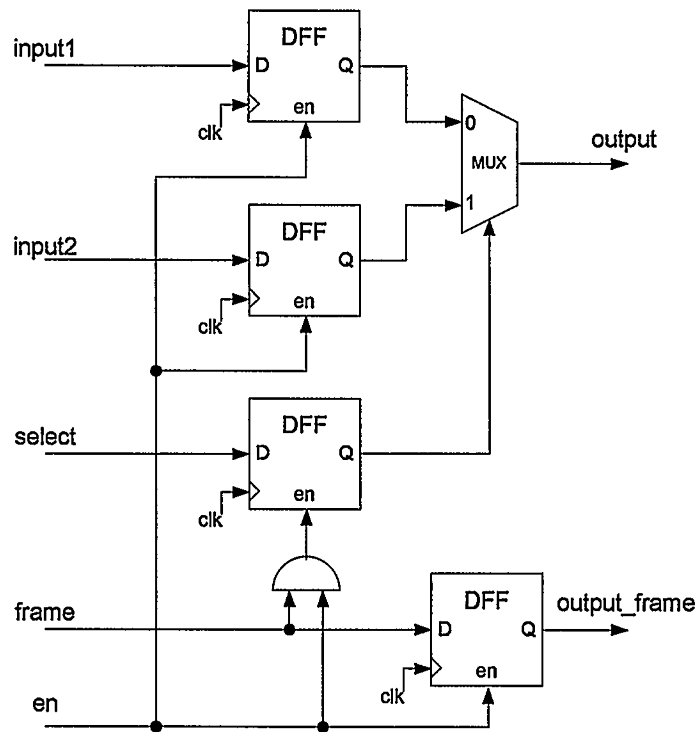


Figure 3.25: Structure of the Multiplexer in Architecture Three.

3.5.5 Fork/Delay Elements

For the previous two self-timed bit-serial architectures, the fork and delay element were always separated. However, it was realized that a delay element is always necessary if two of the branches have different timing characteristics, so the fork and delay element can be combined to reduce hardware cost. The drawback of this design is that since only one delay element will be used, the designer must know which branch will have a longer latency in advance. The frame signal for every word will also be stored and propagate through a shift register in the fork/delay element, eliminating the need for a pointer to separate one system word from another and further reducing hardware usage.

The fork/delay element works by storing and propagating the data and frame signals down a shift register, using an address pointer to select the location of the output. The address pointer is incremented and decremented by an address counter unit, and the control signals for both the address counter unit and the shift registers are generated through combinational logic manipulations of the input frame signal and output's *ready_receive* signals. A state machine controller is not necessary for this processing element. The structure of the fork/delay element in shown in Figure 3.26.

3.5.6 Summary of the 2-Bit Buffer Architecture

The 2-Bit Buffer Architecture addresses the shortcomings in the previous two architectures. It reduced the hardware resource requirement of each processing element by simplifying the control unit. Due to the restrictions imposed on the input and output format, the control unit design can also be shared by processing elements

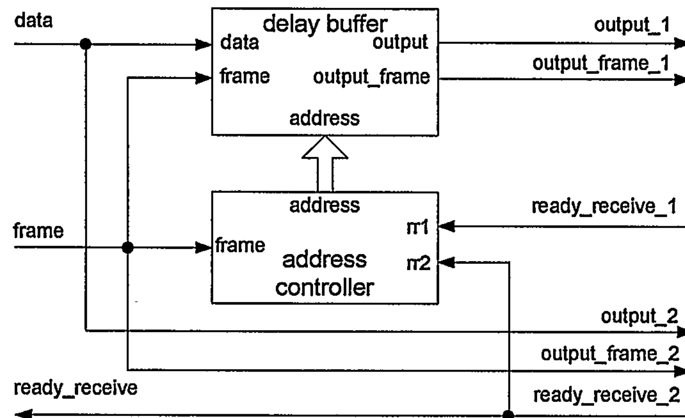


Figure 3.26: Structure of the Fork/Delay Element in Architecture Three.

with the same number of inputs. The hardware cost of the entire system was also reduced because the complex fork and delay elements are replaced by a much smaller fork/delay element. In terms of speed, systems designed with this architecture can achieve the same throughput as static-timed bit-serial systems.

The restrictions imposed on this architecture limited the input and output format, making processing element design not as flexible as in the first architecture. However, the effect of these restrictions are negligible, as virtually all processing elements in the digital signal processing domain can be modified to meet these restrictions. The size and speed estimation of each processing element in this architecture is shown in Table 3.3.

	LUTs	Flip Flops	Max Frequency (MHz)
Adder/Subtractor	15	10	173.010
Multiplier (16-bit coefficient)	75	84	84.289
Fork/Delay Element (32-bit word length)	18	5	136.893
Multiplexer	56	17	136.240

Table 3.3: Size and Speed Estimation of Processing Elements in Architecture Three.

Chapter 4

Floating Point Application

In recent years, the hardware resources and clock rate of modern FPGAs have rapidly increased. Current FPGA devices are reaching hundreds of thousands of logic cells, and can achieve clock frequencies up to 500MHz, making them a cost effective platform for real-time signal processing. Many digital filter systems are designed and implemented on FPGAs to achieve the sampling rate required for the application.

Two different arithmetic techniques dominate the signal processing domain, and they are fixed point arithmetic and floating point arithmetic respectively. Fixed point arithmetic, where the number of digits after the decimal (or binary or hexadecimal) point is fixed, is favored for FPGA or ASIC implementations because it can be easily implemented and consumes less hardware resource. Floating point arithmetic, on the other hand, does not have a fixed decimal point and is mostly used in general purpose digital signal processors. Other more exotic number systems, such as the residue number system, are geared toward specialized applications and are not addressed in this chapter.

One area that is now being addressed in FPGA systems is the implementation of floating point calculations. Many applications in the signal processing domain, such as radar/sonar signal processing, image processing, and molecular analysis, require a large dynamic range, and high accuracy [28]. These requirements are difficult to meet with fixed-point arithmetic, which makes floating-point units a useful addition to FPGA systems. However, current FPGAs are optimized for fixed-point calculations,

and the limited number of available floating point cores, although very good in terms of performance, are expensive both in price and hardware size requirement [29][30]. For an application such as audio signal processing where the sample rate is slow compared to the maximum clock speed of the current technology, it could be beneficial to sacrifice unneeded performance for reduced hardware size. Since the cost of a FPGA grows exponentially as the number of logic cells increases, it is desirable to have a system that has the smallest gate count while still meeting the throughput requirement. In the past, research has been done on implementing sections of a floating-point unit in bit-serial to reduce hardware usage [31]. In this chapter, a complete bit-serial implementation of a floating-point unit is presented.

A bit-serial system is smaller than a bit-parallel system because only enough hardware to process one bit at a time is needed. The interconnections between bit-serial modules are also fewer and potentially shorter compared to bit-parallel designs, resulting in reduced routing resource usage, which is especially important in systems where the routing resource is limited, such as in a FPGA. Since power consumption is directly proportional to the switching capacitance of the system, shorter interconnect paths may lead to reduced interconnect capacitance and lower power consumption. However, a bit-serial system does have its own limitations. Since the system is static timed, data streams must arrive at a processing block at the same time, in order to produce the correct output. Delays are added to the circuit to achieve the corrected timing, but these delays are static and cannot change during runtime. Unfortunately, such a dynamic timing characteristic is needed for the mantissa alignment operation in a bit-serial floating-point adder.

In this chapter, the advantages and drawbacks of current floating-point units

on FPGAs are explained in detail, and a solution to the problems is presented. The solution is to build Floating Point Units (FPUs) using a self-timed bit-serial architecture discussed in Chapter 3. To show the feasibility of this system, a floating-point adder is implemented and tested using the 2-Bit Buffer Architecture. Although the resulting performance lags behind its commercial bit-parallel counterparts, the system size is significantly smaller and the throughput is suitable for low sample rate applications such as audio signal processing. All the designs and tests in this chapter are targeted at the Xilinx Virtex2Pro architecture, and the area and timing estimations are generated by the Xilinx synthesis tool.

4.1 Floating Point Background

4.1.1 Floating Point Number Format and Addition Algorithm

A floating point number has four components: the sign, the significand s (also referred to as mantissa), the exponent base b, and the exponent e. The exponent base b is usually implied and not explicitly represented. Equation 4.1 shows how a floating point number, x, can be represented by the four components. A graphical illustration is shown in Figure 4.1.

$$x = \pm s \times b^e \tag{4.1}$$

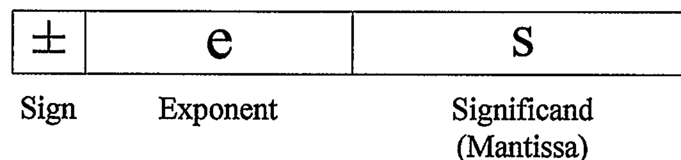


Figure 4.1: Typical Floating-Point Number Format

The sign bit indicates whether the floating point number is positive or negative; a value of zero indicate the number is positive, otherwise the number is negative. The exponent is stored using a biased representation, where a positive bias value is added to all exponents. Normalized values are typically used for mantissa representation, accomplished by shifting the mantissa to the left until the MSB becomes nonzero. In radix 2, the fixed leading 1 can be removed to save one bit; this bit is known as the "hidden 1". In the IEEE floating-point standard, a single precision floating point number is 32-bit in length. 23 bits plus one hidden bit are used to represent the mantissa, and 8 bits are used for exponent.

Addition and subtraction are the most difficult of the elementary operations in a floating-point system. Consider the addition shown in Equation 4.2.

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = \pm s \times b^{e} \tag{4.2}$$

if *e1* and *e2* are different, the mantissa of the number with the smaller exponent must be shifted to the right until it is aligned with the other number. Assuming *e1*>*e2*, the second number involved in the addition must be modified according to Equation 4.3. This step is referred to as alignment shift, or pre-shift.

$$\pm s2 \times b^{e2} = \frac{\pm s2}{b^{e1-e2}} \times b^{e1} \tag{4.3}$$

When the exponents of the two numbers are aligned through alignment shifts, addition can be carried out for the mantissas as shown in Equation 4.4.

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \pm \frac{s2}{b^{e1-e2}}) \times b^{e1} = \pm s \times b^{e} \tag{4.4}$$

If the sign bit values are the same for both operands, a single bit normalizing

shift is always enough. However, when the operands have different signs, the result may have to be shifted to the left by many positions for normalization.

It is the shift operations, particularly the alignment shifts, that make a bit-serial floating point adder difficult to implement because the latency for a shift right operator depends on the number of bits to shift, and can change when the input changes. The self-timed bit-serial architecture solves this problem by dynamically adjust timing during run-time.

4.1.2 Bit-Parallel Floating-Point Units

Many academic and commercial floating-point cores have evolved over the past few years [29][30][32]. Most of these cores are implemented with a bit-parallel architecture employing resources such as fast carry chain adders and embedded multipliers. There is also research on pipelining each stage of the processing to improve performance at the expense of hardware resources and latency [28]. While these floating-point cores have high throughput, they are also large. The floating-point adder requires hardware expensive shifters to accomplish pre-shifting and post-normalization, which occupies a large number of slices on the FPGA. If the mantissa word length is represented by w, each shifter will take at least $w \cdot log_2(w)$ two-input multiplexers. Additionally, the floating-point multiplier requires one or more embedded multipliers to implement, which are also a limited resource on an FPGA.

In applications where the real-time sample rates are low, such as audio applications, extremely high throughput is not a requirement. For example, if the audio sample rate is 48KHz, but the hardware is capable of running at 200MHz, the hardware must be clocked down to match the sample rate. In this example there is a

factor of more than four thousand between the maximum throughput and the required throughput. Therefore, it is advantageous to design floating-point units that have a lower maximum throughput but are much smaller. Another reason for designing smaller but slower FPUs is that in a typical signal processing system, where multiple parallel data paths exist, only a few FPUs will be on the critical path. For non-critical path units, it is beneficial to trade off unit performance for reduced FPGA resource usage.

Another flaw in current FPU architecture on FPGAs is high latency. To achieve the 100MHz+ performance in their design, the hardware is deeply pipelined, which introduces very high latency. For a well known commercial FPU producer, Nallatech [30], the latency for a single precision adder and multiplier is 14 and 6 clock cycles respectively. In applications where data dependencies occur after long and definite intervals, high latency is not an issue. However, if data dependencies exist in short loops, then the high latency could be an overall performance bottleneck. This kind of recursion is fairly common in many signal processing applications. For example, for a first order direct form filter described by equation $y(n)=a \bullet y(n-1)+x(n)$, $y(n)$ is dependent on $y(n-1)$. Therefore, processing for the next iteration can not start until the result from the previous iteration becomes valid. If the adder and multiplier have a 20 clock cycle latency in total, then the performance will be slowed down by a factor of 20, as compared to the system clock rate.

For the above reasons, it is desirable to develop a floating-point architecture which consumes less resource area and has shorter system word latency, with an acceptable drop in performance. Therefore, a bit-serial implementation is explored.

4.2 Self-timed Bit-serial Floating Point Adder

Even though a bit-serial implementation of a FPU seems attractive, static-timed bit-serial architectures has a fundamental flaw that makes such implementation impractical: static timing. Traditionally, delays are added to the bit-serial system to control the timing, but these delays are static and can not change during the runtime. For certain processing elements, such as a variable shift right operation, the latency is depended on the input (please refer to Section 2.2.3), and in the worst case situation, the latency will be a system word length long. This kind of variable latency introduces variable timing into the system, which requires the delays in the system to change dynamically during runtime. Such timing characteristics cannot be implemented using the traditional approach, but are needed for mantissa alignment and post-normalization in a floating point adder. Therefore, a self-timed bit-serial architecture discussed in Chapter 3 is needed. To achieve a balance between hardware size and performance, the 2-Bit Buffer Architecture, described in Section 3.5, was chosen for the floating point adder implementation.

4.2.1 Right Shift Operator

With the new self-timed bit-serial architecture, it is possible to implement processing elements with variable latency, with an example being a variable shift right operator. In floating point addition, a shift right operator is important, because if the two inputs have different exponents, the mantissa of the smaller input has to be shifted right until the two exponents match each other before addition can begin. The amount of the shift is dependent on the difference between the two input expo-

nents, and can change from calculation to calculation. To implement the shift right operator, the shift value is first loaded into a register with parallel output. After that, this value will be used to set a counter to control the amount of truncation of the input data. The structure of the shift right operator is shown in Figure 4.2.

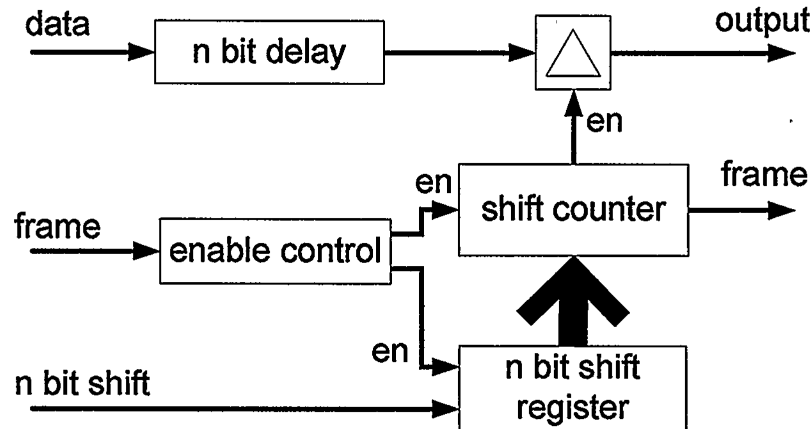


Figure 4.2: Structure of a Shift Right Operator

To implement the pre-shifting algorithm for floating point addition, a mantissa swap operator is used to swap the two-input mantissa if necessary. A variable shift right operator is placed on the input that needs to be shifted, and a variable delay buffer is placed on the other input. As the latency of the shift right operator changes, the delay on the other branch will adjust to accommodate the new timing. This timing adjustment on the delay buffer will be controlled by the handshaking unit of the self-timed bit-serial adder. The structure of this system is shown in Figure 4.3.

4.2.2 Adder Structure

With all the necessary components and a reliable timing scheme established, the implementation of a self-timed bit-serial floating-point adder becomes practical. A

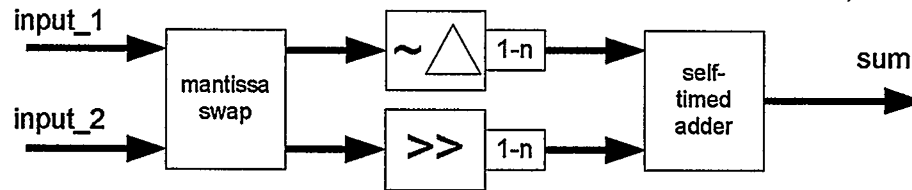


Figure 4.3: Structure of the Mantissa Alignment Shift.

single precision floating point adder, following the IEEE754 format, is implemented to demonstrate the feasibility of the self-timed bit-serial architecture. Denormalized numbers and NaN are not provided in this implementation, as they are rare and do not justify the hardware resources used for their handling. To further simplify the system, only one rounding mode, round to zero, is provided. To maximize performance, the exponent and mantissa are treated as separate signals, and are processed in parallel. The mantissa word length for single precision floating point is 23-bit. For this system, the leading one, guard, round, and sticky bits are added to the mantissa, making it 27 bits in total. The exponent is also scaled to 27-bit to achieve a common word length throughout the system. The detail of the floating-point addition algorithm is discussed in Section 4.1.1, and the basic steps are outlined as the following.

1. Calculate the absolute difference of the input exponents.

2. Complement and swap the two input mantissas as needed.

3. The mantissa of the input with smaller exponent is shifted right until its exponent matches the other input.

4. Add the mantissas.

5. Normalize the sum.

6. Round and complement the normalized sum as needed.

7. Adjust the exponent.

A signal flow graph displaying the above procedure is shown in Figure 4.4. The number at each output represents the clock cycle latency. The system before the alignment shift operation can be static-timed, but any processing dependent on the result from the alignment shift must be implemented using the self-timed architecture, as the timing can change dynamically.

4.3 Adder Performance Comparison

The resulting floating-point adder occupies 112 slices of the Virtex XC2VP2 FPGA, 8% of the total hardware resource of that particular FPGA, significantly smaller than the bit-parallel implementation. The maximum clock frequency of this system is approximately 250MHz. Since the word length is 27 bits, the throughput is 250/27=9.26 MFlops, which is slower than the bit-parallel implementation, which is expected, as the goal of this research is to reduce hardware size. An important characteristic of this floating-point adder is that the worst case latency is only 75 clock cycles, which is less than a three word cycle latency, representing a significant improvement over the pipelined bit-parallel design. A table comparing the bit-serial floating-point adder with two commercial products is shown in Table 4.1.

To implement a first order direct form IIR filter, an adder and a multiplier are needed. A floating point multiplier was not implemented for this research, but a

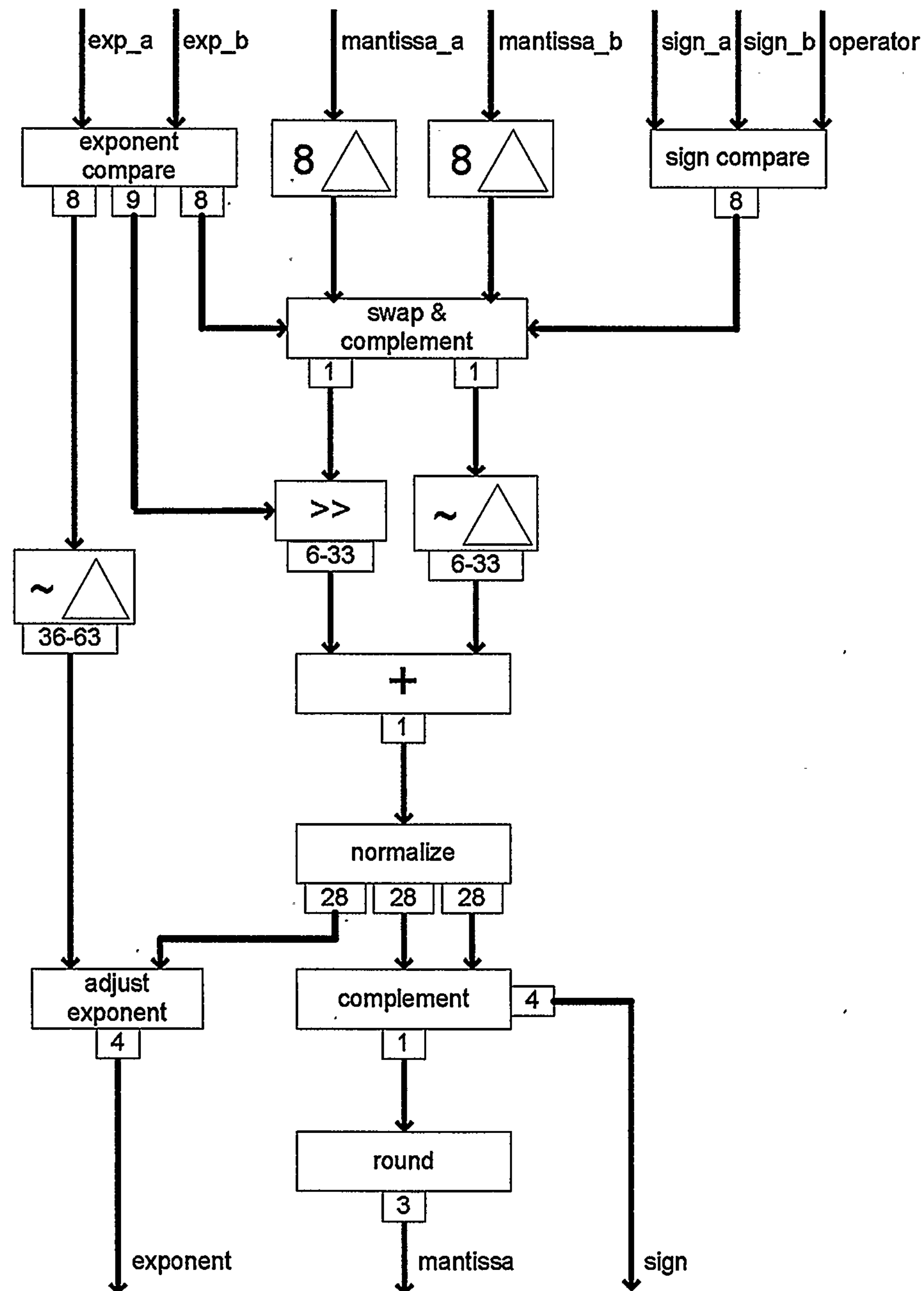


Figure 4.4: Self-Timed Bit-Serial Floating Point Adder Structure and Latency.

	Bit-serial Adder	Nallatech	Quixilica
Area (slices)	112	290	365
Throughput (MFlops)	9.26	184	188.1
Latency (word)	2.78	14	10

Table 4.1: Comparison of Single Precision Floating Point Adders.

theoretical estimation shows that it would occupy approximately 148 slices, can be clocked at 250MHz, and have a latency of one system word. With the self-timed bit-serial architecture, this filter can be implemented with approximately 260 slices, with a maximum throughput of 2.5MFlops, which is 52 times faster than the speed requirement of an audio application with sample rate of 48KHz. Although this system is much larger than a fixed-point implementation, it has the advantages of a lower quantization error and a larger dynamic range.

4.4 Self-Timed Bit-Serial Floating Point Unit Summary

As FPGAs increase in slice count and clock rate, their roles in signal processing systems are becoming increasingly important. For applications where large dynamic range and high accuracy are required, floating-point arithmetic is preferred over fixed-point. Current floating-point cores are focused on performance, using deeply pipelined bit-parallel architectures that occupied a large portion of the FPGA resource area. However, that performance may be unnecessary as some applications have a real-time throughput requirement far lower than the maximum throughput of the FPU, such as in audio applications. For those applications, it is beneficial to reduce the hardware size at the expense of performance. Bit-serial implementations can significantly reduce the resource area required, but the dynamic timing

characteristics of a floating-point adder make the implementation impractical. The self-timed bit-serial architecture solves the dynamic timing issue, and implements the adder in a third of the size of its commercial counterparts. Although the resulting throughput is lower than pipelined bit-parallel implementations, it still meets the sample rate requirement for audio applications. This result demonstrates that a bit-serial implementation of FPUs is not only feasible, but also beneficial in situations where high throughput is not a requirement.

Chapter 5

Speech Synthesizing Vocoder Application

5.1 Rsynth Text to Speech Vocoder

Text to speech systems are digital or analog systems where stored text can be converted to intelligible speech. Rsynth [33] is a text-to-speech synthesis system that uses a modified version of the Klatt formant synthesizer [34]. This synthesizer was developed at the Massachusetts Institute of Technology in the early 1960s, and later made into a complete system called MITalk. The Klatt formant synthesizer is a synthesis by rule system where the sound of speech is created by modeling the acoustic frequency spectra of the pronunciations of each word. Mathematical models are used to simulate the trajectories of speech parameters such as its fundamental pitch contour or its spectral content.

In systems where the Klatt formant synthesizer is used, speech synthesis is accomplished in two stages. In the first stage, an excitation waveform, such as a pulse train or noise source, is generated, and in the second stage, the excitation waveform is filtered to produce the correct speech output. The excitation waveform is a broadband waveform. To convert this waveform into recognizable speech, a series of 2nd-order digital filters, called resonators, are used to shape each resonant peak, or formant, in a speech sequence. The Klatt synthesizer uses both cascade and parallel configurations of the filters to accomplish this task. The Rsynth text to speech system is open source and can be freely modified, making it an ideal choice for this

project.

5.1.1 Advantages in Bit-Serial Implementation

Bit-serial implementation of the vocoder requires significantly less hardware resource when compared to a full bit-parallel implementation. In terms of performance, the bit-serial implementation will be a factor of SWL times slower at the same clock frequency. However, since the vocoder operates at 8KHz, much slower than the 500+ MHz performance of the FPGA, the clock frequency can be increased until the desired throughput is met. Using the 2-Bit Buffer Architecture, the vocoder structure can be implemented with ease. Processing elements can be connected to each other to form the desired structure, without worrying about scheduling.

5.1.2 Vocoder Structure

The Rsynth text to speech system is a complete system with all the necessary databases, excitation sources, and filters. To implement the entire system in hardware would be impractical as the resources on the available hardware are limited. For this project, only the vocoder section, the second stage of the synthesizing system with seven 2nd-order digital filters, is considered for implementation using the self-timed bit-serial architecture. To this end, the vocoder must first be isolated and extracted from the Rsynth system.

In the Rsynth system, a function called "parwave" in "nsynth.c" file is the formant synthesizer responsible for the filtering of excitation sources. In 10ms intervals, a frame of parameters is received by this formant synthesizer, which will change its filter coefficients and produce the next 10ms of speech output. The structure of the

vocoder is shown in Figure 5.1 [34].

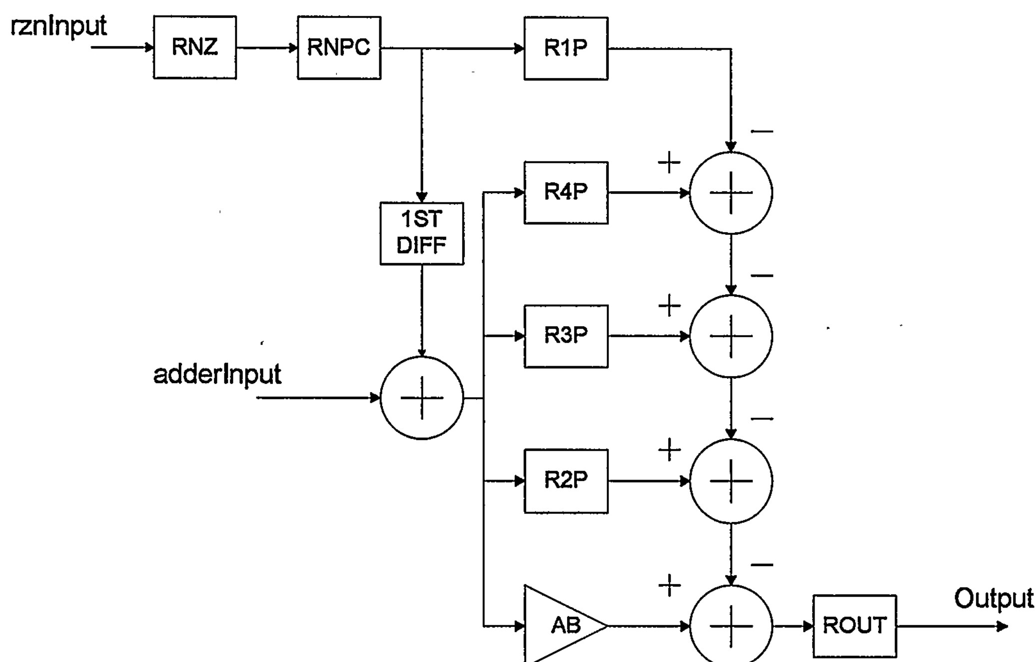


Figure 5.1: Structure of the Vocoder.

In Figure 5.1, four 2nd-order digital resonators are used to inflect oral cavity resonances upon the source waveforms: *R1P*, *R2P*, *R3P*, and *R4P*. The resonances of the nasal cavity are applied to the excitation source with anti-resonator *RNZ* and resonator *RNPC*. The other 2 filters in the vocal tract are a 1st-order differentiator and a 2nd-order lowpass filter, *ROUT*. *AB* is an amplitude bypass gain. The *rznInput* is the input to the cascade section of the filters, which represents the excitation waveform generated in the larynx, and the *adderInput* is the input to the parallel section of the filters, used to produce fricative sounds, such as the 'z' in "zebra".

5.2 Self-Timed Bit-Serial Implementation

To implement a functional vocoder that is capable of generating comprehensible speech, an entire system, rather than just a vocoder, needs to be implemented. The vocoder section is implemented using a self-timed bit-serial architecture, while the rest of the system, such as the audio codec interface, excitation source, and coefficient storage are implemented using traditional bit-parallel approach. A parallel to serial converter unit, specially designed to generate outputs compatible with the self-timed bit-serial architecture is placed at each output of those units.

The 2-Bit Buffer Architecture will be used to implement the vocoder section of the system, with the target FPGA being the Xilinx VirtexE 1000 chip. A block diagram of the system is shown in Figure 5.2.

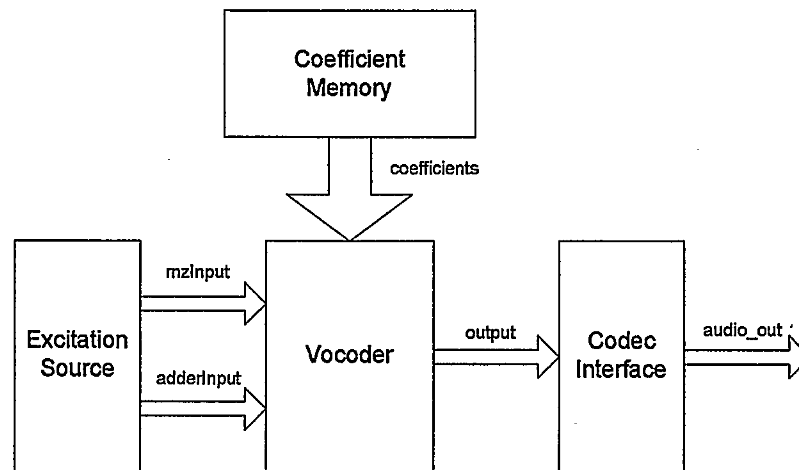


Figure 5.2: Structure of the Vocoder System.

5.2.1 Finite Precision Implementation

The Rsynth system was originally developed for use on desktop computers where all of the signals and filter coefficients are represented using 32-bit floating-point numbers. For this project, it is necessary to convert the floating-point implementation of the vocal track into a finite precision implementation where all the signals are represented using two's complement fixed point integer values. This is because on a limited resource system, such as the FPGA available for this project, floating-point calculations are impractical to implement.

Finite precision implementation is a system where all the numbers are represented using integer values. The size of the signal word length will affect the internal dynamic range of the system. A signal word length of 16-bit can represent a range of numbers from -32768 to +32767 whereas a 15-bit word length can only represent -16384 to +16383. As a calculation in the DSP algorithm approaches the limits of the signal word length representation, there is a risk of overflow and wraparound. For example, if one is added to the number 32767 in a 16-bit 2's complement representation, the result will be -32768. In a speech synthesis system, a loud clicking noise will be audible every time a signal wraparound occurs.

To alleviate this problem, the signal word length in the vocoder must be increased to accommodate the largest input and the worst case filter coefficient. Because the filter coefficients and the inputs are constantly changing, an analysis of a variety of textual inputs to the Rsynth text-to-speech system must be performed to determine the worst case scenario. This analysis is not the primary focus of this project, so it was decided that an arbitrarily large internal signal word length of 32-bit, would

be chosen. As the excitation input of the system is limited to 16 bits, 32-bit word length should be sufficient to prevent overflow.

Coefficient quantization is also necessary. A second order recursive digital filter is a major component of the formant synthesizer. The coefficients of these filters are normally rational fractions. To implement the filters using integer arithmetic, the filter coefficients must first be scaled up by a factor of M, and after the calculation is complete, the result have to be divided again by M. The value M is kept at a power of two to simplify division. The word length of the quantized filter coefficients will affect the locations of the poles and zeros of the corresponding digital filter, as these coefficients can not be exactly represented. For example, a filter coefficient value of 0.9 represented and implemented using finite precision arithmetic with a 8-bit M value is $\text{round}(0.9*2^8)/2^8$, which equals 0.8984375. The difference might seem to be insignificant, but it is enough to displace the locations of the poles and zeros and create distortions in the speech output. If this difference moves the transfer function poles outside of the unit circle, it can also cause the filter to become unstable. This effect will diminish with increased filter coefficient word length. For this project, a M value of 512 was chosen to ensure stability, meaning that 10 bits are required to represent the coefficients in 2's complement.

5.2.2 Vocoder Implementation

As mentioned in section 5.2.1, the system is implemented solely in integer arithmetic. The 2-Bit Buffer Architecture described in section 3.5 is used for implementation.

In the finite precision version of the vocal tract, multiplications need to be performed in three steps. First, the coefficients need to be scaled up by a factor of

M, where M is a power of two. After that, the multiplication is carried out with the scaled coefficients. Finally, the result is divided by M to generate the properly scaled answer. Because M is a power of two, the division can be easily implemented with a right shift of $\log_2(M)$ bits. This process is demonstrated in Figure 5.3. Using the constant word length multiplier in the 2-Bit Buffer Architecture, the second and third step can be combined into one, because the constant word length multiplier will shift the result internally, outputting only the most significant SWL bits.

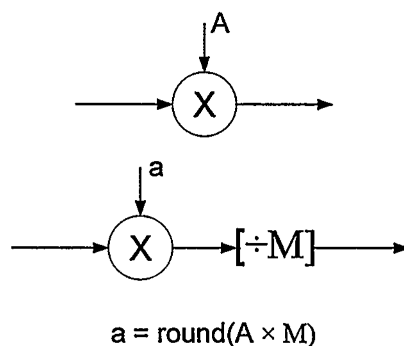


Figure 5.3: Coefficient Quantization Example.

The implementation of the vocoder structure is straightforward. Processing elements such as adders, multipliers, and fork/delay elements are connected together to form the components required. The components are then connected to form the vocoder structure shown in Figure 5.1. Figures 5.4, 5.5, and 5.6 provide a detailed view of the structural implementation of each component within the vocoder. The low pass filter *ROUT* in Figure 5.1 has the same structure as a resonator.

5.2.3 Coefficient Storage

A new set of coefficients is sent to the vocoder every 10ms, which are used to generate 80 audio samples, resulting in an audio frequency of 8Khz. The Rsynth system is used

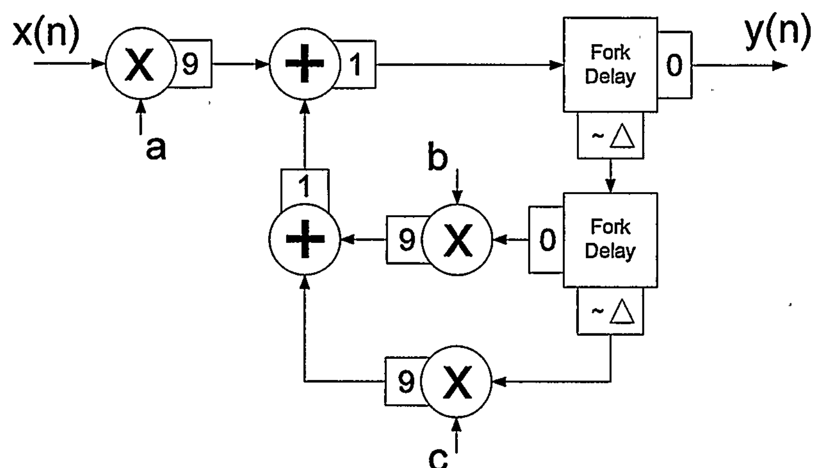


Figure 5.4: Resonator Structure.

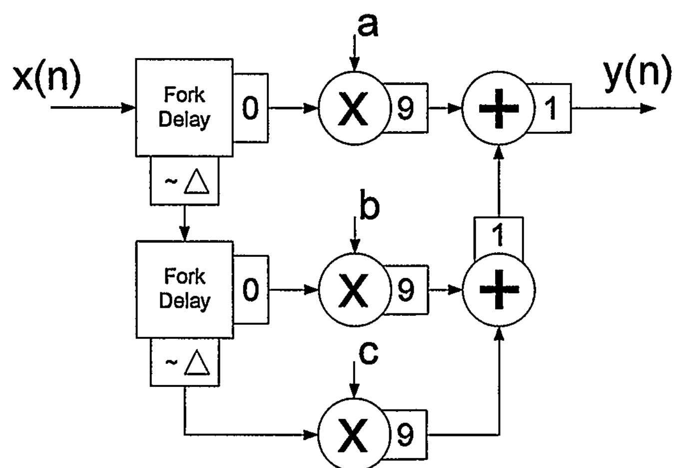


Figure 5.5: Anti-Resonator Structure.

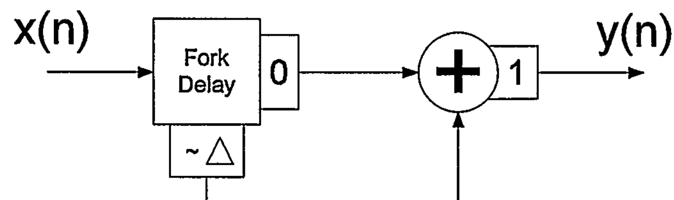


Figure 5.6: 1st Order Differentiator Structure.

to produce the coefficients, and is external to the vocoder system, so the coefficients must be stored and then send to the vocoder with the appropriate timing. This is accomplished by using the Block SelectRAM+ feature of the Virtex FPGA.

The block SelectRAM+ feature in Virtex series of FPGAs offers fast, discrete and large blocks of RAM, with 4096 memory cells in each block. They also have specialized routing to allow cascading multiple blocks with minimal routing delay. This feature achieves a wider or deeper ram structure with less timing penalty than using normal routing channels. For the vocoder system, each coefficient channel has a dedicated ram block, storing all the coefficients required for a particular speech sequence. Ten bits are required to store each coefficient. Since each set of coefficients is responsible for 10ms of output, a memory depth of 200 words is needed for each coefficient channel to generate a typical audio sequence of two seconds, which only requires one block ram module. When a new coefficient is needed, it will be loaded in parallel to a register first. A state machine will shift out the coefficient in accordance to the self-timed bit-serial data format. The block diagram for this module is illustrated in Figure 5.7.

5.2.4 Audio Codec

An audio codec is necessary for the implementation of the speech synthesis system if the digital signal produced by the processor is to be converted into meaningful sounds. It is also a necessary component during the testing and debugging stage as the audio codec can convert the digital signals into analog signals, which then can be viewed on an oscilloscope.

The audio codec chip used for this project is the AK4522VF chip manufactured by

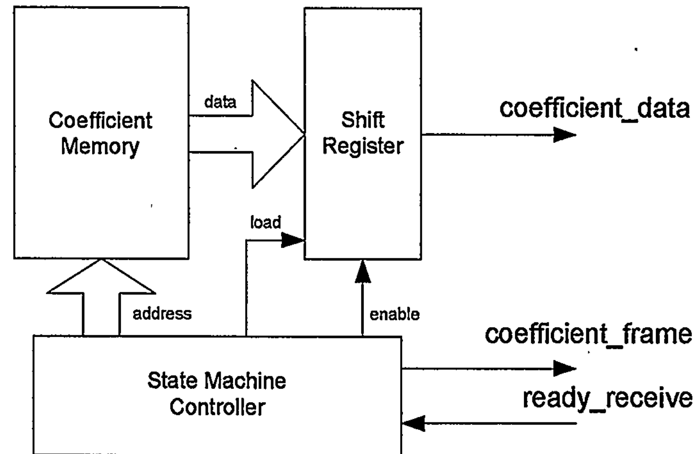


Figure 5.7: Coefficient Memory Interface to the Vocoder.

Asahi Kasei Microsystems. This chip has 20-bit precision and can handle sampling frequencies of up to 48 kHz, which is more than sufficient for the purpose of this project. There is one pin for serial data input and one pin for serial data output on this chip. These two input and output lines each handle two channels as the chip is meant for stereo operation. A frame signal is used to synchronize the input and output data. Even though the input and output to the codec chip is bit-serial, they can not be interfaced directly with the self-timed bit-serial vocoder, as their data formats are different. The bit-stream used by the audio codec is MSB first as oppose to the LSB first data format used by the self-timed bit-serial architecture. Therefore, it was necessary to first design and implement a framework that acts as a interface between the vocoder and the audio codec, as indicated in Figure 5.2. The vocoder output will be first shifted in and stored in a register, and then loaded in parallel to another register, which will shift out the data to the codec MSB first, in accordance to the codec's required format. The structure of the audio codec interface is shown in Figure 5.8

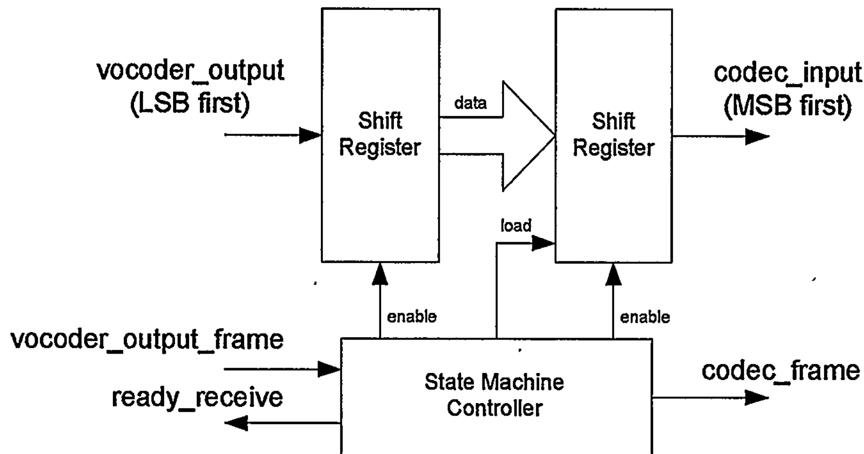


Figure 5.8: Audio Codec Interface.

5.3 Self-Timed Bit-Serial Vocoder Summary

The 2-Bit Buffer Architecture proves to be effective for the vocoder implementation. The vocoder section of the finished system occupied 2277 slices, 18 percent of the total slice count of the VirtexE 1000. Each slice in a VirtexE FPGA has two 4-input LUTs and two flip flops. This bit-serial implementation is much smaller than what a bit-parallel implementation would be. Since the VirtexE series FPGA does not have any embedded multipliers, a bit-parallel implementation would be roughly SWL times bigger than the bit-serial implementation. As the SWL is *32-bit*, an equivalent full bit-parallel implementation of the vocoder system would occupy approximately 72864 slices, or 576% of the hardware resource on a VirtexE 1000 FPGA. A more detailed description of the hardware resource usage by each section of the system is shown in Table 5.1. The estimated maximum clock frequency for the vocoder is 76.864Mhz. Since the system word length is 32-bit, this translates to a throughput of 2.4Mhz, 300 times faster than the 8Khz requirement for the audio

output. The self-timed bit-serial architecture made implementation straight forward. All the necessary components are connected to form the desired structure, with no timing calculations necessary.

	LUTs	Flip Flops	Block Ram	Max Frequency (MHz)
Vocoder	1853	1565	0	76.864
Excitation Source	330	39	32	105.108
Coefficient Memory	777	672	21	161.368
Codec Interface	80	60	0	181.225

Table 5.1: Self-Timed Bit-Serial Vocoder Summary.

Because there is a factor of 300 between the maximum throughput and the required throughput, further hardware resource saving is possible if the system is redesigned to time share it's components, such as the resonator. This time sharing can also be considered to be run-time reconfiguration, as data paths are changed during run-time. Run-time reconfigurations are made easy on self-timed bit-serial architectures, as any scheduling change as a result of data path change will be handled automatically. This characteristic makes it an attractive alternative to traditional design approaches, such as bit-parallel architectures or static-timed bit-serial architectures.

Chapter 6

Conclusion and Future Work

6.1 Thesis Summary

This thesis introduces self-timed bit-serial architectures that are capable of adjusting their own scheduling during run-time. This dynamic scheduling scheme has a number of advantages. First of all, the system will be easier to design and to scale up compared to the static-timed bit-serial approach because a centralized controller is no longer needed, and any scheduling change due to system modifications will be self-adjusted during run-time. The second advantage of a self-timed system is dynamic word length, as systems designed using a self-timed bit-serial architecture are no longer restricted to a fixed word length as in the case with the static-timed bit-serial approach. This could potentially lead to better average throughput, as the system can reduce its word length when possible. The third advantage is the possibility of run-time reconfiguration, where the system can dynamically change its data path and time share its components during execution.

Three different self-timed bit-serial architectures were proposed and developed. The Controller Regulated Architecture achieves self-timing through a state machine controller in each processing elements to regulate all input and output bit-streams. It allows flexible processing element design, but the fork and delay elements required a large amount of hardware resources to implement, resulting in large system size. The Full Word Buffer Architecture achieves self-timing by buffering the entire word

of all inputs before the processing would begin. It is significantly smaller than the first architecture, but the penalty on throughput is too great. The 2-Bit Buffer Architecture is a compromise between the first and second architecture, and achieves self-timing at a significantly smaller hardware resource requirement compared to the first architecture, and is able to maintain the same throughput as the static-timed bit-serial architecture at the same clock frequency. For this reason, it was chosen for both sample applications.

Two sample applications were implemented on a Xilinx FPGA to demonstrate the feasibility and benefits of self-timed bit-serial architectures. The first application is a floating point adder. The implementation of the floating point adder using the 2-Bit Buffer Architecture shows that systems with dynamic timing requirements, the kind of requirement that cannot be fulfilled by the static-timed bit-serial architecture, can be implemented using a self-timed bit-serial architecture. The components for the floating point adder are implemented and simulated with ModelSim version 6.0, and the size and performance estimates are produced by the Xilinx ISE Foundation design tool version 7.1. The second application is the speech synthesizing vocoder. Implementing the vocoder using the 2-Bit Buffer Architecture demonstrated that large systems can be implemented with ease using a self-timed bit-serial architecture, and that detailed knowledge of bit-serial arithmetic and timing is not need. This vocoder is implemented on the Xilinx VertexE 1000 FPGA. The size and performance estimates are also produced by the Xilinx ISE Foundation design tool version 7.1.

6.2 Future Work

The goals achieved in this thesis may pave the way for the future advancement of this technology. This is a field that has been neglected by most researchers, but has room for improvements. First of all, current self-timed bit-serial architectures need to be improved in term of robustness and functionality. Hardware saving through run-time reconfiguration should also be explored. Ultimately, a compiler should be written for a self-timed bit-serial architecture to simplify system implementations.

6.2.1 Architectural Improvements

Current self-timed bit-serial architectures, while adequate for simple systems, lack features for complex system implementations. Processing elements such as comparators, bitwise operators, and constant storage should be implemented to expand their potential applications. A variety of system structures should be implemented using self-timed bit-serial architectures to test their robustness, which is important to ensure that they are reliable enough in different situations. Situations that can not be handled by the architectures should be documented, so that they can be avoided by designers.

6.2.2 Run-Time Reconfiguration

Run-time reconfiguration refers to dynamic change in data path and component functionalities during execution, which can result in system size reductions, as hardware resources are shared in time. Self-timed bit-serial architectures make run-time reconfiguration possible in a system, because any timing change as a result of recon-

figuration will be automatically compensated for. Further research is need to exploit the usefulness of this characteristic.

6.2.3 Self-Timed Bit-Serial Compiler

Even though self-timed bit-serial architecture makes system implementation easier, it is still necessary to be proficient in VHDL to use them effectively. The ultimate goal would be to have a compiler that is capable of converting high level description of a system directly to a self-timed bit-serial implementation. As hardware compiler technology advances, hardware description language such as VHDL will yield to higher level languages, similar to the way assembly have yield to C and C++. With more research and development, self-timed bit-serial architectures, with their own set of unique benefits, will prove to be a useful alternative to the more well established approaches.

Bibliography

[1] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI Signal Processing*, vol. 28, pp. 7 – 27, 2001.

[2] P. Denyer and D. Renshaw, *VLSI Signal Processing: A Bit-Serial Approach.* Addison Wesley, 1985.

[3] R. I. Hartley and K. K. Parhi, *Digit-Serial Computation.* Kluwer Academic Publishers, 1995.

[4] T. Isshiki, A. Ohta, T. Watanabe, T. Nakada, K. Akahane, I. Sisla, D. Li, and H. Kunieda, "High density bit-serial FPGA with LUT embedding shift register function," in *Asia-Pacific Conference on Circuits and Systems*, 2002.

[5] A. Y. Carreira, "Parameterizable bit-serial FIR filter cores for field programmable gate arrays," Master's thesis, University of Calgary, 2002.

[6] R. J. Andraka, "FIR filter fits in an FPGA using a bit serial approach," in *3rd Annual PLD Conference*, 1993.

[7] S. G. Gibb, P. J. W. Graumann, and L. E. Turner, "FIR filter implementation using bit-serial arithmetic and partial summation trees," in *Proceedings SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations VI*, 1996, pp. 63 – 74.

[8] W. E. Donath, "Placement and average interconnection lengths of computer logic," *IEEE Transaction on Circuits and Systems*, pp. 272 – 277, 1979.

[9] R. Heaton, D. Blevins, and E. Davis, "A bit-serial VLSI array processing chip for image processing," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 364 – 368, 1990.

[10] C. B. Dieterich, T. J. Christopher, G. G. Tamer, P. D. Filliman, R. E. Morris, R. F. Nutter, R. J. Kolczynski, and D. R. McClary, "A complete audio decoder IC for U.S. television stereo using serial arithmetic," in *IEEE Custom Integrated Circuits Conference*, 1990.

[11] L. B. Jackson, J. F. Kaiser, and H. S. McDonald, "An approach to the implementation of digital filters," *IEEE Transaction on Audio Electroacoustics*, vol. 16, pp. 413 – 421, 1968.

[12] J. Valls, M. M. Peiro, T. Sansaloni, and E. Boemo, "A study about FPGA-based digital filters," in *IEEE Workshop on VLSI Signal Processing: Design andImplementation (SiPS'98)*, 1998, pp. 192 – 201.

[13] L. E. Turner, P. J. W. Graumann, and S. G. Gibb, "Bit-serial FIR filters with CSD coefficients for FPGAs," in *5th International Workshop on Field-Programmable Logic and Applications (FPL'96)*, 1995, pp. 311 – 320.

[14] H. Lee and G. E. Sobelman, "FPGA-based FIR filters using digit-serial arithmetic," in *Proceedings of IEEE International ASIC Conference*, 1997, pp. 225 – 228.

[15] A. Ohta, T. Isshiki, and H. Kunieda, "New FPGA architecture for bit-serial pipeline datapath," in *Proceedings.IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 58 – 67.

[16] M. Dimmler, A. Tisserand, U. Holmbeg, and R. Longchamp, "On-line arithmetic for real-time control of microsystems," *IEEE/ASME Transactions on Mechatronics*, vol. 16, pp. 213 – 217, 1999.

[17] M. Ercegovac and T. Lang, "On-line arithmetic for DSP applications," in *Proceedings of the 32nd Midwest Symposium on Circuits and Systems*, 1989, pp. 365 – 368.

[18] D. Lau, A. Schneider, M. D. Ercegovac, and J. Villasenor, "FPGA-based structures for on-line FFT and DCT," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, pp. 310 – 311.

[19] J. F. Wakerly, *Digital Design, Principles & Practices.* Tom Robbins, 2000.

[20] R. N. Schneider, "Hardware acceleration of the finite-difference time-domain (FDTD) method," Master's thesis, University of Calgary, 2002.

[21] J. Valls, M. Martinez-Peiro, T. Sansaloni, and E. Boemo, "Fast FPGA-based pipelined digit-serial/parallel multipliers," in *International Symposium on Circuits and Systems*, May. 1999.

[22] J. Valls and E. Boemo, "Efficient FPGA-implementation of two's complement digit-serial/parallel multipliers," *IEEE Transactions on Circuits and Systems*

II: Analog and Digital Signal Processing, vol. 50, pp. 317 – 322, June 2003.

[23] (2005) Andraka consulting group, inc. [Online]. Available: http://www.andraka.com/

[24] L. B. Jackson, *Digital Filters and Signal Processing, THIRD EDITION*. Kluwer Academic Publishers, 1996.

[25] A. Rettberg, M. Zanella, T. Lehmann, and C. Bobda, "A new approach of a self-timed bit-serial synchronous pipeline architecture," in *14th IEEE International Workshop on Rapid Systems Prototyping (RSP'03)*, June. 2003.

[26] R. Hogg, W. Hughes, and D. Lloyd, "A novel asynchronous ALU for massively parallel architectures," in *4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)*, 1996, p. 282.

[27] (2005) xilinx. [Online]. Available: http://www.xilinx.com/

[28] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, April. 2004.

[29] (2005) Quixilica. [Online]. Available: http://www.quixilica.com/

[30] (2005) Nallatech. [Online]. Available: http://www.nallatech.com/

[31] W. B. L. III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *6th Annual IEEE Symposium on FPGA for Custom Computing Machins (FCCM'98)*, April. 1998.

[32] P. Belanovic and M. Leeser, "A library of parameterized floating-point modules and their use," in *12th International Conference on Field Programmable Logic and Application (FPL'02)*, September 2002.

[33] (2005) Rsynth. [Online]. Available: http://www.speech.cs.cmu.edu/comp.speech/Section5/Synth/rsynth.html

[34] S. Jacobsen, "Digital filtering for speech synthesis in text-to-speech," in *ENEL 599 report*, 2001.