# High-Performance Bit-Serial Datapath Implementation for Large-Scale Configurable Systems

PhD Dessertation

Tsuyoshi Isshiki

April 13, 1996

**Abstract**

Custom computing machines based on FPGA devices have attracted much attention as a new approach toward high-performance computing. By combining state-of-the-art FPGA technology and state-of-the-art MCM packaging technology, a "configurable system" with a gate capacity of several millions of gates contained in a desk-top size box will become a reality in the near future.

Developing applications for such a large configurable system is a grand challenge. FPGAs are inherently resource limited devices in terms of logic, routing, and IO. Without a careful circuit implementation strategy, one would waste a large portion of the potential capacity of the configurable hardware. Also, high-level design entry support is essential for such large-scale hardware. A C++ design tool has been implemented which maps the computational algorithms onto bit-serial pipeline networks which exhibit high performance and maximize the device utilization of each FPGA. With this tool, the designer is able to develop applications in a very short time, and also is able to try out different algorithm implementations easily to see the trade-offs in terms of performance and hardware size instantaneously. Based on this C++ design tool, a number of DSP applications such as 1D and 2D filters, adaptive filters, Inverse Discrete Cosine Transform, and digital neural networks were designed. Results show that the bit-serial pipeline networks have a low Rent exponent of 0.22 to 0.37 which indicates high efficiency in terms of routing and IO consumption. This is confirmed by the layout results of the various bit-serial designs in which all the circuits were successfully routed even with the logic utilization as high as 100%.

# Contents

**Acknowledgements** ii

**1 Introduction** 1
- 1.1 Key Contributions . . . 2
- 1.2 Organization of the Thesis . . . 3

**2 Background** 4
- 2.1 High-Performance Computation Applications . . . 4
- 2.2 FPGA-Based Custom Computers . . . 5
  - 2.2.1 Field-Programmable Gate Arrays . . . 5
  - 2.2.2 Custom Computing . . . 7
- 2.3 Field-Programmable Multi-Chip Module . . . 9
  - 2.3.1 Multi-Chip Module Technology . . . 9
  - 2.3.2 Impact of MCM Technology on FPGA-based System . . . 9
  - 2.3.3 UCSC FPMCM-I . . . 10
  - 2.3.4 Altera EPF8050M FPMCM . . . 12
  - 2.3.5 FPMCMs for Large-Scale Configurable Systems . . . 12
- 2.4 Challenges of Application Development for Large-Scale FPMCM-Based Configurable Coprocessor System . . . 13
  - 2.4.1 Limitation of High-Level Synthesis Approach on FPGA Design . . . 14
  - 2.4.2 Resource Limitations and Low Performance of SRAM-based FPGAs . . . 15
- 2.5 Bit-Serial Datapath Synthesis on FPGA . . . 17
  - 2.5.1 Bit-Serial Vs. Bit-Parallel . . . 18
  - 2.5.2 Bit-serial Vs. Signed Binary Digits . . . 23
- 2.6 Datapath Design Capture in C++ . . . 24

**3 High-Speed Bit-Serial Circuit Design on FPGAs** 27
- 3.1 Principle of Bit-Serial Operations . . . 27
  - 3.1.1 Number Representation and Number Precision . . . 27
  - 3.1.2 Distribution and Timing of Control Signals . . . 28
  - 3.1.3 Operation Latency . . . 29
- 3.2 Detail Description of Bit-Serial Operators . . . 30
  - 3.2.1 IO Data Conversion . . . 30
  - 3.2.2 Addition and Subtraction . . . 32
  - 3.2.3 Bit Shifting . . . 32
  - 3.2.4 Multiplication . . . 35

3.2.5 Scaling . . . 39
3.2.6 Rounding . . . 41
3.2.7 Saturation . . . 42
3.3 Circuit Complexity of Bit-Serial Operators . . . 43

**4 C++ Bit-Serial Datapath Synthesis System 45**
4.1 Basic Concept of the Function-Based Hardware Design Capture . . . 45
4.2 Datapath Description Model . . . 46
4.3 Components of C++ Datapath Description . . . 47
4.4 Difference Equation Description Using Overloaded Arithmetic Functions . . . 49
4.5 Behavioral Design Verification . . . 50
4.6 Sequential Process Description . . . 51
4.7 Hardware Module Generator . . . 54
4.8 Bit-Serial Circuit Library . . . 55
4.8.1 Hierarchical Data Structure of Bit-Serial Pipeline Network . . . 55
4.8.2 Directed Hyper-Graph Representation . . . 55
4.8.3 Module-level netlist $HG(\overline{E_m}, V_m)$ . . . 58
4.8.4 Cell-level netlist $HG(\overline{E_c}, V_c)$ . . . 58
4.8.5 Logic-level netlist $HG(\overline{E_l}, V_l)$ . . . 58
4.8.6 Netlist Hierarchy and Circuit Element Labeling . . . 59
4.8.7 Hierarchical Netlist Generator . . . 60
4.9 Memory Interface Model . . . 64
4.10 Ultra-Fast Counter for Simple Control Flow and Memory Address Generation . . . 66
4.10.1 Arbitrary size single-level logic counter . . . 67
4.10.2 Arbitrary Radix Counter . . . 68
4.10.3 Line Buffer Implementation Using High-Speed Counters . . . 70
4.11 High Throughput Pipelined Datapath Structure . . . 71
4.12 Pipeline Synchronization . . . 72
4.12.1 Conversion from Directed Hyper-Graph to Directed Graph . . . 73
4.12.2 Critical Path Analysis and Minimum Sampling Period Calculation . . . 75
4.12.3 Pipeline Register Insertion . . . 77
4.13 Bit-Serial Control Signal Routing . . . 77
4.14 Pipeline Synchronization Register Minimization . . . 81
4.15 IO Scheduling . . . 87

**5 Design Examples Using the Bit-Serial Datapath Synthesis System 90**
5.1 Digital Filters . . . 91
5.1.1 1-D FIR Filters . . . 91
5.1.2 2-D FIR filters . . . 93
5.1.3 1-D Recursive Filters . . . 97
5.1.4 Comparison With Other Filter Designs on FPGAs . . . 99
5.2 Adaptive Filter . . . 103
5.3 2-D Inverse Discrete Cosine Transform . . . 111
5.3.1 Chang's DCT Implementation Using 14 Multiplications . . . 112
5.3.2 Two-Pass 2-D IDCT Implementation With Data Buffering at the Host Computer . . . 115
5.3.3 Two-Pass 2-D IDCT Implementation With Data Buffering at the Coprocessor Memory . . . 117
5.4 Digital Neural Network . . . 117

**6 Bit-Serial Pipeline Partitioning Method for Maximum Logic Utilization 122**
6.1 General Circuit Network Partitioning . . . 122
6.2 Problem Formulation for Bit-Serial Pipeline Network Partitioning for Multi-FPGA Systems . . . 124
6.3 Bit-Serial Pipeline Network Multi-Way Partitioning Methodology . . . 124
6.3.1 Basic Concept . . . 125
6.3.2 Fiduccia-Mattheyses Bipartitioning Method . . . 127
6.3.3 Details of the Recursive Bipartitioning Method . . . 128
6.4 Multi-Way Bit-Serial Partitioning Results . . . 130
6.5 Routability Analysis for Bit-Serial Circuits . . . 134
6.5.1 Rent's Exponent for Bit-Serial Circuits . . . 134
6.5.2 Physical Layout Results . . . 138

**7 Conclusion 149**
7.1 Summary . . . 149
7.2 Future Work . . . 151

**A FPMCM-I Architecture and Layout Synthesis Strategy 153**
A.1 FPMCM-I Architecture review . . . 153
A.2 FPMCM Layout Synthesis Strategy . . . 154
A.3 FPMCM Layout Flow . . . 157

**B Prototype Configurable Coprocessor System 158**
B.1 System Architecture . . . 158
B.2 Board Configuration and Operation . . . 160

**C SBus Driver and Software Issues 166**
C.1 SBus Addressing . . . 166
C.2 SBus Device Driver . . . 166

**D C++ code descriptions from Chapter 5 170**

# Acknowledgement

I would like to express my greatest appreciation to Professor Wayne Wei-Ming Dai who has encouraged and supported this thesis research from the very beginning. His highly inspirational insights, advises and discussions were invaluable in carrying out this research. Also I would like to thank Professor Glen Langdon for his image compression class where I first developed the idea of using bit-serial circuits on FPGAs, and Professor Kevin Karplus for his VLSI design class where I acquired the necessary digital design skills to pursue this thesis. My project in his class was the bit-serial multiplier which become the fundation of this thesis. Also, I would like to thank Professor Pak K. Chan for his class on ASIC designs where I was introduced to the FPGA technology for the first time. Also, I would like thank my colleagues working on the FPMCM project, especially Joel Darnauer, for their constant efforts into this exciting and long lasting research project. Without the help of these people, this thesis would not have been possible.

# Chapter 1

# Introduction

Field-Programmable Gate Array (FPGA) technology has emerged during the last decade as a new VLSI implementation technology, adding to the already existing custom chips and Mask-Programmable Gate Arrays (MPGAs).An FPGA is a multi-level programmable logic device in which its functionality can be programmed after the fabrication of the device [89]. Because of their low design cost, low non-recurring engineering (NRE) cost and short turnaround time, FPGAs are widely used in prototyping and also in the final products for low-volume markets. The most interesting development concerning SRAM-based FPGAs where the device can be reprogrammed multiple times is the new idea of constructing a high-performance computer using these FPGA devices. The main features of the so called "FPGA-based custom computing machines" or "configurable systems" are the drastic speed up achieved by the highly parallelized hardwired computations whose architecture is strictly customized to a specific application, and also the large flexibility of the SRAM-based FPGAs for implementing a wide variety of applications. There are a number of existing configurable systems where each have successfully demonstrated the potential of the configurable systems in performing computation-intensive applications [92], [5], [8], [25].

Another interesting development in this field is the emergence of the Field-Programmable Multi-Chip Module (FPMCM) which combines the FPGA technology and the advanced packaging technology of the Multi-Chip Module (MCM). Our research group in UC Santa Cruz has been one of the first group to design and fabricate such FPMCM [27] using 12 Xilinx XC3042 FPGAs and a 1000-pin Field-Programmable Interconnect Chip (FPIC) from Aptix, where the total gate gate capacity is estimated to be between 24k to 44k gates. Other FPMCMs have also emerged [88], [2]. We are currently working on our second generation FPMCM which will have a gate capacity of 200k gates. Using the state-of-the-art FPMCM with a gate capacity of 200k gates, to even 400k gates using the latest FPGAs, a large-scale configurable system having a gate capacity of several millions of gates contained in a desk-top size box will soon become a reality.

There are a number of hardware and software challenges in designing such a large-scale configurable system. This thesis focuses on the software aspects of the challenges, which is the task of application development for this large-scale system. First, FPGAs are very resource-limited and slow compared to MPGAs and custom chips. A poor logic design often leads to a very low device utilization caused by the insufficient amount of routing resource or IO resource. Also, SRAM-based FPGAs suffer a great performance penalty on the signal routing implemented by pass-transisters. A large portion of the configurable hardware would be wasted under these circumstances which severely limits the potential of the configurable system. Second, a sophisticated design automation tool is essential in developing application for the large-scale configurable systems. Exising High-Level Synthesis tools [33], [65], [82], [90], [5] are not yet powerful enough to efficiently generate highly parallelized datapath architectures. Designs synthesized from these tools often require a large routing resource and IO resource which lead to poor device utilization. Many of the efficient designs on FPGAs are still done with low-level tools in order to overcome those

inefficiencies imposed by the existing design automation tools.

## 1.1 Key Contributions

In this thesis, these key issues essential in developing a design environment for the large scale configurable system have been focused on:

**Bit-serial circuit design for LUT-based FPGAs** In order to exploit the full potential of the configurable system, an effective circuit implementation method is needed which has high area-time efficiency and is easily mapped to the FPGA architecture. For this purpose, bit-serial circuits have been employed.

1. A set of bit-serial circuits have been designed on Xilinx XC3000 FPGA architecture. Special attention has been paid to the circuit implementation so that all the bit-serial circuits can be mapped onto the LUT-based logic blocks and can operate at the highest speed. They are all implemented in a single-level LUT logic where the critical path delay is only one LUT plus the flip-flop clock setup delay excluding the routing delay.
2. My bit-serial multiplier was compared against bit-parallel multipliers on Xilinx 4000 series FPGAs with fast carry-logic, which greatly favors bit-parallel operations. Multiplication is the fundamental operation in most of the high-performance applications. This study shows that bit-serial multipliers are not only smaller, but have a comparable throughput to the non-pipelined bit-parallel multipliers. The it-serial multiplier requires much less logic resources compared to bit-parallel multipliers. Therefore, bit-serial multipliers have a significantly lower area-time complexity, which makes them greatly favored the FPGA environment, where logic resources are scarce and the circuit speed is low.
3. The high routability and the low IO consumption of the bit-serial circuits have been confirmed. All the bit-serial designs developed in this study were fully routed by the automatic placer and router, even with the logic utilization as high as 100%. This characteristic is analytically confirmed by the Rent exponent approximation of the various bit-serial designs which ranges from 0.22 to 0.38. This is exceptionally low compared to the "real-life" circuits which was observed to be between 0.47 and 0.75 [64], [44]. The study by Donath [32] concludes that circuits whose Rent's exponent is lower than 0.5 are easily routable on a two-dimensional layout surface.
4. Most of the placement and routing results of these bit-serial circuits show a consistant performance of under 25ns signal propagation delays on Xilinx XC3100 FPGAs.

**C++ design capture system** A function-based design capture system was developed based on C++ operator overloading. This approach does not require a customized compiler which often limits the flexibility of the high-level programming language.

1. Any kind of expressions supported by C++ are allowed which greatly facilitates the description of the algorithm. By using overloaded operators, hardware declarations are done in the same way as the ordinary software programming. Therefore, the transition from the algorithm level to the hardware description level is straigtforward.
2. Design verification is done easily and thoroughly at the high level by mixing the software code inside the hardware description. The ease of transforming algorithm description into hardware description enables the designers to explore different algorithm implementations very easily which is essential in deriving a high-performance design.
3. The features of my C++ bit-serial synthesis system are demonstrated by various signal processing application designs. Several interesting trade-offs between the performance, hardware size, and the algorithm behavior have been identified which are highly algorithm-dependent.

4. This function-based hardware description system is very flexible for future expansions since the design capture scheme is object-oriented. New hardware description features can be added by simply linking the added object libraries to the existing synthesis program.
5. Circuit libraries for the datapath elements are effectively implemented using the object-oriented programming style of C++. Upgrading the circuit library and adding new circuits to the library is therefore straightforward.

**Bit-serial synthesis system** A fully automated bit-serial pipeline network synthesis tool has been developed. The synthesis process consists of pipeline synchronization, bit-serial control signal routing, pipeline register minimization, and IO scheduling. This is the first bit-serial synthesis system for FPGAs. Results show that this bit-serial synthesis system is capable of producing manual design quality circuits, which clearly outperforms some of the high-level synthesis tools for FPGAs. A new partitioning algorithm for multiple-FPGA systems was also developed which deals with the highly routable bit-serial circuits. This simple partitioning method is capable of producing circuit partitions which maximize the logic utilization.

## 1.2 Organization of the Thesis

This thesis is organized in the following way. In Chapter 2, the background of the research concerning the configurable systems is discussed. In Chapter 3, the bit-serial circuit designs are described in detail. In Chapter 4, the C++ bit-serial datapath synthesis system is described. In Chapter 5, several design examples using our bit-serial synthesis system are presented. In Chapter 6, a bit-serial datapath partitioning method is presented. The Rent's Rule analysis of the several bit-serial circuits are also presented in this chapter. After the conclusion in Chapter 7, the description of the prototype configurable system is presented as an appendix.

# Chapter 2

# Background

## 2.1 High-Performance Computation Applications

Together with the rapid development in VLSI and computer technologies, applications requiring huge computational power are being generated at an enormous pace. These applications cover a variety of fields including:

1. Digital signal processing :
   - Audio : Filtering, compression, synthesis, speech recognition.
   - Image : Filtering, compression, enhancement, restoration, segmentation, object recognition.
   - Video : compression, motion detection, film editing, video-on-demand server.
   - Sonar, Radar.
2. Computer graphics : CAD, virtual reality simulator, amusement games, special effects in motion pictures.
3. Simulation : Thermodynamics, electrodynamics, hydrodynamics, digital and analogue circuits.
4. Database : DNA sequence search, text search.

Despite the large computational power of the current state-of-the-art microprocessors for personal computers and workstations, many of these applications require much higher computational performance than those general-purpose processors can deliver. In order to provide the sufficient throughput for these *high-performance computation applications*, we can consider the following options:

1. *General-purpose processor architecture enhancement* : With the help of the process technology advancement, newer generations of microprocessors have continued to increase their performance significantly due to various architecture enhancements such as increasing bus bandwidth and memory bandwidth, pipelining, reduced instruction set design, multiple instruction execution, and dynamic instruction execution. Due to the enormous growing market for personal computers, development cost of microprocessors can afford to be quite large, and therefore the various technologies involved in the microprocessor have matured to a very high level. While this trend of performance increase through architecture enhancement will certainly continue, demands of some of the high-performance computation applications are already too high for these general-purpose microprocessors to be able to deliver the sufficient performance in the near future. And also, there will always be new applications emerging which demand more than what the state-of-the-art processors of the time can offer.

2. *Domain-specific architecture enhancement* : While general-purpose microprocessor enhancement has its limits due to the wide variety of applications they need to support, further enhancement is possible by narrowing down the applications to a certain domain. A perfect example is the digital signal processors (DSP) in which their architecture is optimized for applications in digital signal processing domain. A majority of digital signal processing applications include a large number of multiply-accumulate operations, and therefore introducing a multiply-accumulate unit in the datapath is a popular enhancement option. Also, fixed-point arithmetic is still popular in DSPs for circuit simplicity and speed, although floating-point DSPs have emerged to handle wider applications. Since DSPs also have a high-volume market because of their wide applications in digital signal processing, a large amount of resources have been invested and therefore the technologies behind them have matured as well.

3. *Application-specific architecture enhancement* : If we further limit the supporting applications to just one, opportunities for drastic improvements may often arise. This is due to the fact that unlike general-purpose microprocessors or DSPs which are designed to support wide variety of applications, these so called ASICs (application-specific ICs) are optimized for only a single application:

   (a) Optimized resource allocation :
   Since the computation sequence is known in advance, it is possible to allocate sufficient number of various types of functional units, register files, memory elements and bus connectivity so as to maximize the computation efficiency as well as the overall performance. This optimization process is well studied in *high-level synthesis*.

   (b) Optimized functional units :
   If each functional unit has a unique task or sufficiently small number of types of operations, it can be implemented with significantly fewer logic resources. A complete ALU functionality is unnecessary if its only function is an addition. A two-input multiplication unit can be simplified if one of the input is known constant.

   Since ASICs have limited use, many of them tend to have a small unstable market. Large investment in research and development is often hard in such cases, and many of them rely on cheaper VLSI implementation technologies such as gate arrays and FPGAs instead of a full custom design.

4. *Parallel processing* : Another popular approach in delivering high computation throughput is to increase the number of processing units. The difficulties in this approach are

   (a) Software : how to effectively capture computation parallelism (parallel programming language, parallel processing compiler).

   (b) Hardware : how to maximize parallel efficiency and minimize communication overhead (network architecture, communication routing policy).

   Despite these difficulties, it has a potential of achieving a speedup of more than a factor of hundreds. Therefore, for those high-end applications, parallel processing is sometimes the only solution. Development costs for general-purpose parallel processing systems are extremely high, and therefore their applications are limited to academic and military fields.

## 2.2 FPGA-Based Custom Computers

### 2.2.1 Field-Programmable Gate Arrays

An FPGA is a general-purpose, multi-level programmable logic device that is customized in the package by the end user [89]. It is composed of blocks of logic elements connected by programmable interconnects. The functionality

of the logic blocks may be fixed, or it can be programmable where each logic block is implemented by small SRAM segments. The programmable interconnect between logic blocks allows users to implement multi-level logic, removing many of the size limitations of the two-level logic structure seen in Programmable Logic Arrays (PLAs).

There are a number of FPGA devices with different architectures and technologies. There are several aspects of the FPGAs in which we can categorize these devices:

1. Logic element structure :
   (a) Programmable functionality implemented by SRAM lookup tables (LUTs).
   (b) Fixed functionality (multiplexer, AND, OR, EX-OR).
2. Programmable interconnect architecture :
   (a) Two-dimensional mesh routing with switch box matrices for changing routing directions.
   (b) One-dimensional segmented channel routing with feedthroughs for inter-channel routing.
   (c) Hierarchical routing with long lines for long distance connections and high fanout connections, and short lines for local connections.

   The first and the third interconnect architectures are often combined together.
3. Device configuration programming technology :
   (a) SRAM-based configuration where the interconnection routing is implemented by switching on the pass-transistors which is controlled by the SRAM cells. The added resistance and capacitance of the pass-transistors increases the signal propagation delay of the interconnection.
   (b) Anti-fuse configuration where the interconnection routing is implemented by breaking the anti-fuse which establishes electrical connection between wires. This process is not reversible, therefore this type of device is only one-time programmable. One the other hand, this technology provides a faster interconnection compared to SRAM-based configuration as well as higher density of components.

The advantages of the FPGA technology compared to other VLSI technologies such as Mask-Programmable Gate Arrays (MPGA) and full-custom fabrication are the low design cost, low non-recurring engineering (NRE) cost and short turnaround time. In terms of design cost, full-custom fabrication requires a huge investment in the development since the designer is responsible for not only designing the circuit but also laying out the circuit onto silicon. Standard-cell approach aids the designer the circuit layout process by providing the predesigned layout of each circuit segment. MPGAs and FPGAs only need the structural information of the circuit and not the physical implementation, and therefore the design cost is significantly reduced. Automatic tools are provided in order to transform structural circuit information into the physical layout for MPGAs and FPGAs. In terms of NRE cost, full custom is most expensive since it needs to generate custom masks for each manufacturing step, where MPGAs only require the wiring step to be customized. FPGAs do not need any customized steps in manufacturing the chip and therefore the NRE cost is reduced to essentially zero. Since the NRE cost is zero, FPGAs have the fastest turnaround among the three. Another big advantage of FPGAs is that they are standard products. The FPGA manufacturers incur the cost of improving the density and performance of the devices with new architecture and process technology. Also, FPGAs are speed-graded where slower parts which were produced by the variation in the fabrication process sell for lower price, giving the users the price/performance trade-off. This trade-off is not possible for MPGAs and full-custom designs since they are usually not standard products and must throw away the slow chips which do not meet the spec. Therefore, they are typically designed considering the worst-case process characteristics, which may be approximately twenty percent slower than the nominal characteristics.

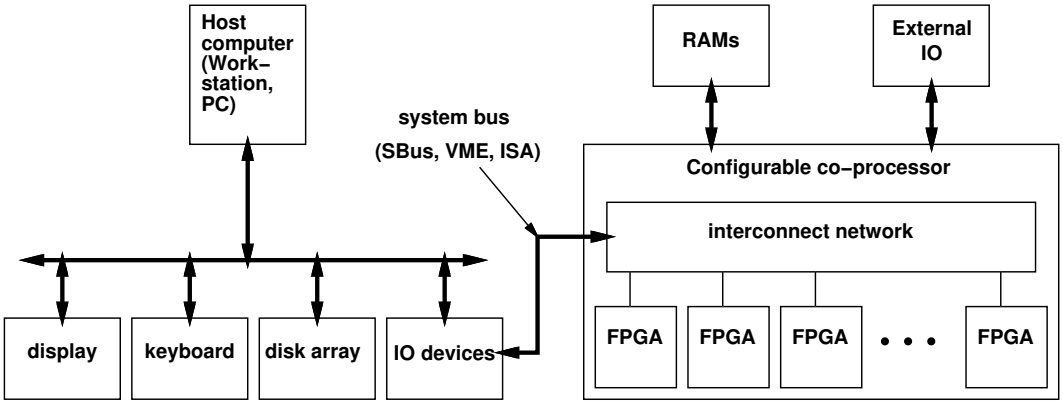



Figure 2.1: Configurable coprocessor system architecture.

On the other hand, chip size, chip cost and circuit speed are the major disadvantages of the FPGAs. The area penalty for introducing field-programmability is significant, where the current FPGAs are about ten times larger for the same gate capacity as the equivalent MPGAs, and therefore more expensive on a per-chip basis. Full-custom chips have an obvious edge on per-chip cost. Therefore, as the production volume grows to a certain point, full-custom chips are most cost-effective, and FPGAs becomes most expensive. Also because of the low logic density, FPGAs are limited in the logic capacity up to 50,000 gates, where the largest MPGAs are in the range of hundreds of thousands of gates, and custom chips are in the range of millions of gates. Another disadvantage of FPGAs is the circuit speed. Especially with SRAM-based programmable interconnections, routing delays are created not only from the parasitic wiring capacitance but also the resistance and capacitance of the routing switches of the pass-transistors and multiplexers. Current FPGAs are two to three times slower than MPGAs [89], which are in turn two to three times slower than full custom.

### 2.2.2 Custom Computing

The concept of *custom computing* emerged from the successful developments of SRAM-based *reprogrammable* logic devices. Its basic idea is to build a custom hardware which executes a certain application using a system composed of SRAM-based FPGAs. By implementing computations directly on the hardware, the achieved speedup of this FPGA-based system can be in the order of hundreds or more. Since SRAM-based FPGAs are reprogrammable, they can be reconfigured to implement different applications. These systems (let us call them the *configurable systems*) inherit the architectural advantages of the ASIC implementation and the flexibilty of the general-purpose microprocessors and DSPs. All the existing configurable systems work as coprocessors for a host computer. The general architecture of the configurable coprocessor system is illustrated in Fig.2.1.

1. Host computer : This is the main platform in which the designer develops the application, configures the configurable coprocessor, and executes the application software.
   (a) Application development : The designer develops the application hardware for the configurable coprocessor and the application software for the host computer on this platform.
   (b) Coprocessor configuration : The host computer supplies configuration bitstream to each programmable devices on the configurable coprocessor through the system bus.

(c) Application software execution: The host computer supplies the data to the configurable coprocessor, reads back the results and displays them on the screen in some textual or graphical format.

2. System bus : The data transaction between the host and the coprocessor is done through this system bus.

3. Configurable coprocessor : The application hardware is implemented on this platform. The programmable devices are SRAM-based, therefore can be reprogrammed for different applications.

4. RAMs : This is used for storing the intermediate results and buffering the input/output data.

5. External IO: If the system bus cannot provide enough bandwidth to sustain the required throughput, dedicated external IOs may be added.

The following describes some of the important configurable systems to date.

1. *The Quickturn Logic Emulator* : Quickturn addressed the problem of prototyping large-scale ASICs and systems in the RPM Logic Emulator by using an array of Xilinx FPGAs and custom routing chips [92]. The main purpose of this system is to verify the functionality of the ASIC chips with up to several millions of gates. Once downloaded on the system, the emulated design executes up to a million times faster than a software simulation.

2. *Ganglion* : This system is designed to implement digital neural networks using 40 Xilinx FPGA chips (36 XC3090s and 4 XC3042s) [25]. A digital neural network consists of a number of scalers (multiplier with a constant value) and accumulators. Ganglion implements a three-layer feed-forward network where there are 12, 14, and 4 nodes at each layer, respectively. The main effort has been spent to minimize the scaler circuit size by splitting the multiplication into two parts and implementing each partial multiplication directly with LUTs.

3. *Programmable Active Memory* : DEC Paris has produced several generations of configurable systems (Perle-0, Perle-1) based on Xilinx FPGAs for experimenting with logic architectures for highly parallel computations [8]. They have been programmed to implement a variety of algorithms, including image filtering, very long word arithmetic and RSA encryption. RSA encryption on 512-bit keys can be executed at a rate of 1500 bits per second and decoded at 200kbit/s, which is ten times faster than the fastest reported custom IC solutions. A neural network emulator built on Perle-1 operates at 500 million synapses per second.

4. *SRC Splash and Splash 2* : Splash, developed at IDA Supercomputing Research Center consists of a linear array of function units, each consisting of a Xilinx XC3090 FPGA with a 128K byte external memory. It is targeted to one-dimensional systolic problems and has been used for pattern matching DNA sequences. Splash 2 is the successor of Splash which consists of Xilinx XC4010 FPGAs and a crossbar switch to establish inter-processor connections [5], [48].

5. *HP Teramac* : Teramac developed at Hewlett-Packard is composed of custom FPGA chips which are packaged in an MCM [2]. Each MCM holds 27 such FPGA chips on a 39-layer ceramic substrate. Four MCMs are mounted on one board, and up to 16 boards can be connected to the system. They have spent their effort on optimizing the architecture of the system, from the board-level to the chip-level, in which the compilation time of the design can be significantly shortened. Their FPGA routing architecture is designed so that the total compilation time per chip is less than ten seconds, which is several orders of a magnitude smaller than the Xilinx compiler. The compilation time of the whole system takes around 2 hours.

Other notable configurable systems are ACME (UCSC), Anyboard (North Carolina SU) [90], Borg-I,II (UCSC), Champ (Lockheed Sanders) [13], PRISM-I,II (Brown U) [1], Rasa (Carnegie Mellon U) [82].

## 2.3 Field-Programmable Multi-Chip Module

A field-programmable multi-chip module combines the digital circuit implementation technology and the advanced VLSI packaging technology to deliver a high-density field-programmable device. Notable work has been done at UCSC (FPMCM-I) [28], [29], Altera (EPF8050M) [88], Hewlett-Packard (Teramac) [2].

### 2.3.1 Multi-Chip Module Technology

A Multi-chip module (MCM) is a device in which several ICs are attached to a single substrate and then packaged as a single unit. MCM technologies are categorized by the types of substrate fabrication processes and substrate materials. Low-cost MCM-L technology uses high density laminated PC boards, MCM-C uses ceramic which can deliver large number of IO pins, and MCM-D uses silicon (thus called a silicon-on-silicon MCM) which has the highest wiring density of all and various efforts have been made in lowering the cost of this technology.

AT&T Bell Laboratories have being developing the MCM-D technology which targets high volume consumer products in the telephone business [40]. In their MCM-D process, the substrate contains two metal layers that are used as interconnect or as passive components such as resistors and capacitors [41]. Conventional ICs, sometimes postprocessed to meet solder assembly requirements, are diced, bumped, and tested. They are then attached, face down, on the the substrate with solder bumps in a manner similar to surface mount. The resulting assembly is attached, wirebonded, and encapsulated in a second level package. The second level package, used for insertion into the final system, is typically a pin or ball-grid array (BGA). There are two layers with a wire pitch on the order of 1.5 mils. The process has also been carefully optimized to minimize the cost without sacrificing the performance of digital circuits [40].

### 2.3.2 Impact of MCM Technology on FPGA-based System

The advantages of MCM over conventional single-chip packaging are well known: smaller feature size, lower electrical parasitics due to chip interconnections, increased interconnect resources, and reduced packaging cost. Also, the new issues of *design for MCM*, optimizing the IC design for strictly MCM, dramatically enhance these advantages. The impact of MCM technology on FPGA-based systems in particular could be described as follows:

1. Assuming BGA packaging for both single-chip packaging and FPMCM packaging, the area of FPMCM can be estimated to be around 20%, counting the number of pins in the packages, compared to that of a single-chip packaging FPGA-based hardware with the same amount of logic resources (Fig.2.2). Especially when the FPGA-based system is used as a co-processor which is desired to fit inside the chassis of a workstation or even inside a lap-top personal computer, this size compaction effect is quite appealing. Also as a logic emulator, the fact that a single FPMCM can emulate middle range ASICs with tens of thousands of gates, which in fact have been the fastest growing market for logic emulation, has already drawn much attention as seen in the response to Altera's introduction of their silicon-on-ceramic high-density programmable multi-chip module EPF8050M [88].

2. Power savings due to the low capacitive loads of shorter MCM interconnections is an important feature for FPGA-based co-processor for add-on cards and portable computer terminals. If the IO pad drivers on the FPGA die are to be redesigned and optimized for MCM assembly purpose, the power savings increases dramatically [40].

3. One of the slowest parts in FPGA is the IO pad drivers. The large penalty of chip-to-chip communications severely limits the functional speed of the FPGA-based system. With optimized IO drivers for MCM, this performance limitation can be significantly relaxed.

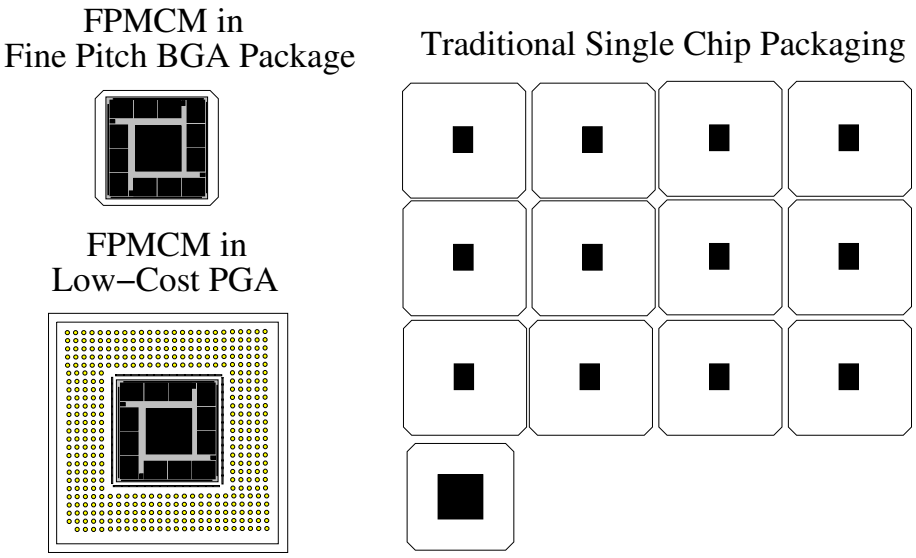



Figure 2.2: Hardware size comparison between conventional single-chip packaging and MCM packaging.

4. Area IO can be one of the most interesting issues of *design for MCM*, especially on FPGA chips[69]. First of all, area IO can be a solution for the problem of pin limitation on conventional peripheral IO. This pin limitation problem is most evident on logic emulation systems, where the pin availability tends to determine the logic capacity of the system rather than the logic availability itself. And therefore the effort of minimizing the chip-to-chip communications is one of the main research topics in this application. Second, since the area IO provides pins across the whole chip area, routing resources spent on routing the IO pins to the periphery can be saved which will increase the routability of each individual FPGA. Finally, area IO can have a great impact on logic partition and placement problems. Conventionally, these two problems had to be solved separately and sequentially. Since area IO make the chip boundary more vague, a simultaneous partition and placement algorithm may well be developed in the future.

### 2.3.3 UCSC FPMCM-I

The project team at UCSC have designed and fabricated a first generation Field Programmable Multi-Chip Module in 1994. (FPMCM-I) [28], [29]. The purpose of the first generation device is to fully exercise the MCM fabrication and assembly technology, clarify the technical problems of FPMCM-I design, and try out an initial architecture. The main features of our FPMCM-I are:

1. First device to assemble FPGA dice using flip-chip silicon-on-silicon MCM-D assembly technology. The silicon substrate consists of ground plane, power plane, and two signal routing layers. The size of the substrate is 30.6mm in each dimension, 9 $cm^2$ in area. The design rule imposed on the substrate is 15$\mu$ metal wire and 20$\mu$ spacing. The wiring density of the substrate was unusually high for the AT&T process, the fabricater of

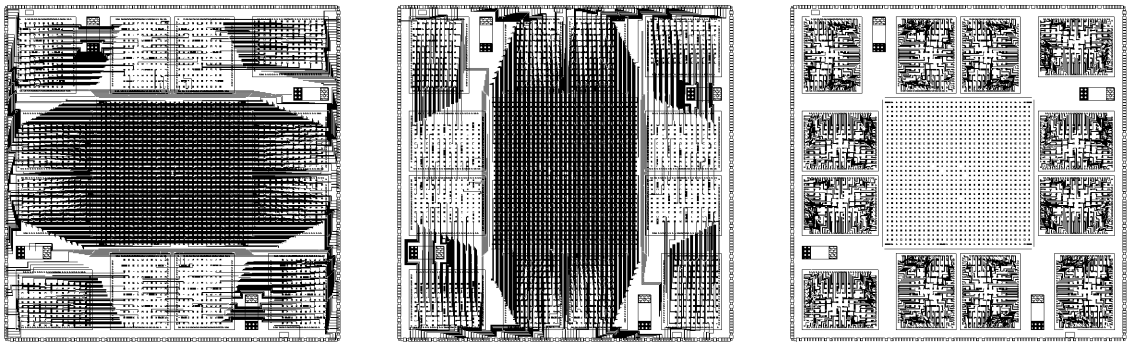


Figure 2.3: Physical Design of FPMCM-I. Outer dimensions of the module are 30.6 mm square. The device contains 12 X3042's and a single Aptix FPIC. Four ceramic chip capacitors are provided to stabilize the on module power and ground planes. The figure shows the three metal routing layers. The first two (X and Y) layers shown are on the silicon substrate, while the third consists of an identical rerouting pattern for each X3042 and the solder bump grid of the FPIC which is necessary because the bond pad pitch on the 3000 series is too small for inexpensive, reliable flip chip assembly.

the substrate and also the assembler of our FPMCM, and therefore it became a good benchmark for testing the reliability of their well-tuned MCM process (Fig.2.3).

2. Consists of 12 XC3042 dice fabricated at AT&T plus a Field-Programmable Interconnect Chip (FPIC) from Aptix (Fig.2.4).
   - IO pins on each XC3042 FPGA die are redistributed by adding one metal routing layer on the conventional FPGA wafer. This additional process was necessary to provide enough pad spacing (16.5 mil pitch) for the AT&T low cost flip-chip assembly using solder paste stencil print.
   - FPIC consists of an area IO grid of $32 \times 32$. There are 976 user pins which implement a partial crossbar.
3. 24k$\sim$44k usable gate counts (estimated based on Xilinx databook).
4. Four ceramic chip capacitors mounted on the substrate to reduce ground bounce.
5. 476 bondpads on FPMCM-I substrate, in which 171 are power and ground pins and 305 are signal pins. All 305 signal pins are connected to the FPIC. However, only 204 signal pins are routed to the FPMCM-I package because of the our limited access to the high pin count MCM packaging.
6. Estimated maximum clock frequency: 15 MHz (worst case), 25 MHz (best case). The worst case occurs when one or more FPGA dice are the slowest "-50" speed grade. The best case occurs when all the 12 FPGA dice are the fastest "-125" speed grade. In the assembly process we are using, we do not have the freedom of selecting speed parts prior to assembly. This presents a serious problem in guaranteeing a certain performance of the whole MCM. Speed testing of the bare die need to become available in order to overcome this problem in the future.

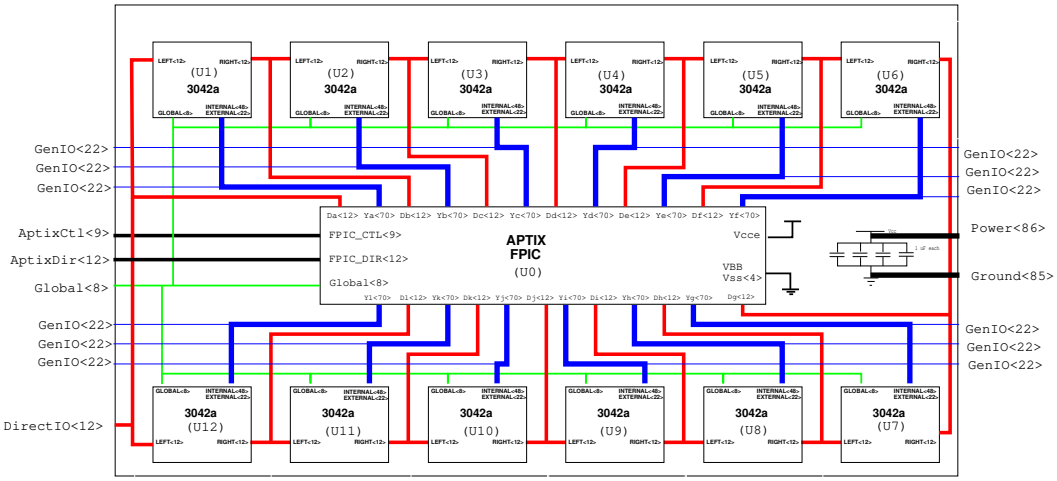



Figure 2.4: Architecture of the first generation FPMCM.

### 2.3.4 Altera EPF8050M FPMCM

Altera has designed their own field-programmable MCM as a product which can deliver up to 50K gates on a single module. Its features are:

1. Uses 32-layer ceramic substrate with a size of 2.16" square, 560-pin PGA.
2. Consists of four EPF81188 PLDs, each having up to 12K usable gates, already packaged in a ceramic BGA package mounted on the top side of the ceramic substrate, and an FPIC from Aptix which is mounted on the bottom size in the bare die form by the flip-chip assembly.
3. Total user gate capacity of up to 50K gates.

### 2.3.5 FPMCMs for Large-Scale Configurable Systems

By utilizing the high-density packaging capability of the MCM technology, we are currently designing a second generation FPMCM using the latest 50k-gate Altera FPLDs. The total logic capacity of the FPMCM is 200k gates (4 FPLDs) which will be mounted on a 4.7 $cm^2$ silicon substrate. With such a large logic capacity within such a small area, building a desktop-size configurable system with the logic capacity of several million gates becomes a reality within a few years. With such a large-scale configurable system, an effective programming environment for supporting large scale circuit design becomes crucial.

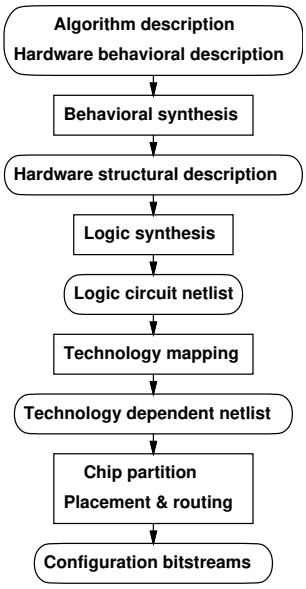



Figure 2.5: FPGA design flow

## 2.4 Challenges of Application Development for Large-Scale FPMCM-Based Configurable Coprocessor System

In this thesis, the author investigates a thorough methodology of designing computation-intensive applications on a large-scale FPMCM-based configurable coprocessor system. The target applications are characterized as :

1. Composed mainly of repetitive computations where extensive parallelism can be exploited by bit-level and function-level pipelining.

Many of the signal processing applications and real-time systems fall into this category. Other computation-intensive applications often have these highly repetitive routines as well. Applications which exhibit only a moderate parallelism may also benefit by using the configurable systems, however its speedup impact may not be large enough compared with the state-of-the-art microprocessor to justify the effort. This thesis focuses on the applications which present opportunities to achieve massive speedups.

The major problem in designing such applications for the configurable systems is *how to utilize the programmable logic resources efficiently and convert them into effective computational power*. This is a challenging problem since it involves many aspects of digital system design (Fig.2.5).

1. Algorithm description : For a given problem, a correct and an efficient algorithm has to be designed and verified. High-level programming languages such as C, C++ and Pascal or a hardware description languages such as Verilog and VHDL can be used for this purpose. For signal processing applications, specialized programming languages such as SILAGE [91] have been developed to capture the signal processing algorithm in terms of *difference equations* which are used to describe time-discrete systems.

2. Hardware structure description (RTL description): The algorithm description is converted into a hardware description to derive a basic hardware structure which implements the algorithm. This involves allocating

the operations into hardware functional units, scheduling the operations on each functional unit, and deriving the necessary interconnections.

3. Logic circuit netlist : From the RTL description, datapath elements and controllers are synthesized into logic circuits. Datapath elements are synthesized from the predefined circuit library, and controllers are synthesized from a logic synthesis tool.

4. Technology mapping : The logic circuits are then mapped to a specific FPGA architecture which is composed of a specific logic element such as LUT or multiplexers.

5. Chip partitioning : If the logic circuit is too large to fit into a single FPGA chip, it needs to be partitioned into multiple clusters. Many partitioning algorithms have been proposed [60], [38], [22], [23], [21], but the only currently available commercial partitioner is included in Altera's MAX-Plus II CAD tool.

6. Placement and routing : Finally, logic blocks are placed and interconnections are routed automatically with a vendor-specific tool.

### 2.4.1 Limitation of High-Level Synthesis Approach on FPGA Design

Methods which automatically transform the algorithm-level description (or a hardware behavioral description) into a hardware structural RTL description (register-transfer-level description) have been developed mainly for signal processing applications [91], [15], [50]. For FPGA datapath synthesis, several tools are developed which perform this behavioral synthesis [82], [90], [1], [59], [33],[65]. These behavioral synthesis systems are based on the recent work on *high-level synthesis*. The main focus point of this research is to derive a horizontally coded VLIW datapath architecture with a given performance specification so as to minimize the overall circuit area. The high-level synthesis paradigm has attracted a large number of researchers because of its formal problem formulation using integer-linear programming. This approach, which was originally targeted for facilitating ASIC designs is tailored for producing efficient circuits under strict resource constraints. However, there is a severe limitation in this approach for FPGA synthesis. Let us look into one example of such study by Leeser et al. [65]. They have applied their behavioral synthesis system *Bedroc* to synthesize a fifth-order elliptic wave digital filter on Altera MAX FPLDs. First, since the multiprocessor architecture needs to be exclusively bit-parallel, they had to reduce the word size to 6 bits in order to reduce the overall size of the hardware. Second, they needed to reduce the word size of each filter coefficient to 3 bits in order to make the multiplier small. Third, they compared with a direct implementation where all the multiplications and additions were mapped to distinct hardwares, and the result was that the synthesized filter needed twice the logic resource compared with the direct implementation with a speedup of only 50%. The direct implementation was about 7800 gates (391 Altera MAX macrocells) and the sampling period was 50 kHz, whereas the synthesized version was about 15000 gates (776 Altera MAX macrocells) and the sampling period was 77 kHz. The very same design can be implemented using our bit-serial synthesis system which produces a circuit size of 5312 gates (381 Xilinx CLBs) whose sampling rate is 1.92 MHz. Our implementation will be described more in detail in Chapter 5. The important lessons to be learned here are

1. The main feature of the VLIW architecture is the high degree of hardware multiplexing, or in other words, many different operations are executed on each functional unit. If the degree of hardware multiplexing is low, or in a radical case *one* as in the case of direct implementation, each execution unit can be drastically optimized. A complete ALU can be simplified into an adder or a subtracter. Bit shifting can be done on a wire instead of using an ALU. A two-input multiplier can be simplified into a scaler if one of the multiplicand is a known constant. With this multiprocessor architecture, not only the circuit size becomes larger because of the high degree of hardware multiplexing which necessitates the functional units to be flexible, but also extra register elements may be needed to store the filter coefficients which would have been hardwired with

a low hardware multiplexing implementation. The ability of hardwired implementation is one of the biggest advantages of configurable systems.

2. In order for this VLIW architecture to be effective, each execution unit such as multipliers and ALUs have to function at a high speed. In ASIC implementation, each functional unit can be optimized by carefully designing the circuit structure and layout. The inevitable disadvantage of slow speed of FPGAs (especially SRAM-based FPGAs) seriously degrades the effectiveness of the VLIW architecture. Therefore, most of these high-level synthesis studies for FPGAs is targeted for mere prototyping and still resort to ASIC fabrication for the ultimate implementation.

3. The VLIW datapath architecture introduces considerable control circuitry overhead to control the hardware multiplexing and bus communications. This can be microcoded in a ROM or be hardwired as a sequencer. Routing these control signals throughout the circuit consumes significant amount of routing resources. These control circuits and control signal routing would have been minimal if the hardware multiplexing was low.

4. The overall circuit of the VLIW datapath architecture is highly connective because of the multiple bit-parallel bus connections and the centralized control mechanism where each control signal may have to be distributed all over the circuit. These highly connective circuits are very hard to partition and route.

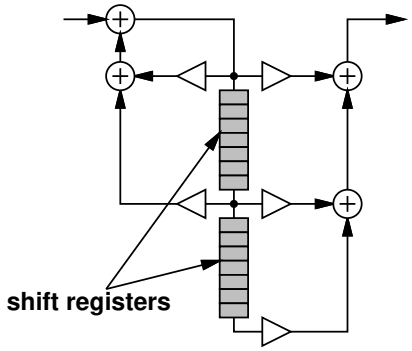
Another reason why the author believes that the high-level synthesis approach may not be effective for high-performance configurable systems comes from observing the recent developments by one of the leading research groups in this field. Catthoor et al. from IMEC who are the main developers of CATHEDRAL-I (bit-serial synthesis system) and CATHEDRAL-II (high-level synthesis system) [91] have been working on the subsequent synthesis systems, namely CATHEDRAL-III and CATHEDRAL-IV [18], [75][30]. CATHEDRAL-III is a synthesis system which produces a cooperating datapath architecture with hierarchical controller which can exhibit higher throughput than the VLIW datapath architecture derived by the high-level synthesis approach of CATHEDRAL-II. The difference between a multiprocessor datapath and a *cooperating* datapath is the degree of hardware multiplexing. A cooperating datapath is composed of relatively complex, pipelined execution modules. It takes a signal flow graph as the design input, and the designer specifies the actual pipelined datapath architecture, i.e. the assignment of each operation to each functional units. There is no automatic behavioral synthesis tool which derives this pipelined datapath architecture. The formal approaches of the high-level synthesis does not derive this type of architecture. CATHEDRAL-IV is the newest synthesis system which produces a regular array parallel processing architecture in which the throughput is the highest among their four synthesis tools. Again, the automatic behavioral synthesis tool is not provided for this as well. These new developments suggest that the high-level synthesis approach has a limitation of delivering high computational throughput, which forced them to search for new synthesis approaches. The fact that their two newer synthesis systems do not (or unable to) support behavioral synthesis in exchange for higher performance is, I believe, an indication of a shift in the direction of the hardware synthesis paradigm:

- Behavioral synthesis methodologies developed in the "high-level synthesis" studies are still not the ultimate solution in deriving a high performance hardware, and therefore there are still many opportunities of developing new and more effective hardware synthesis methodologies.

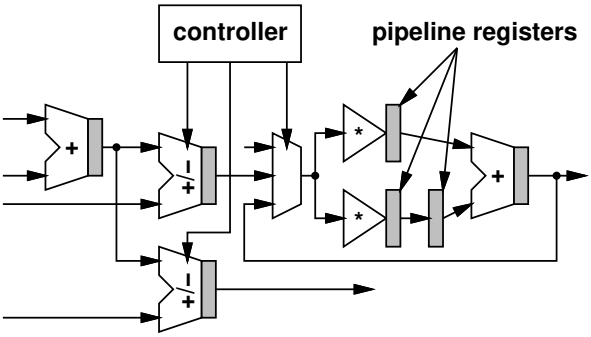
These four types of different architecture styles of the CATHEDRAL compilers are illustrated in Fig.2.6.

### 2.4.2 Resource Limitations and Low Performance of SRAM-based FPGAs

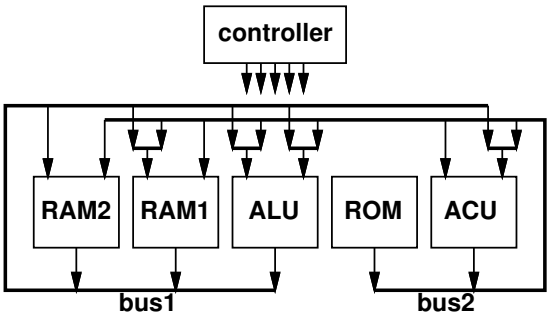
The challenge of FPGA synthesis is how to maximize the resource utilization since FPGAs are low in logic density and slow compared to MPGAs and custom chips.



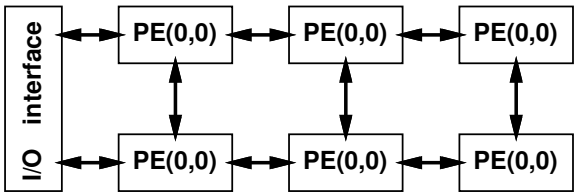

(a) Bit–serial pipeline architecture
(CATHEDRAL–I)




(c) Cooperating pipelined datapath architecture
(CATHEDRAL–III)




(b) Multiprocessor datapath architecture
(CATHEDRAL–II)




(d) Regular array parallel processor architecture
(CATHEDRAL–IV)

Figure 2.6: Four types of architecture styles derived by generations of CATHEDRAL silicon compilers. The two newer generations are targeted for applications with a higher throughput requirement.

1. SRAM-based FPGAs have limited routing resources. A study by El Gamal [34] shows that a circuit having a Rent exponent [64] of 0.66 requires $O(\log N)$ routing tracks per channel on a $N \times N$ two-dimensional lattice layout. This implies that as the chip size grow, the routing resource also have to grow which reduces the logic density. On the other hand, for an FPGA family such as Xilinx 3000 series which has the same number of routing tracks per channel for all device sizes, the logic utilization normally becomes lower for larger devices because of the insufficient routing resource. In either cases, as the device size grow with the help of the advanced silicon technology, larger portion of the silicon area are utilized only by wiring.

2. The IO pin limitation is another factor which decreases the device utilization. While IO count can normally grow to the square of the chip area, this growth in IO count is not sufficient for circuits which have Rent exponent higher than 0.5. The IO pin limitation becomes more severe as the device size grows. To make the matters worse, large multiple-FPGA systems require additional IO pins to route the inter-chip connections through intermediate FPGA chips if there are no direct wire connections between the two chips which needs to be connected. Therefore, the IO pin limitation becomes a major problem for the large-scale configurable systems which this thesis targets on.

3. The SRAM-based routing scheme using pass-transistors produces large routing delays. Circuit performance therefore depends heavily on the quality of the placement and routing which is extremely hard to predict.

For these reasons, efficient and high-performance designs are mostly derived by careful structural hardware design as well as logic design and floorplanning. Bertin and Touati discuss their experience of the application development on their FPGA-based configurable system (PAMs) [9] where they consider the high-level synthesis tools as *impractical* because of the inefficiency which the tools introduce both in the device utilization and speed. A meet-in-the-middle approach facilitates the design process by providing circuit libraries optimized for FPGA implementation. This design style seems to be most popular among FPGA designers who are familiar with most of the hardware design aspects. However, for a majority of the application developers who may only be familiar with C or C++, this approach is still a meet-in-the-*bottom* approach.

## 2.5 Bit-Serial Datapath Synthesis on FPGA

Bit-serial architecture has been investigated by many researchers in the past mainly for VLSI implementations. Several silicon compilers using bit-serial architecture have been developed for implementing digital filters and other real-time applications [58], [73]. Jain et al. have developed a silicon compiler called CATHEDRAL-I in which high-level design support was one of the main emphasis of their system[58]. CATHEDRAL-I is composed of several synthesis tools:

1. A filter synthesis tool automatically derives the optimal filter structure, filter coefficients and signal word size from the given frequency response specification.

2. A bit-serial architecture synthesis tool automatically derives the bit-serial pipeline network while optimizing the placement of the pipeline synchronization registers.

3. A layout synthesis tool places each bit-serial circuit using a standard cell placement algorithm and routes the interconnections.

Murray and Denyer also have developed their silicon compiler called FIRST [73] which synthesizes signal processing ASIC chips and systems exclusively using the bit-serial architecture. They claim that bit-serial architecture has a major advantage over bit-parallel alternatives, especially in terms of signal routing and inter-chip communications, and also in computation efficiency through bit-level pipelining. They do not support high-level features as seen

in CATHEDRAL-I, and the designer needs to describe the structure of the datapath, i.e. the interconnections of the bit-serial modules. The pipeline synchronization is done by hand (synchronization registers are handled as bit-serial modules) as opposed to automatic synchronization in CATHEDRAL-I. On the other hand, FIRST provides a variety of bit-serial circuit options, which in return can implement a wider variety of applications than CATHEDRAL-I. One major difference is that FIRST provides a 2's complement multiplier with full double precision output whereas CATHEDRAL-I only supports scalers (a combination of shifters and adders to implement a multiplication with a constant value) with only a single precision output where the lower half of the product is truncated. FIRST does not support scalers and the constants are loaded from a ROM.

### 2.5.1 Bit-Serial Vs. Bit-Parallel

The cooperating datapath architecture derived by CATHEDRAL-III is actually an expansion of the bit-serial datapath architecture approach of CATHEDRAL and FIRST where each operation in the data flow graph is assigned a distinct functional unit. Instead of using bit-serial, they use bit-parallel circuits. And instead of having no hardware multiplexing as in bit-serial datapaths where each operation is assigned to a distinct functional unit, they allow a small degree of hardware multiplexing, although this is done strictly by hand. This low degree of hardware sharing is allowed in order to reduce the hardware size without decreasing the throughput. This is possible when there are cyclic data dependencies in the computation in which case the new iteration of the algorithm has to wait for the results of the previous input-dependent computations. In such a case, each distinct functional unit may become idle, in which a different operation (but the same operation type) may be executed. For the bit-serial datapath synthesis system developed here, the pipelined datapath is directly from the data flow of the computation sequence described in C++. This system does not automatically derive a hardware multiplexed architecture since a behavioral synthesis formulation for this type of low degree hardware multiplexing is not available today [18]. Our C++ programming environment support control-flow features in which the designer can actually describe a hardware multiplexed datapath structure manually. Also, as I will demonstrate in Chapter 5, it is often possible to modify the algorithm so that the idle cycles created by the cyclic data dependency can be omitted.

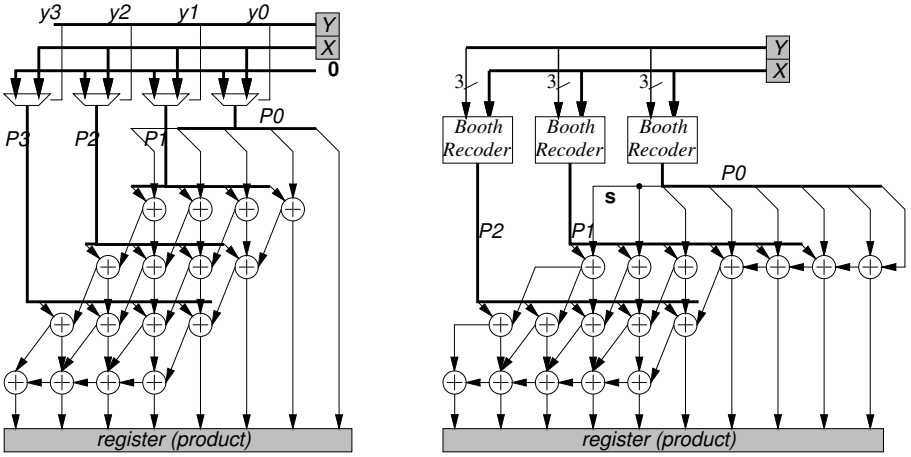
Therefore, the question is: which is better, bit-serial or bit-parallel? Denyer [73] claims that by adding to the fact that bit-serial circuits are much smaller than bit-parallel circuits, the bit-serial approach can be quite effective in implementing high throughput datapaths because of their short critical paths and local connectivity.

Let us give one example of the comparison between bit-parallel versus bit-serial for the multiplier designs on Xilinx 4000 FPGA. This FPGA family has dedicated fast carry logic (0.75 ns per bit) for implementing fast ripple-carry adders which greatly favors bit-parallel circuits. I show here that even on this FPGA architecture which is tuned to implement efficient bit-parallel circuits, bit-serial circuits can actually outperform bit-parallel circuits. For bit-parallel multiplication, three schemes are considered: conventional array multiplication, array multiplication using Booth's recoding technique, and a Booth-encoded Wallace Tree architecture designed by Kumar et al. [63]. Fig.2.7 shows these two bit-parallel multipliers implemented with ripple-carry adder arrays. In Fig.2.7(b), Booth's recoder can be implemented with $N+1$ CLBs for each $N+1$-bit partial product outputs with single-level logic.

The delay model shown in Fig.2.8 is used to estimate the speed of the multiplier circuits:

| | | |
|---|---|---|
| $t_{carry}$ | : | Carry input to carry output |
| $t_{cin}$ | : | Input operands to carry output |
| $t_{sum}$ | : | Carry input to sum output |
| $t'_{sum}$ | : | Input operands to sum output |
| $t_{pd}$ | : | Multiplicand ($X$) input to partial product ($X \cdot y_i$) output |
| $t_{ff}$ | : | Flip-flop delay (clock to output) |

It is assumed that the both operands $X, Y$ and product $S$ are latched. The critical path length of the parallel multipliers, especially the adder array in particular, is analyzed in the following way:




(a) Array multiplier (b) Booth's multipler.

Figure 2.7: Bit-parallel multipliers.

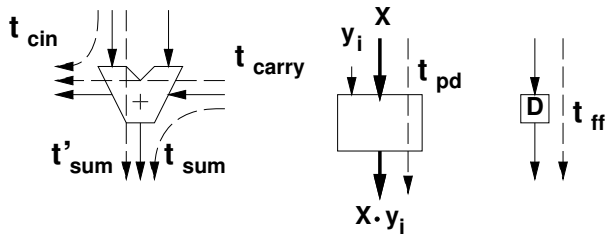



Figure 2.8: Circuit delay model.

| | | |
|---|---|---|
| $T^{(add)}$ | : | Single adder stage delay. |
| $T^{(stack)}$ | : | Delay from the partial products $X \cdot y_i$ to the input of the rightmost adder cell of the final adder stage. |
| $T^{(overhead)}$ | : | Partial product calculation delay and flip-flop delay. |

The total delay of a parallel multiplier is given by the sum of the above three numbers : $T = T^{(add)} + T^{(stack)} + T^{(overhead)}$. For conventional array multiplier, these three delays are given as

$$\begin{array}{lcll} T^{(add)}_{array} & = & t_{cin} + (N-2) \cdot t_{carry} + t_{sum} & , \\ T^{(stack)}_{array} & = & \lfloor \frac{N}{2} \rfloor \cdot max(2t'_{sum}, t_{cin} + t_{sum}) + t'_{sum}, & \\ T^{(overhead)}_{array} & = & t_{pd} + t_{ff} & . \end{array} \tag{2.1}$$

And for Booth's multiplier, the three delay figures are

$$\begin{array}{lcll} T^{(add)}_{booth} & = & t_{cin} + (N-1) \cdot t_{carry} + t_{sum} & , \\ T^{(stack)}_{booth} & = & (\lceil \frac{N}{4} \rceil) \cdot max(2t'_{sum}, t_{cin} + t_{sum}) + t'_{sum}, & \\ T^{(overhead)}_{booth} & = & t_{pd} + t_{ff} & . \end{array} \tag{2.2}$$

where $T_{array}$ is the critical path delay for the parallel array multiplier, and $T_{booth}$ is that of the Booth's multiplier. Using their -5 speed grade parts for the XC4000 FPGA family, the delay factor becomes

$$\begin{array}{lcl} t_{carry} & = & 0.75 \text{ ns}, \\ t_{cin} & = & 5.5 \text{ ns}, \\ t_{sum} & = & 6.0 \text{ ns}, \\ t'_{sum} & = & 4.5 \text{ ns (same as LUT delay } t_{lut}), \\ t_{pd} & = & 4.5 \text{ ns (same as LUT delay } t_{lut}), \\ t_{ff} & = & 3.0 \text{ ns}. \end{array}$$

Using these values, eq.(2.1) and (2.2) become

$$\begin{array}{lcl} T_{array} & \sim & 6.5N + 20.0 \text{ (ns)} \\ T_{booth} & \sim & 3.625N + 22.25 \text{ (ns)} \end{array} \tag{2.3}$$

The circuit size of the two multipliers on Xilinx 4000 series FPGA are given as

$$\begin{array}{lcll} A_{array} & = & \lceil \frac{N}{2} \rceil \cdot N & \text{(adder array)} \\ & + & N \cdot \lceil \frac{N}{2} \rceil & \text{(partial products),} \\ A_{booth} & = & \lceil \frac{N}{2} \rceil \cdot \lceil \frac{N+1}{2} \rceil + 1 & \text{(adder array)} \\ & + & N \cdot \lceil \frac{N+3}{2} \rceil & \text{(Booth's recoder)} \end{array} \tag{2.4}$$

where the unit is number of CLBs (Configurable Logic Blocks). Kumar claimes that their Wallace Tree multiplier design has a critical path delay of

$$T_{wallace} = 10 + \lceil \log_2 N \rceil (8.5 + 1.5N) \text{ (ns)} \tag{2.5}$$

and a circuit size of

$$A_{wallace} = 1.5N^2 - 5 \text{ (CLBs)}. \tag{2.6}$$

| Word size | 8 bits | 12 bits | 16 bits | 24 bits | 32 bits | 48 bits | 64 bits |
|---|---|---|---|---|---|---|---|
| Sampling period (ns) | | | | | | | |
| Array | 74 | 100 | 126 | 178 | 230 | 334 | 438 |
| Booth | 51.25 | 65.75 | 80.25 | 109.25 | 138.25 | 196.25 | 254.50 |
| Wallace | 71.50 | 105 | 140 | 214 | 292.5 | 459.59 | 637 |
| Bit-serial | 60 | 90 | 120 | 180 | 240 | 360 | 480 |
| Circuit size (CLBs) | | | | | | | |
| Array | 64 | 144 | 256 | 576 | 1024 | 2304 | 4096 |
| Booth | 59 | 124 | 213 | 463 | 809 | 1789 | 3153 |
| Wallace | 91 | 211 | 379 | 859 | 1531 | 3451 | 6139 |
| Bit-serial | 36 | 56 | 76 | 116 | 156 | 236 | 316 |

Table 2.1: Performance comparison of bit-parallel array multiplier, Booth's multiplier, Booth-encoded Wallace tree multiplier and bit-serial multiplier. $N$ is the data word size.

For our bit-serial multiplier design whose details will be described in Chapter 3, circuits are implemented exclusively with single-level logic. Therefore, the clock rate becomes

$$\begin{aligned} T_{clk} &= t_{lut} + t_{ff} \\ &= 7.5 \text{ (ns)}, \end{aligned} \tag{2.7}$$

and therefore the data sampling period becomes

$$\begin{aligned} T_{serial} &= T_{clk} \cdot N \\ &= 7.5N \text{ (ns)}. \end{aligned} \tag{2.8}$$

Also, our bit-serial multiplier needs 5 CLBs per bit except for the two edge cells which need only 3 CLBs each. Therefore,

$$A_{serial} = 5N - 3 \text{ (CLBs)}. \tag{2.9}$$

Table 2.1 shows the data sampling period (Fig.2.9(a)), the circuit size (Fig.2.9(b)), and area-time constant derived by the product of the data sampling period and the circuit size (Fig.2.10), for the three types of bit-parallel multipliers and our bit-serial multiplier on Xilinx 4000-5 FPGAs. We can see that the bit-serial multiplier is not only smaller but also has a fairly comparable sampling period. The readers may find it odd that the bit-serial multiplier has such performance. A simple explanation for this is that the bit-parallel multipliers calculate the whole product in one cycle whereas bit-serial multipliers calculate the product sequentially from the least significant bit to the most significant bit. The sampling period of the bit-serial multiplier is measured by the interval between two consecutive multiplier inputs, and therefore, new inputs arrive while the current product is being calculated. The latency, the interval between the arrival time of the input MSB and the output MSB is

$$T_{serial\_lat} = 15N - 7.5 \tag{2.10}$$

which is little less than two sampling periods. If we pipeline the bit-parallel multipliers at the bit-level into $k$ stages, their sampling period will decrease roughly by a factor of $k$ and eventually will be faster than our bit-serial multiplier. Arechbala et al. proposed a full systolic parallel multiplier where the multiplier is pipelined into $3N-1$ pipe stages, where the critical path delay becomes only a few gates [3]. Their systolic parallel multiplier design requires significant number of register elements to adjust the timing of each data bit entering the multiplier and to

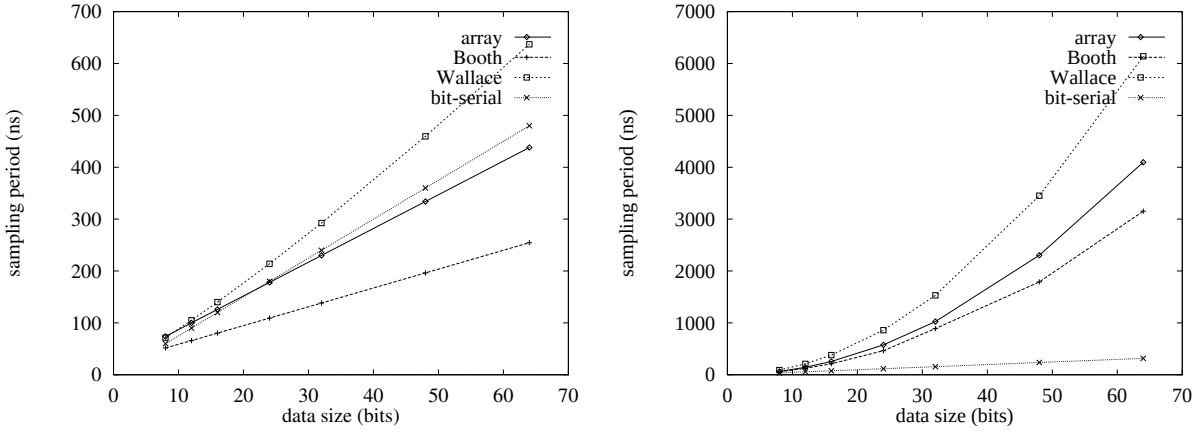



Figure 2.9: Sampling period and circuit size of various multiplier circuits.

synchronize the timing of each data bit coming out of the multiplier. The important feature of CATHEDRAL-III is this bit-level pipelining synthesis in order to maximize the throughput. However, even if they were pipelined, it will be impractical to implement bit-parallel multipliers on FPGAs because of their enormous circuit size which grows proportionally to the square of the word size ($O(N^2)$), whereas the size of the bit-serial multiplier grows linearly to the word size ($O(N)$). The sampling period of the bit-parallel multipliers is $O(N/k)$ where $k$ is the number of pipe stages and $O(N)$ for the bit-serial multiplier. In terms of efficiency which is measured by the product of area and time ($A \cdot T$), bit-parallel multipliers are $O(N^3/k)$ (although as $k$ which is the number of pipe stages grows, there will be extra registers needed to synchronize the paths with uneven pipe stages) and bit-serial multiplier is $O(N^2)$. Another measure which is frequently used to determine the complexity of the algorithm is the $AT^2$ measure [14]. In this term, $AT^2$ complexity for the bit-parallel multipliers shown here are $O(N^4/k)$ whereas the bit-serial multiplier is $O(N^3)$. In order for the bit-parallel multipliers to be competitive with bit-serial multiplier in terms of efficiency, the number of pipe stages have to be close to the word size which may not be possible in some cases.

In addition to this performance analysis which greatly favors bit-serial architecture, there are more advantages concerning the physical mapping of the circuits. In general, bit-serial operators are low fan-in low fan-out circuits, with the exception of parallel-serial converters and saturation circuits (please refer to Chapter 3 for details). The control signals are distributed by shift register network so that each signal is only used locally. Also, the connectivity of the bit-serial operators are low in normal circumstances. The data is transmitted using one to three wires plus a control signal indicating the word boundary. This low connectivity is extremely advantageous on both routing and partitioning. In particular, this class of circuits can be routed quite efficiently on the FPGAs having a 2-dimensional routing grid consisting of many short length wires and switch boxes as seen in Xilinx 3000 and 4000 series FPGAs. A bit-serial design even with the CLB utilization of 100% have never been encountered throughout this thesis study (please refer to Chapter 6).

From the above observations, the author firmly believes that the development of the bit-serial datapath synthesis system for FPGA-based configurable systems is the solution to the ultimate question of how to utilize the programmable logic resources efficiently and convert them into effective computational power.

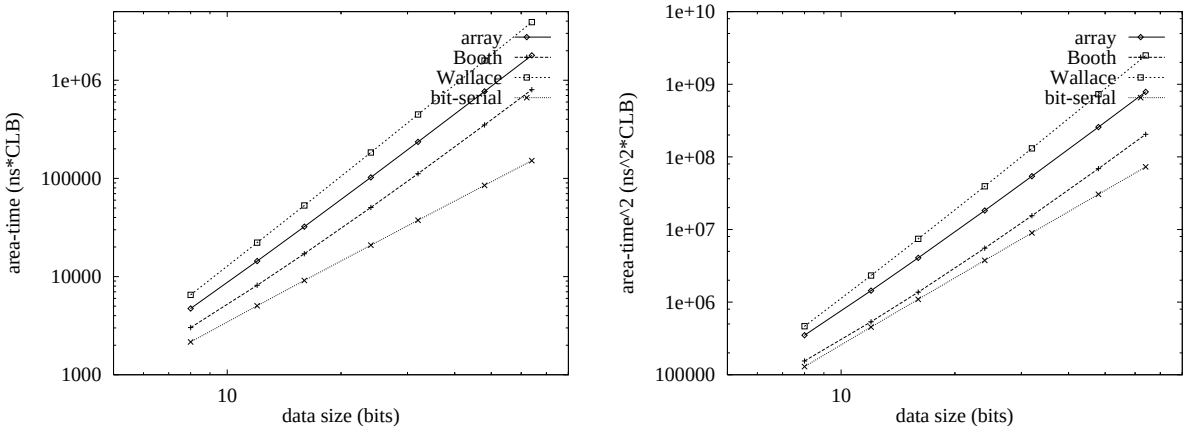



Figure 2.10: Area-time complexty of the various multiplier designs. The left graph shows the $AT$ complexity and the right graph shows the $AT^2$ complexity.

### 2.5.2 Bit-serial Vs. Signed Binary Digits

Signed digit number representation is a redundant system in which a number can be expressed in several ways. In particular, a radix-2 signed digit number uses two bits to represent one radix-2 digit. The advantage of this redundancy is that the arithmetic operations do not need to propagate the carry information from the LSB to the MSB since it is already amortized in the representation. Therefore, using this number system, it is possible to design arithmetic circuits which operate faster than the conventional bit-parallel circuits. Bht serial version of this signed binary digit arithmetic is called *on-line arithmetic* [72], [36], ,[61]. The main advantage over conventional bit-serial arithmetic is that the order of the bits can be either MSB-first or LSB-first, and that the operation latency is less than the bit-serial for complex operations such as multiplication, although bit-serial addition has a latency of only one clock whereas on-line arithmetic has a latency of two clocks. For applications where there are loops in the computation, on-line arithmetic usually can achieve higher throughput than conventional bit-serial arithmetic. Moran et al. presents a set of on-line arithmetic circuit designs on FPGAs in [72]. Also, an iterative computation methods for implementing division, square and trigonometric functions use on-line arithmetic. There are some drawbacks, however:

1. Since they use two wires for each bit representation, the on-line arithemetic circuits tend to be roughly twice as complex and slow as the bit-serial circuits for adders and multipliers.

2. The redundant system has an inherent problem of *representation overflow* in which the circuit creates an overflow even though the value is still within the range. In order to prevent this representation overflow, a *redundancy reduction circuit* is needed.

3. A dedicated circuit for converting signed digit numbers to 2's complement numbers is needed.

4. Double precision numbers are not supported in the existing on-line arithmetic systems, and therefore the computation accuracy is a concern. Even if we were to implement double precision multipliers and adders using on-line arithmetic, the circuit becomes more complex and slower compared to bit-serial circuits.

Therefore, as long as there is not a strong need for low latency circuits, bit-serial circuits are usually more compact, faster and more accurate for many of the important operations such as addition, subtraction and multiplication.

## 2.6 Datapath Design Capture in C++

There are already a number of design capture methods using a subset of high-level programming languages such as Ada, Fortran, and C for hardware designs [33], [31], [87], [1], [94], [42], [53]. In these cases, the input description is simply treated as a hardware-description language (HDL) code. One obvious advantage of using these commonly used high-level programming language for behavioral descriptions of hardware is the *portability* of the design descriptions. From the application designer's point of view, the easy transition from the algorithm level which is often designed using such high-level programming languages to the hardware level is also a large factor. In these design capture methods, the hardware description written in a high-level language is fed into the compiler and transformed into an intermediate format which essentially represents the control-data flow graph of the computations. This intermediate format is then subjected to the behavioral synthesis which we have described earlier. Some use the standard compiler where the hardware description code is composed of only standard semantics which prohibits the specifications of various hardware related parameters such as signal word size, memory configuration, sampling period, IO interface format, etc. Others have their customized compiler in which the code includes several specialized statements to specify the various hardware parameters. Also in most cases, the use of some portions of the programming constructs provided in these high-level programming languages are prohibited which may cause restrictions in constructing the hardware description.

A rather different approach is chosen in this study. Instead of relying on the compiler to parse the code to extract the control-data flow information, the hardware description is done by object function calls on C++. The code is compiled using the standard GNU C++ compiler (`g++`) which produces an executable file. This executable is both the circuit generator and the simulator. And as in any executable files, the circuit generator can be parameterized from the user interface after the compilation. This compilation flow differentiates from other high-level programming tools where the code is compiled into a *netlist* instead of a circuit generator. This compilation flow is quite similar to the `wirec` circuit design capture system [71] where they generate C++ codes from their schematic capture and the executable file compiled from these codes generates a circuit netlist which can be used by the logic simulator. Development tools for PAM configurable systems also uses C++ as a circuit-level design tool [9]. I have applied their approaches to the high-level hardware design paradigm. In our case, the executable file also acts as a high-level behavioral simulator in which any kinds of C++ statements and functions to facilitate the algorithm verifications such as reading an image file and displaying the simulated results back to the screen are allowed. The compilation flow of the existing high-level synthesis systems and our C++ function-based synthesis system is illustrated in Fig.2.11. In this study, special emphasis was directed in making the programming semantics similar to an ordinary C++ programming by utilizing the operator overloading feature of C++. For instance, consider the following code:

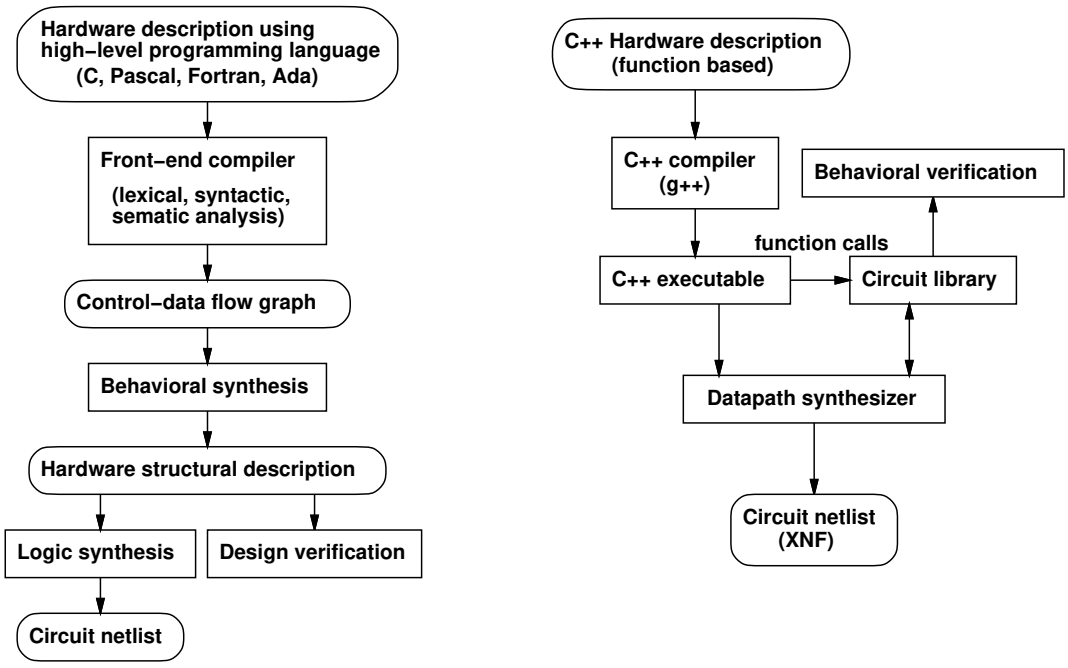
```
Single a, b, c, d;

a = b + c + d;
```

Variables are the members of a *customized hardware object class* `Single` which declares single precision signals on the configurable hardware. The equation is a collection of overloaded operators whose arguments are the members of class `Single`:

```
SIG0 = operator+(b, c);
SIG1 = operator+(SIG0, d);
a = SIG1;
```

The function `operator+(b, c)` is called when the C++ compiler encounters the statement "`b + c`". This function establishes a data structure which describes the data flow graph of the computation and returns an automatically generated signal `SIG0` which is connected to the output of the adder. Next, the compiler will encounter




(a) Compilation flow of existing high−level synthesis systems.

(b) Compilation flow of our C++ function−based synthesis system.

Figure 2.11: Compilation flows of existing high-level synthesis systems and our C++ function-based synthesis system.

the statement "`SIG0 + d`" which calls another overloaded operator function amd returns another internally created signal `SIG1`. Finally, the statement `a = SIG1` is reached in which the signal `a` is connected to the output of the second adder. By utilizing the overloading operators, it is also possible to describe control-flows using the ordinary C++ statements. In the current implementation of the tool, this feature is quite limited and can only handle nested for-loops with a fixed loop counts. The details are described in Chapter 4.

# Chapter 3

# High-Speed Bit-Serial Circuit Design on FPGAs

## 3.1 Principle of Bit-Serial Operations

### 3.1.1 Number Representation and Number Precision

Fixed-point 2's complement binary numbers are used as the number representation. For external data, the decimal point is set explicitly between the most significant bit (MSB) and the second most significant bit. The value of an $N$-bit 2's complement number $X = (x_{N-1}.x_{N-2} \cdots x_1 x_0)$ is then calculated as

$$X = -x_{N-1} + \sum_{i=0}^{N-2} x_i 2^{i-N+1} \tag{3.1}$$

where its value range is $[-1, 1-2^{1-N}]$. Bit-serial data bits are transmitted in the least significant bit first (LSBF). The number of cycles it takes to transmit one data is called the *data sampling period* which also determines the size of the data word $N$.

A total of four types of signals (*single precision, double precision, extended-single precision, extended-double precision*) are provided in order to represent binary numbers with various dynamic ranges and various fraction precisions which uses different number of data wires.

1. **Single precision** : This signal has the dynamic range of $[-1, 1-2^{1-N}]$ with the precision of $2^{1-N}$ which uses one data line (*high*).

2. **Double precision** : This signal has the dynamic range of $[-1, 1-2^{1-2N}]$ with the precision of $2^{1-2N}$ which uses two data lines (*high, low*).

3. **Extended-single precision** : This signal has the dynamic range of $[-2^N, 2^N - 2^{1-N}]$ with the precision of $2^{1-N}$ which uses two data lines (*overflow, high*).

4. **Extended-double precision** : This signal has the dynamic range of $[-2^N, 2^N - 2^{1-2N}]$ with the precision of $2^{1-2N}$ which uses three data lines (*overflow, high, low*).

The values of the four signal types are calculated as:

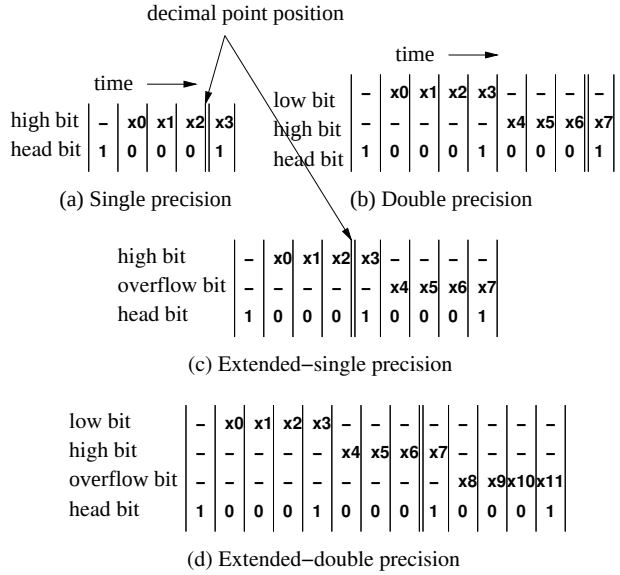



Figure 3.1: Timing of data bits for various number precisions. (data word size : $N = 4$.)

$$X = \begin{cases} -x_{N-1} + \sum_{i=0}^{N-2} x_i 2^{i-N+1} & (\text{ single precision}) \\ -x_{2N-1} + \sum_{i=0}^{2N-2} x_i 2^{i-2N+1} & (\text{ double precision}) \\ -x_{2N-1} + \sum_{i=0}^{2N-2} x_i 2^{i-N+1} & (\text{ extended-single precision}) \\ -x_{3N-1} + \sum_{i=0}^{3N-2} x_i 2^{i-2N+1} & (\text{ extended-double precision}) \end{cases} \tag{3.2}$$

### 3.1.2 Distribution and Timing of Control Signals

Control signals are needed to indicate the word boundary for each bit-serial operator. This control signal becomes active every $N$ cycles for exactly one cycle long. It can be generated by a shift register ring and broadcast throughout the chip, or distributed by a chain of shift registers (Fig.3.2). A centralized control using a single shift register ring is very efficient in terms of logic resource usage since it only needs one shift register ring with a length of $N$. But this could suffer a severe routing problem due to large fan-out to every bit-serial operators causing a performance degradation due to large routing delays and low logic utilization. Furthermore, the circuit speed and the required logic resource can vary greatly for different datapath designs having different structures which makes the performance prediction and hardware size estimation extremely difficult before the actual layout. On the other hand, distributed control using a chain of shift registers causes little problem in routing because the control signals tend to be physically located close to the corresponding data signals. Routability is significantly improved,

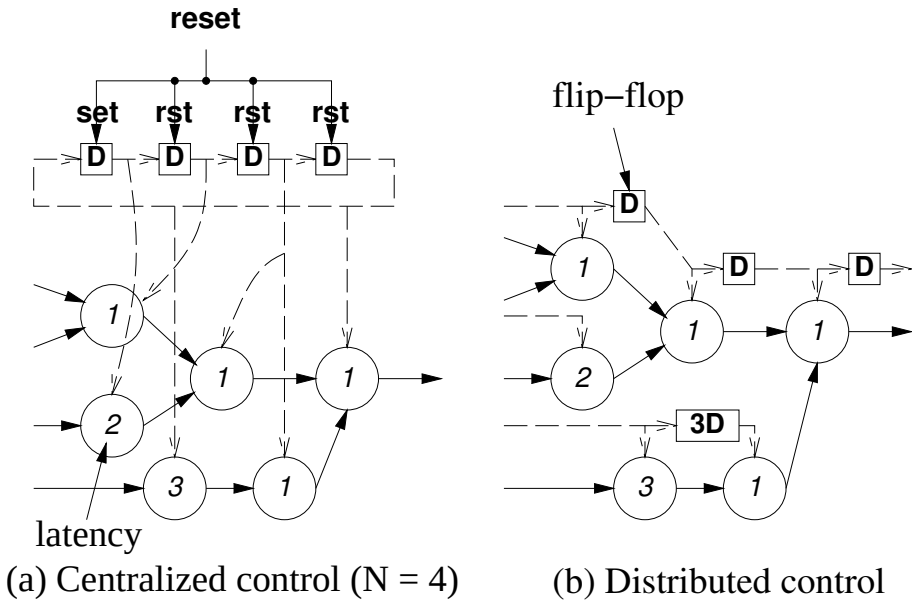



Figure 3.2: Control scheme for bit-serial operators.

thus raising the logic utilization to its maximum, and also keeping the routing delay of the control signals to a minimum which speeds up the circuit and makes the performance prediction easier. The drawback is the evident high consumption of the flip-flops. The distributed control signal scheme was chosen for its advantages of enabling accurate hardware size prediction and performance prediction, and also for low routing penalty on performance. The bit-serial operators are responsible not only to calculate the output bits but also to generate the control bit for the output data, this usually requires one or two extra flip-flops for each operator. These control signals localized to each data bits are called the *head bits*.

For the timing of the control signal relative to the data signal, most bit-serial multiplier designers have used the timing where the control signal becomes active at the same time as the LSB is present at the data wire [68][11]. A different timing is used where the control signal becomes active 1 clock cycle before the LSB arrives (Fig.3.1).

### 3.1.3 Operation Latency

Operation latency is often measured from the arrival time of the first bit of the input to the time the first bit of the output is produced. Here, the decimal point position is used instead of the first arriving bit as the reference point for latency measurement. In other words, *latency* is defined as (Fig.3.3)

- *Latency* $\delta$ is the time difference between the arrival of the $(N-1)$-th least significant bit *after the decimal point* at the input and the production of the $(N-1)$-th least significant bit *after the decimal point* at the output.

Our definition of latency localizes the inherent latency of rounding double precision variable to a single precision variable to the very operation which produced the double precision variable in the first place. This latency measure better reflects the nature of each bit-serial operations.

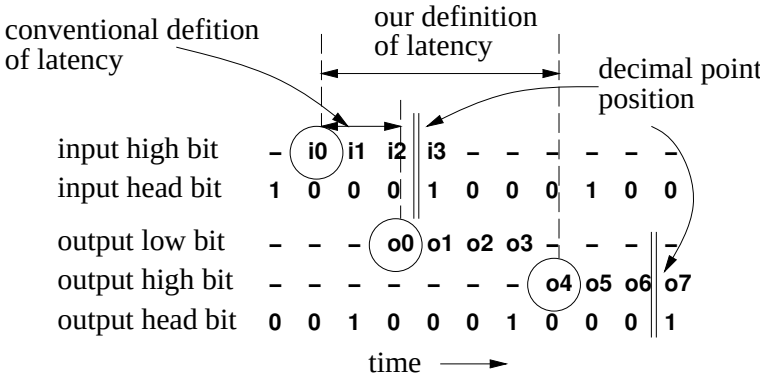



Figure 3.3: Difference in the conventional definition of latency and our new definition of latency. This figure show the case where $N = 4$ with single precision input and double precision output. With the conventional definition, the latency becomes $\delta = 2$. With our new definition, the latency becomes $\delta = 6$.

## 3.2 Detail Description of Bit-Serial Operators

In this section, a wide variety of bit-serial operators are presented. Readers should note that the circuit designs shown in the subsequent diagrams do not reflect the actual circuit implementation, but only the circuit behavior. All the circuits presented are implemented with single-level LUT (lookup table) logic where LUT is assumed to have up to 5 inputs. Fig.3.4 shows the diagram of a logic block (CLB) in Xilinx 3000 FPGA which includes such LUT elements. In this CLB, the LUT can be configured as either a 4-input 2-output LUT or a 5-input 1-output LUT. These single-level LUT logic circuits enables high speed clock operation and low routing delay.

### 3.2.1 IO Data Conversion

It is assume bit-parallel data external to the bit-serial datapath considering the compatibility between the existing parallel bus systems for PCs and workstations. Conversion from external bit-parallel data to internal single precision bit-serial data is implemented by a parallel-serial shift register (Fig.3.5). With a cycle period $T$, up to $T$ bit-parallel data inputs can share the same bus. Bit-serial control signals for each data input are generated here. Similarly, conversion from internal single precision bit-serial data to external bit-parallel data is implemented by a serial-parallel shift register (Fig.3.6). One note here is that all the data coming into the serial-parallel converter have to be synchronized. Or in other words, LSBs of all the data have to arrive at the same clock cycle. Up to $T$ bit-parallel data outputs can share the same bus. However, if the data inputs and outputs uses the same bus, the total number of inputs and outputs combined cannot exceed $T$, or else it will cause a congestion on the parallel bus. Also, the word size of the bit-parallel data cannot exceed $T$, or else the bit-parallel data will not be captured properly by the parallel-serial converter. It is important to note here that even though the data sampling rate is limited to $f/T$ where $f$ is the clock frequency in the bit-serial datapath, the IO bandwidth is not limited in this term using these IO data converters. This feature is especially important when multiple data inputs are required to process one or more data outputs within one data sampling period, such as 2-D filters and various block processing algorithms including Discrete Cosine Transform (DCT). Bit-serial datapath can be powerful in IO intensive applications where the IO bandwidth is relaxed by a factor of $T$. The figures here show the conversion between bit-parallel data and bit-serial data in LSBF order.

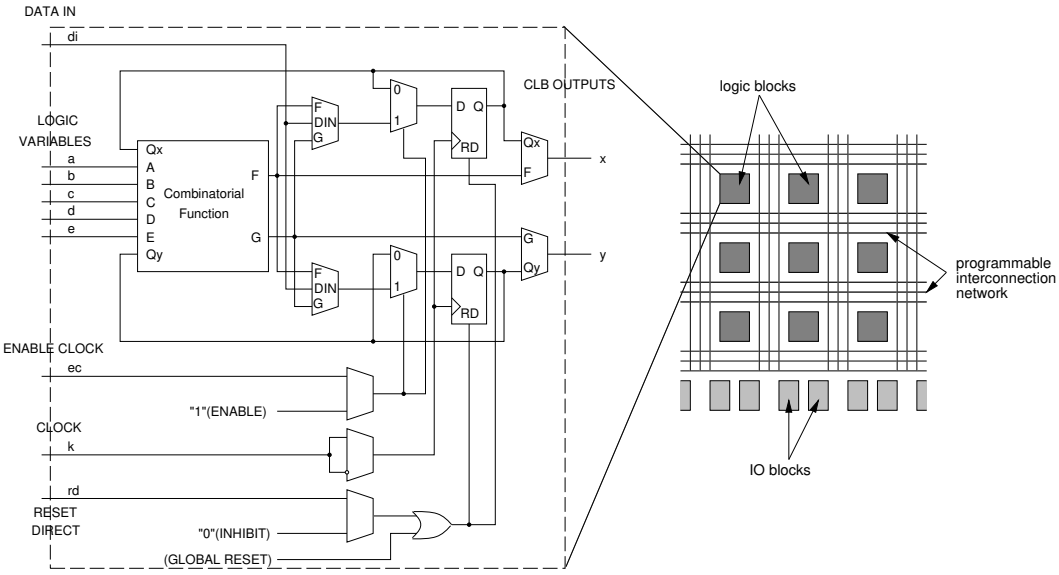
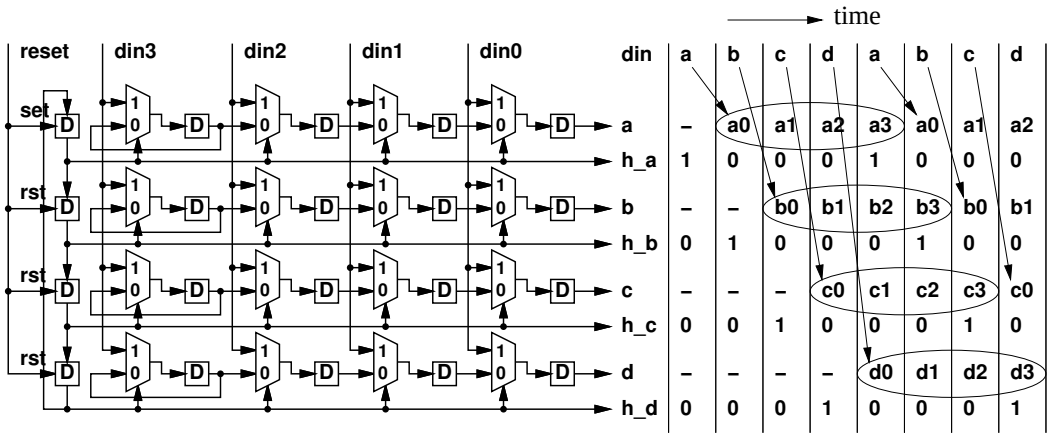
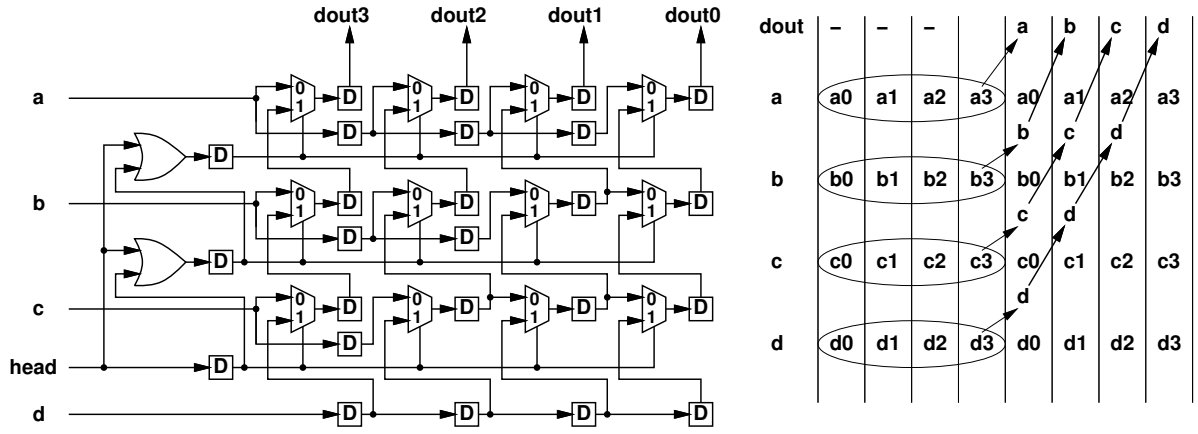



Figure 3.4: Xilinx 3000 FPGA architecture.




(a) Parallel–serial converter (b) Signal timing of parallel–serial converter

Figure 3.5: Data conversion circuits for converting bit-parallel data into bit-serial data. In this circuit, four bit-parallel data are written on the input port in four consecutive cycles. The LSBs of the bit-serial outputs appear one cycle after their bit-parallel inputs are written on the bit-parallel port.




(a) Serial–parallel converter (b) Signal timing of serial–parallel converter

Figure 3.6: Data conversion circuits for converting bit-serial data into bit-parallel data. The four bit-serial input data into this converter is synchronized. The four bit-parallel output data are presented at the bit-parallel port in four consecutive clock cycles.

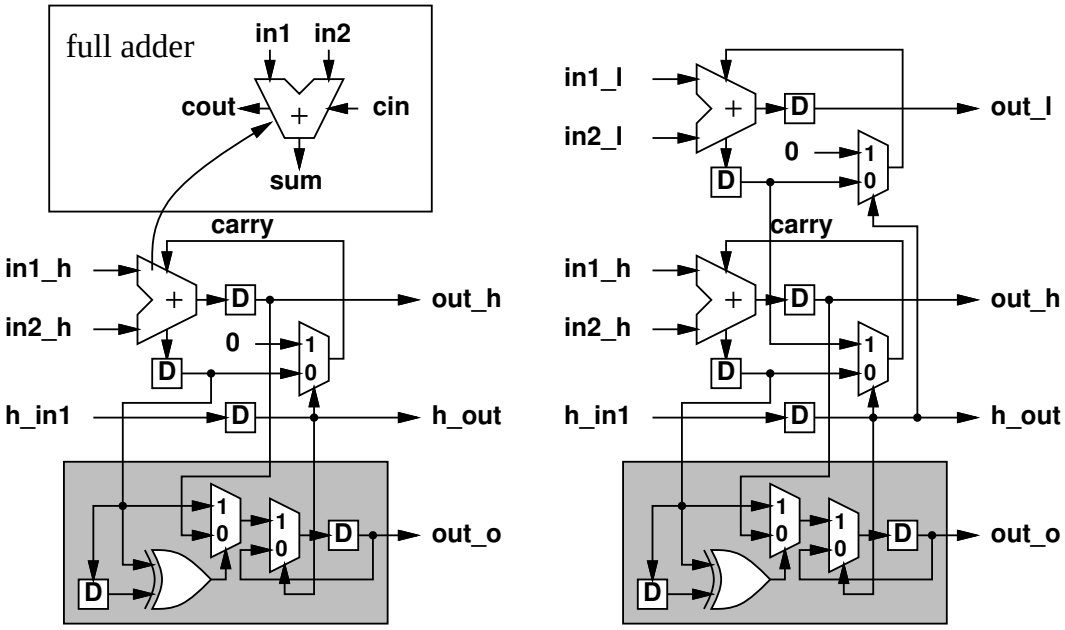
### 3.2.2 Addition and Subtraction

Fig.3.7(a) shows the single precision adder with overflow extension. Using the full adder circuit, the carry output is latched, then multiplexed with 0 controlled by the delayed head bit (*h_out*), and fed back to the carry input. The overflow extension generator, shown in the hashed box in the figure, detects the overflow by taking the EX-OR of the carry bit and the previous carry bit, and selects either the carry bit or the sum bit for sign extension. For the bit-serial subtracter, the second input (*in2*) is negated and the default carry input is set to 1 (Fig.3.8(a)). Here, I wish to make a note here concerning the sequential logic circuit designs in general on FPGAs. In any sequential circuits, there is a default state, or an initial state in which the circuit is forced to fall into sometime after the power up of the hardware. This is achieved by feeding some kind of an initialization vector into the circuit inputs. On FPGAs, when a configuration bitstream is downloaded on the chip, all the flip-flops are reset to zero. Therefore, right after the power, carry input to the subtracter is set to 0, which is not the default value. This means that this circuit will not function correctly for the first sampling data. In order to avoid this, the carry bit of the subtracter is made to be *active-low* instread of *active-high*. Therefore, the carry output and carry inputs to the circuits are negated(Fig.3.8(b)).

A double precision adder is composed of two bit-serial adders where the default carry input of the second adder is the carry output of the first adder (Fig.3.7(b)). Extended-single precision adder is identical as the double precision adder.

### 3.2.3 Bit Shifting

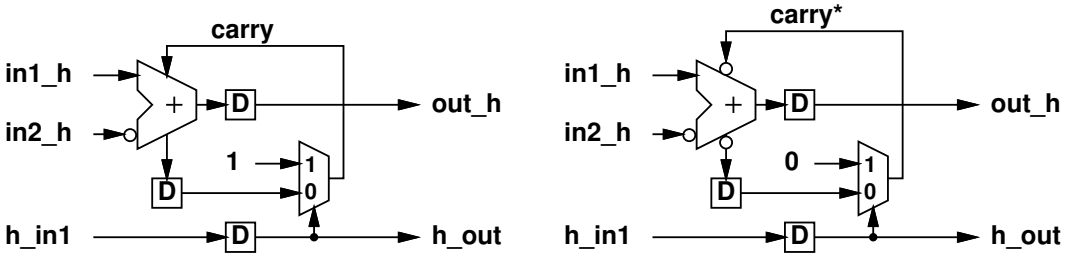
A $k$-bit shift-right operation is described as $Y = 2^{-k} \cdot X$ is implemented by the circuit described in Fig.3.9. The select signals for the multiplexers are generated by the circuit described in Fig.3.9(a). This select signal goes active for a period $k$ cycles at the same time the LSB arrives. Fig.3.9(b) shows the $k$-bit shift-right circuit for double precision variables, and Fig.3.9(c) show the timing of the input and output signals. The latency $\delta$ is $k+1$ cycles. For a single precision variables, signal *in_l* in Fig.3.9(c) with 0. In this case, the LSB of the double precision output is produced before the arrival of the input LSB, however, by our definition of operation latency, $\delta$ remains $k+1$




(a) Single precision adder with overflow extention

(b) Double precision adder with overflow extention

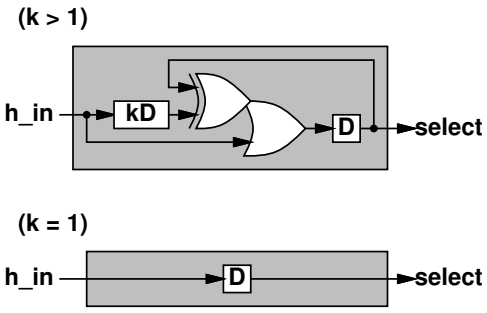
Figure 3.7: Bit-serial adder circuits. Hashed boxes shows the overflow extension circuit. Latency : $\delta_{add} = 1$.




(a) Single precision subtractor with active–high carry.

(a) Single precision subtractor with active–low carry.

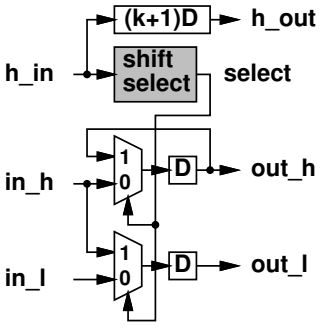
Figure 3.8: Bit-serial subtracter circuits.



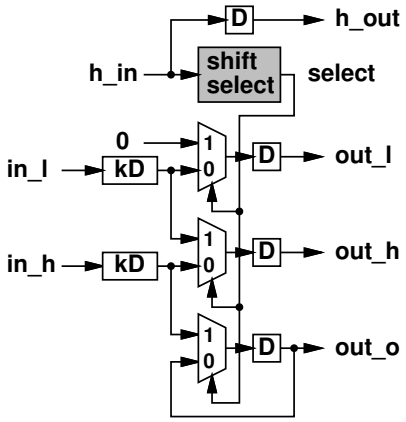

(a) k–bit shift select bit generator

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| in_l | – | i0 | i1 | i2 | i3 | – | – | – | – | – | – | – |
| in_h | – | – | – | – | – | i4 | i5 | i6 | i7 | – | – | – |
| h_in | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| select | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| out_l | – | – | – | – | i2 | i3 | i4 | i5 | – | – | – | – |
| out_h | – | – | – | – | – | – | – | – | i6 | i7 | i7 | i7 |
| h_hout | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

(c) Timing of input and output signals for double precision 2–bit shift–right




(b) Double precision k–bit shift–right




(d) Double precision k–bit shift–left with overflow extension

Figure 3.9: Bit-serial $k$-bit shift-right and shift-left circuits. Latency : $\delta_{SR} = k + 1$ ($k$-bit shift-right), $\delta_{SL} = 1$ ($k$-bit shift-left).

instead of a negative number.

$k$-bit shift-left operation described as $Y = 2^k \cdot X$ is implemented by the circuit described in Fig.3.9(d). The figure describes the circuit for double precision variables with overflow extension. The latency $\delta$ is 1.

### 3.2.4 Multiplication

Bit-serial multiplication have been investigated by many researchers until now [68][73][11][52]. The criteria of the quality of the bit-serial multiplication schemes are

1. Ability to handle 2's complement number.
2. Ability to produce double precision product.
3. Operation clock frequency.
4. Throughput.
5. Latency.
6. Circuit complexity.

Among them, Algorithm proposed by Bi and Jones [11] stands high among others for satisfying the first four criteria. Other multiplier designs have the drawback of low throughput, requiring $2N$ cycles to produce the $2N$-bit product [52] or needs an extra idle cycle in addition to the minimum $N$ cycles for correct 2's complement multiplication operation [68][73]. Their scheme is able to handle 2's complement numbers, able to produce $2N$-bit double precision products every $N$ cycles, and maximizes the operation clock frequency by exploiting high degree of pipelining using multiplexer-based one-level logic. However, their scheme suffers from a large latency of the deep pipeline stages. Its takes $2N + 4$ cycles to produce the LSB of the double precision product. Also, their scheme of handling the sign extension of the partial product makes the circuit unnecessarily complicated. This complicated sign extension circuit and the deep pipeline stages consume large logic resource, requiring 21 flip-flops per bit.

A new bit-serial multiplication algorithm is proposed here which have the following features:

1. For two $N$-bit 2's complement bit-serial binary inputs, it is capable of calculating a $2N$-bit double precision 2's complement bit-serial product every $N$ cycles.
2. The latency from the first LSB inputs to the first LSB output is $N - 1$ cycles ($\delta = 2N - 1$).
3. Critical path is kept to a minimum using one-level LUT logic for maximum clock frequency.
4. Requires 10 flip-flops per bit and can be implemented by 5 CLBs (Xilinx 3000 series Configurable Logic Block).

#### Basic Operation of 2's Complement Bit-Serial Multiplication

Lyon's bit-serial multiplication algorithm [68] forms the basis of most of the other proposed multiplication algorithms, including ours. There are $N$ processing elements (PEs) cascaded into a one-dimensional systolic array which implement this algorithm. Let $X = (x_{N-1}.x_{N-2} \cdots x_0)$ be the multiplicand, $Y = (y_{N-1}.y_{N-2} \cdots y_0)$ be the multiplier, and $Z = (z_{2N-1}.z_{2N-2} \cdots z0)$ be the double precision product. $PE[i]$ is responsible for calculating the partial product $S_{(i+1)}(i = 1, 2, \cdots, N)$ as in the following recursion:

$$\begin{array}{lcll} S_{(0)} & = & 0, & \\ S_{(i+1)} & = & (S_{(i)} + X \cdot y_i) \cdot 2^{-1} & (i = 0, 1, \cdots N-2), \\ S_{(N)} & = & (S_{(N-1)} - X \cdot y_{N-1}). & \end{array} \tag{3.3}$$

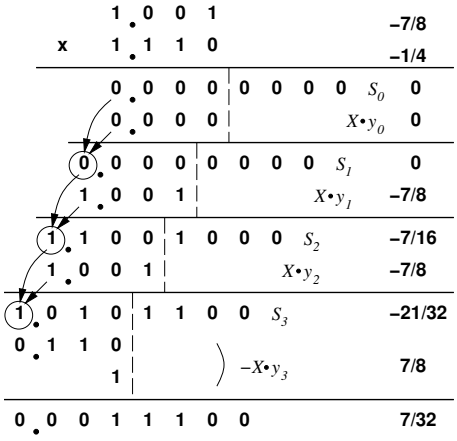



Figure 3.10: 2's complement multiplication. $((-\frac{7}{8}) \times (-\frac{1}{4}) = \frac{7}{32})$

This algorithm can be easily proven that it produces correct output $S_N$ by unfolding the recursion as

$$\begin{aligned} S_{(N)} &= -X \cdot y_{N-1} + S_{(N-1)} \\ &= -X \cdot y_{N-1} + \sum_{i=0}^{N-2} X \cdot y_i \cdot 2^{i-N+1} \\ &= X \cdot (-y_{N-1} + \sum_{i=0}^{N-2} y_i \cdot 2^{i-N+1}) \\ &= X \cdot Y. \end{aligned} \tag{3.4}$$

Let $S_{(i)h}$ and $S_{(i)l}$ be the upper and lower half of the partial product $S_{(i)} = (s_{(i)2N-1}.s_{(i)2N-2} \cdots s_{(i)0})$:

$$\begin{aligned} S_{(i)h} &= (s_{(i)2N-1}.s_{(i)2N-2} \cdots s_{(i)N}), \\ S_{(i)l} &= (0.0 \cdots 00 s_{(i)N-1} s_{(i)N-2} \cdots s_{(i)0}). \end{aligned} \tag{3.5}$$

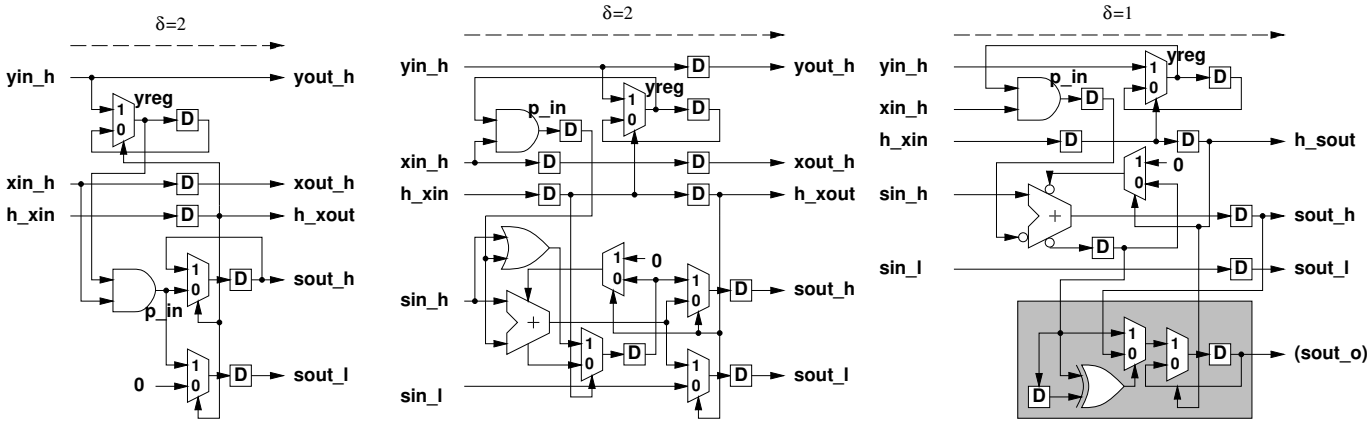
Then the first recursion in eq.(3.3) is rewritten as

$$\begin{aligned} S'_{(i)h} &= S_{(i)h} + X \cdot y_i \\ S_{(i+1)} &= (S'_{(i+1)h} + S_{(i)l}) \cdot 2^{-1} \end{aligned} \tag{3.6}$$

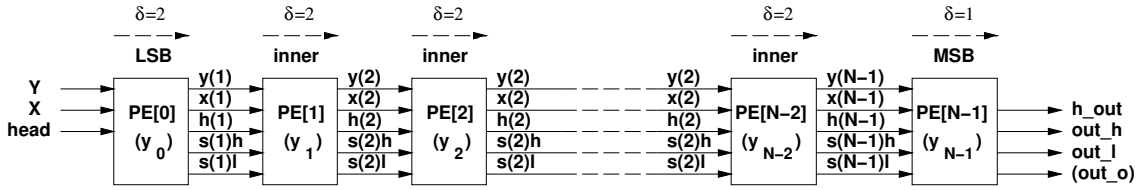
The handling of the 2's complement addition $S'_{(i)h} = S_{(i)h} + X \cdot y_i$ needs special attention because of the overflow. $S'_{(i)h} = (s'_{(i)2N} s'_{(i)2N-1}.s'_{(i)2N-2} \cdots s'_{(i)0})$ requires $N+1$ bits to represent its true value. The existing bit-serial multiplication algorithms either requires extra idle cycles to prevent the overflow [68][73][52], or provides a circuit dedicated to calculate the proper sign bit [11]. However, simply attaching an overflow extension generator shown in the hashed boxes in Fig.3.7 would do the job.

- **Sign bit calculation by overflow extension**

  1. If the overflow does not occur, extend the sign bit $s'_{(i)2N}$ from $s'_{(i)2N-1}$.
  2. Otherwise set the sign bit $s'_{(i)2N}$ as the carry output.




(a) LSB multiplier cell (PE[0]) (b) Inner multiplier cell (PE[i](i = 1, 2, ... , N−2)) (c) MSB multiplier cell (PE[N−1])




(d) 2's complement bit−serial multiplier

Figure 3.11: 2's complement bit-serial multiplier. Latency : $\delta_{mult} = 2N - 1$.

Although this implementation is significantly simpler than Bi and Jones implementation of the sign bit calculation, this can be simplified further more and be implemented with only a single OR gate.

- **Sign bit calculation by logical-OR of input sign bits**

  1. If either $X \cdot y_i$ or $S_{(i)h}$ are negative ($x_{N-1} \cdot y_i \cup s_{(i)2N-1}$), set the sign bit $s'_{(i)2N}$ to 1.
  2. Otherwise, set to 0.

This algorithm is justified by the following argument:

- If both $S_{(i)h}$ and $X \cdot y_i$ $(i = 0, 1, \cdots, N-2)$ are non-zero, their sign bits are identical since $S_{(i)h} = X \cdot Y_i$ where $Y_i = \sum_{j=0}^{i-1} y_j \cdot 2^{j-i}$ is non-negative.
- If either $S_{(i)h}$ or $X \cdot y_i$ are negative, neither of them can be positive, therefore their sum $S_{(i+1)h}$ is always negative.
- If both $S_{(i)h}$ and $X \cdot y_i$ are non-negative, then their sum $S_{(i+1)h}$ is always non-negative.

This implementation of sign bit generation for 2's complement bit-serial multiplication imposes *no* extra hardware from the unsigned bit-serial multiplier.

The second recursion in eq.(3.3) $S_{(N)} = S_{(N-1)} - X \cdot y_{N-1}$ requires only a single precision adder. This addition does not create an overflow for all cases but one: when both operands are -1 ($1.00\cdots0$), the correct product is 1 ($01.00\cdots0$) which cannot be represented by a double precision number. Although in many cases this can be guaranteed to never happen by proper variable scaling, if this cannot be avoided, and if this overflow cannot be ignored, the overflow extension generator is simply attached at the last MSB cell. Fig.3.10 illustrates our bit-serial multiplication algorithm.

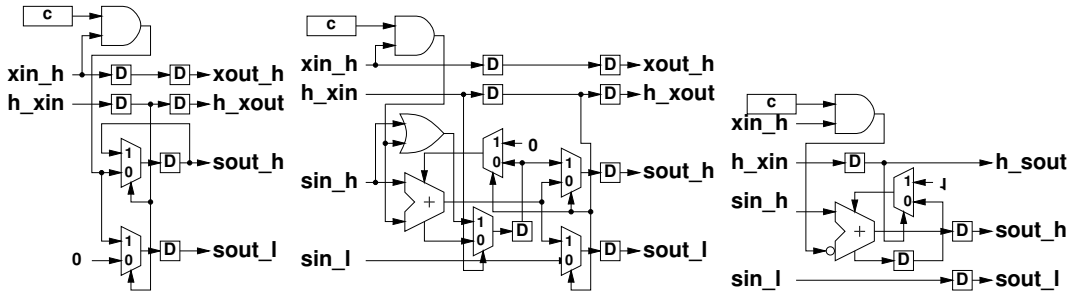
### Implementation

$PE[i](i = 0, 1, \cdots, N-1)$ each captures the multiplier bit $y_i$ corresponding to its position $i$ into its register. The multiplicand bits $x_j(j = 0, 1, \cdots, N-1)$ flows through the PE. The left-hand-most PE ($PE[0]$) only has an AND gate computing the product $p\_in = x_j \cdot y_0$ and a 1-bit shift-right operator (Fig.3.11(a)). This can be implemented with 2.5 CLBs. The inner PE ($PE[i](i = 0, 1, \cdots, N-2)$) has an AND gate computing the product $p\_in$ whose output is latched, a single precision adder with a modification on carry generation, followed by a double precision 1-bit shift-right operator (Fig.3.11(b)). This can be implemented with 5 CLBs. Note here that since the product output $p\_in$ is latched, the multiplier bit $yin\_h$ and multiplicand bit $xin\_h$ are one cycle ahead of the two partial product bits $sin\_h$ and $sin\_l$ in order to synchronize the two data at the adder. The right-hand-most PE ($PE[N-1]$) consists of an latched AND gate and a single precision adder. Note here that since this cell includes a subtracter whose default carry value is 1, this carry is made active-low. This can be implemented with 4 CLBs. If an overflow extension generator is needed, this adds 1 additional CLB.

As a whole, it requires $5N-3$ CLBs ($5N-2$ CLBs for extended-double product) to implement a $N$-bit bit-serial multiplier, and the latency $\delta$ is $2N-1$. All circuits are implemented in single-level LUT logic.

### Register Sharing

As we can see in out bit-serial multiplier design, the shift-registers for the multiplicand ($xin$) and this bit-serial control signal ($hin$) occupy 40% of the circuit. If there are two multipliers which share the same multiplicand, these multipliers can share the same shift-registers which reduces the hardware size by 20% so that the two bit-serial multiplier cell can be merged into 8 CLBs. This register sharing technique is used in the FIRST bit-serial compiler for implementing a complex number multiplier which is composed of four multipliers and a pair of adder and




(a) LSB scaler cell (PE[0])

(b) Inner scaler cell (PE[i](i = 1, 2, ... , N−2))

(c) MSB scaler cell (PE[N−1])

Figure 3.12: Bit-serial scaler cells.

subtracter [73]. The register sharing is only supported for up to two multipliers, since it requires at least 4 CLBs to implement one bit-serial multiplier cell in the first place and therefore does not contribute in further reducing the hardware size.

Also, these shift registers can be shared with the pipeline synchronization registers needed to equalize the pipe stages for each signal path. These synchronization registers are needed in large quantity for signal processing applications where the signals are delayed for several sampling cycles.

### 3.2.5 Scaling

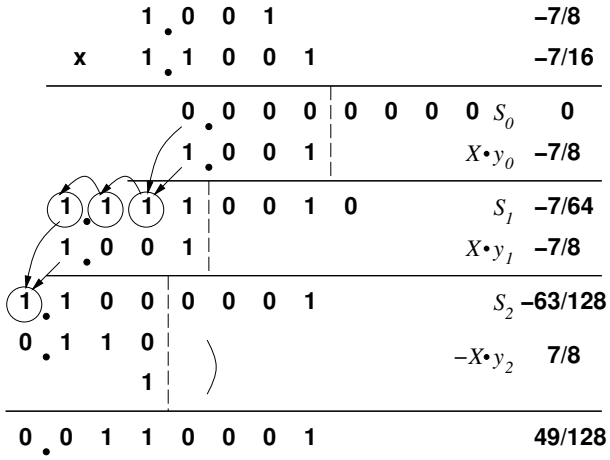
Scaling is the operation where a variable is multiplied by a constant value. The bit-serial scaler circuits can be derived from the bit-serial multiplier circuits. Since the second operand is no longer transmitted in bit-serial format, the word size of the 2's complement constant is not limited to $N$. In fact, it is possible to extended the word size of the multiplier constant to $N+1$ bits having $N$ fraction bits instead of $N-1$ bits. This increases the accuracy of the scaling operation which is important in most DSP applications.

#### Unified Cell Implementation

Signal *yin_h* in Fig.3.11 becomes a known value, and therefore the circuit for capturing the multiplier bit *yreg* and calculating the product *p_in* can be hardwired inside the LUT. In Fig.3.12, $c$ is either 0 or 1 depending on the constant which can be programmed inside the LUT. The LSB scaler cell ($PE[0]$) is implemented with 2 CLBs, the inner scaler cell ($PE[i](i = 1, 2 \cdots, N-2)$) with 4 CLBs and the MSB cell ($PE[N-1]$) with 3 CLBs (or 4 CLBs including the overflow extension). If we prohibit the constant to be -1, then the overflow extension is unnecessary. The circuit structure is independent to the constant multiplier value. Changing the constant requires only changing the configuration of the LUT. This can be beneficial for digital filter applications and neural network applications, where filter coefficients may need to be change frequently or the weights of the neuron connections has to be readjusted for different classification problems [25]. As a whole, it takes $4(N+1)-2$ CLBs to implement a $N+1$-bit constant scaler. Latency $\delta$ is $2N+1$ cycles.

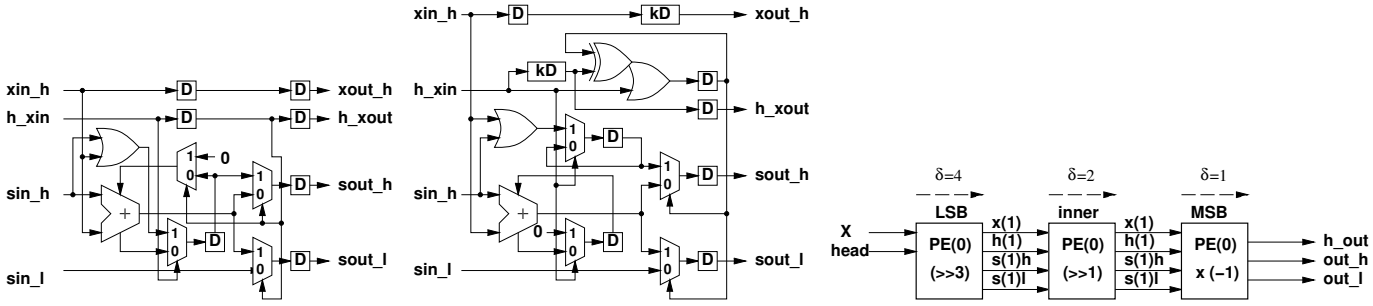
#### Customized Cell Implementation

If a frequent adjustment of the constant multiplier value is not an issue, it is possible to implement the scaler significantly more efficiently by customizing each cell structure according to the constant bits. When the constant multiplier bit $c_i$ is 0, there is no need for adders, and therefore the $i$-th PE can be eliminated. This helps not only




Scaling (1.001) by a constant (1.1001). Run–length expression of (1.1001) is (002).

Figure 3.13: Scaling (1.001) by a constant (1.1001). Run-length expression of (1.1001) is (002).




(a) 1–bit shift inner scaler cell (b) k–bit shift inner scaler cell (k = $r_i$ + 1) (c) Customized structure scaler (constant = 1.10001)

Figure 3.14: Customized cell implementation of 2's complement scaler.

to reduce the circuit area but also to reduce the operation latency $\delta$. In fact, this customized cell implementation is very similar to the scaler design used in CATHEDRAL-I bit-serial synthesis system [58]. However, they do not implement a full double precision scaler, therefore their design is very simple where they did not have to worry about multiplexing the *high* bit and the *low* bit. Let us describe the bit sequence of the constant $C = (c_N.c_{N-1} \cdots c_1 c_0)$ with a run-length code

$$(c_0, c_1, \cdots, c_{N-1}, c_N) \Rightarrow (r_0, r_1, \cdots, r_{M-2}, r_{M-1}).$$

The values $r_i (i = 0, 1, \cdots M-1)$ represent the number of consecutive 0s before reaching a non-zero bit. For example, a binary sequence (0,0,1,0,1,1,0,0) can be written as (2, 1, 0), there are two 0s before the first 1, there is one 0 before the second 1, no 0s before the third 1, and the remaining 0s are ignored because there are no more 1s in the sequence. $M$ corresponds to the number of 1s in the sequence. The sum of the run values $R = \sum_{i=0}^{M-1} r_i$ is the number of 0s up to the last 1. The first run value $r_{M-1}$ indicates the sign of the constant $C$ :

- If $r_{M-1} = 0$, $C$ is negative.
- Otherwise $C$ is positive.

Using this run-length sequence, the bit-serial multiplication recursion of eq.(3.3) can be modified as

$$\begin{array}{lcll} S_{(0)} & = & 0, & \\ S_{(i)} & = & (S_{(i)} + X) \cdot 2^{-1-r_i} & \\ & & (i = 0, 1, \cdots, M-2), & \\ S_{(M)} & = & \begin{cases} (S_{(M-1)} + X) \cdot 2^{-1-r_{M-1}} & (r_{M-1} \neq 0) \\ (S_{(M-1)} - X) & (r_{M-1} = 0) \end{cases} & \end{array} \tag{3.7}$$

The run value $r_i$ corresponds to the number of additional shift-right operations to the sum $S_{i+1}$. If $r_i$ is 0, the circuit remains the same as in Fig.3.12 with $c = 1$ (Fig.3.14(a)). If $r_i$ is greater than 0, then the scaler cell is modified by implementing the $(r_i + 1)$-bit shift-right operator, instead of 1-bit, at the partial product outputs. For the LSB scaler PE ($PE[0]$), a shift-right operator is used. For the inner scaler PEs, the circuit is shown in Fig.3.14(b) which is implemented with $4 + r_i$ CLBs.

Let $M'$ be the total number of zero run values $r_i (i = 0, 1, \cdots, M-1)$:

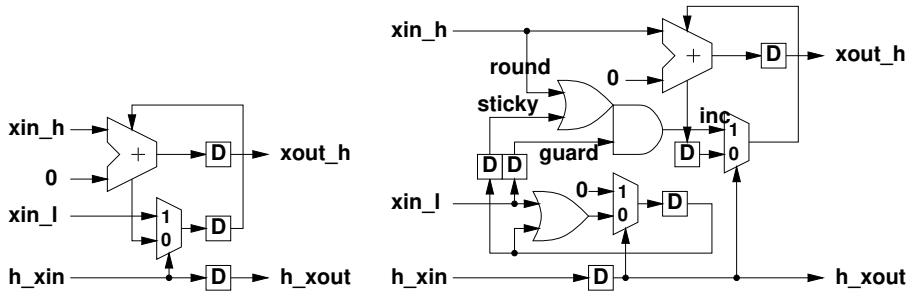
$$M' = \sum_{i=0}^{M-1} r_i. \tag{3.8}$$

The total number of CLBs are given by $4M + R + M'$ for positive multiplier constant, and $4M + R + M' - 2$ for negative multiplier constant. The operation latency $\delta$ is $2M + R$ for positive constants and $2M + R + 1$ for negative constants. The area saving compared to the first implementation is roughly $9M/7N$, the less the number of 1s there are in the multiplier constant $C$, the more area saving you achieve using this customized cell implementation. Like the register sharing in the case of the bit-serial multiplier, scalers also allowed to share registers up to two.

### 3.2.6 Rounding

When it is necessary to convert a double precision variable into a single precision variable or an extended-double precision into an extended single precision variable, bit-serial rounding operator is used. Let $X = (x_{2N-1}.x_{2N-2} \cdots x_1 x_0)$ be the double precision input and $X' = (x_{2N-1}.x_{2N-2} \cdots x_{N+1} x_N)$ be the truncated upper half of the input word. There are several methods of rounding:

1. **Truncate**
   Simply ignore the *low bit*. No logic is required.




(a) Round–to–nearest–integer (b) Round–to–nearest–even

Figure 3.15: Bit-serial rounding operators. Latency : $\delta_{RNI} = \delta_{RNE} = 1$.

2. **Round-to-nearest-integer**
   If $x_{N-1}$ is 1, increment $X'$. This is implemented by modifying the bit-serial adder (Fig.3.15(a)) with 2 CLBs. The latency $\delta_{RNI}$ is 1.

3. **Round-to-nearest-even**
   This is the standard floating-point rounding scheme [76]. This attempts to avoid the biased rounding scheme of the above at the halfway point between the integers.

   - If the fraction part is exactly 0.1, or 0.5 in decimal form, $X'$ is rounded up if the integer part is an odd number, otherwise it is truncated.
   - Otherwise round to the nearest integer.

   This is implemented by three signals : *sticky bit*, *guard bit*, and *round bit*.

$$\begin{array}{lcl} sticky & = & x_{N-2} + x_{N-3} + \cdots x_1 + x_0 \\ guard & = & x_{N-1} \\ round & = & x_N \end{array} \tag{3.9}$$

   With these three signals, the default carry input to the incrementer is determined as

$$inc = guard \cdot (sticky + round) \tag{3.10}$$

   This function is implemented with 3 CLBs. The latency $\delta_{RNE}$ is 1.

Rounding extended-double precision variable into extended-single precision variable is implemented in the similar way where two incrementers are cascaded to increment both the *high* word and *overflow* word.

### 3.2.7 Saturation

Conversion of extended-single precision variable $X = (x_{2N-1} \cdots x_{N-1}.x_{N-2} \cdots x_0)$ into single precision variable $Y = (y_{N-1}.y_{N-2} \cdots y_0)$ can be done by the methods described below. Conversion of extended-double precision into double precision can be done in the similar way.

- **Conversion by shifting**
  Shift $X$ to the right for some number of bits such that the output is guaranteed to be within the range $[-1, 1 - 2^{1-N}]$.

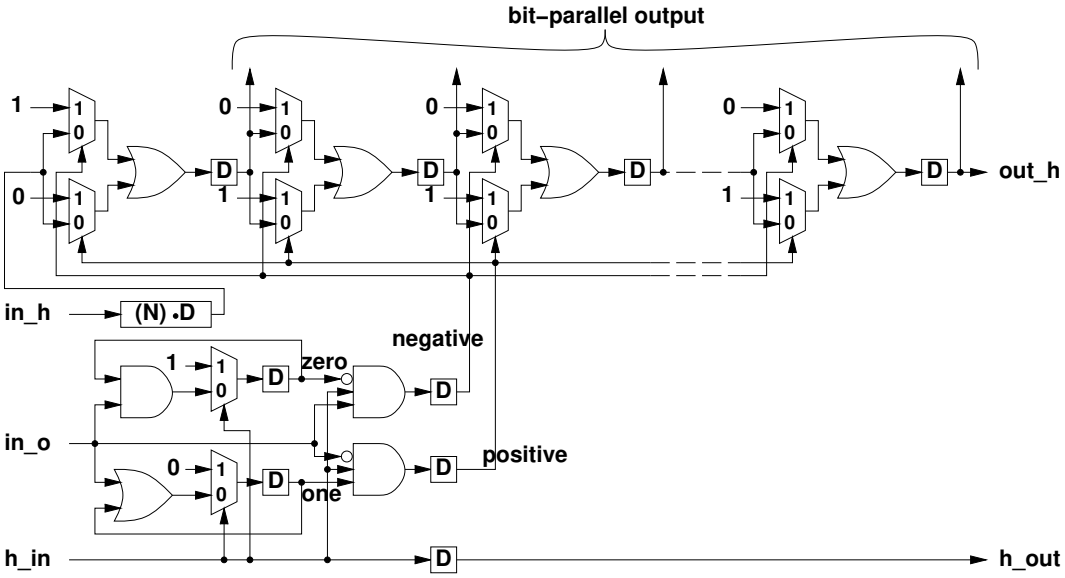



Figure 3.16: Bit-serial saturation operator. Latency : $\delta_{sat} = 2N + 1$

- **Conversion by saturation**
  Saturate the fraction part if $X$ is outside the range $[-1, 1 - 2^{1-N}]$.

  1. If $X \geq 1$, then $Y = 1 - 2^{1-N}$,
  2. else if $X < -1$, then $Y = -1$,
  3. else $Y \Leftarrow X$.

  Fig.3.16 shows the saturation circuit. It consumes $2N + 3$ CLBs and the latency $\delta_{sat}$ is $2N + 1$ cycles. This circuit can also produce bit-parallel data, therefore able to replace the serial-parallel converter as the output interface.

Saturation is a very expensive operation on LSBF bit-serial data on the hardware size and on latency where both grows linearly to the word size $N$. It is therefore recommended to use the first method whenever possible.

## 3.3 Circuit Complexity of Bit-Serial Operators

Table 3.1 shows the overall statistics of the various bit-serial operators that were designed. When a CLB is used only as latching one signal, it is counted as a half CLB consumption. Bit-serial subtracter have the identical numbers as the adders. As I have stated earlier, all of these operators are implemented in one-level LUT logic, therefore the critical path is maximally balanced. And all operators have the same throughput $f/N$ where $f$ is the operational clock frequency. These circuits can be roughly categorize into two groups.

1. Operators whose hardware size are independent of the data word size $N$, namely add, subtract, shift-right, shift-left, rounding. Their AT (area·time) factor is $O(N)$. With exception of comparator, latency of most operators in this category is independent of $N$ as well.

| | input | output | size (CLBs) | latency (cycles) |
|---|---|---|---|---|
| $K$-input parallel-serial | parallel | SP | $K(N+1)/2$ | 1 |
| $K$-output serial-parallel | SP | parallel | $K(N+K)/2$ | $N+1$ |
| Add(I) | SP | SP [ESP] | 2 [+1] | 1 |
| Add(II) | DP | DP [EDP] | 3 [+1] | 1 |
| $k$-bit shift-right(I) | SP | SP | $\lceil 0.5k+1.5 \rceil$ | $k+1$ |
| $k$-bit shift-right(II) | SP, DP | DP | $\lceil 0.5k+2 \rceil$ | $k+1$ |
| $k$-bit shift-left(I) | SP | SP [ESP] | $k+1$ [+1] | 1 |
| $k$-bit shift-left(II) | DP | DP [EDP] | $\lceil 1.5(k+1) \rceil$ [+0.5] | 1 |
| Multiplication | SP | DP [EDP] | $5N-4$ [+1] | $2N-1$ |
| Scaling (unified) | SP | DP | $4N+2$ | $2N+1$ |
| Round-even(I) | DP | SP | 3 | 1 |
| Saturation(I) | ESP | SP | $2N+2$ | $2N+1$ |

Table 3.1: Circuit area and operation latency of the various bit-serial operators. Xilinx 3000 architecture is assumed for CLB counts. (SP = single precision, DP = double precision, ESP = extended-single precision, EDP = double precision, $N$ = data word size)

2. Second group consists of operators whose hardware size grow linearly with $N$. Their AT factor is $O(N^2)$. With exception of parallel-serial converter, latency of most operators in this category grows linearly with $N$ as well.

# Chapter 4

# C++ Bit-Serial Datapath Synthesis System

The C++ hardware design capture method developed in this thesis has the following features:

1. The C++ code is compiled by the standard GNU C++ compiler (`g++`), therefore the code can include any type of expressions supported by C++. Existing hardware design tools using various high-level programming languages all have dedicated compilers where a great deal of effort is needed in developing and maintaining these tools [33], [31], [87], [1], [94], [42], [53].

2. The C++ code is both a hardware simulator and a synthesizer. Software codes can reside within the hardware description codes for simulation purposes such as reading in test vectors or even creating one, adding break points, displaying the simulation results, or emulating the whole application entirely. This enables the designer to perform a thorough and easy design verification. This feature is also unique compared to other tools.

3. Distinction between hardware operators, computations which are to be implemented on the configurable hardware, and software operators is made by the C++ compiler which examines the data types of the argument variables. Expressions with predefined *hardware class objects* are automatically translated into various hardware object functions which construct the hardware description.

4. By utilizing the operator overloading feature of C++, the C++ coding style for hardware description becomes very similar to ordinary C++ coding.

5. The computational algorithm is described in difference equations. This form of design specification is widely used in many of the High-Level Synthesis systems for applications of digital signal processing such as SILAGE [91]. There are some restrictions imposed to the difference equation expression such as equation ordering.

## 4.1 Basic Concept of the Function-Based Hardware Design Capture

The largest distinction between the existing hardware design capture system using high-level programming languages and the approach taken here is that any kind of customized compiler is not used to extract the hardware descriptions. Instead, the *extensibility* feature of C++ is utilized, which is the operator overloading [77]. The hardware designer, which in this case, a *hardware programmer*, utilizes two types of data objects: *hardware objects* and *software objects*. Software objects are the ordinary data types such as integer, floating point, character, etc.,

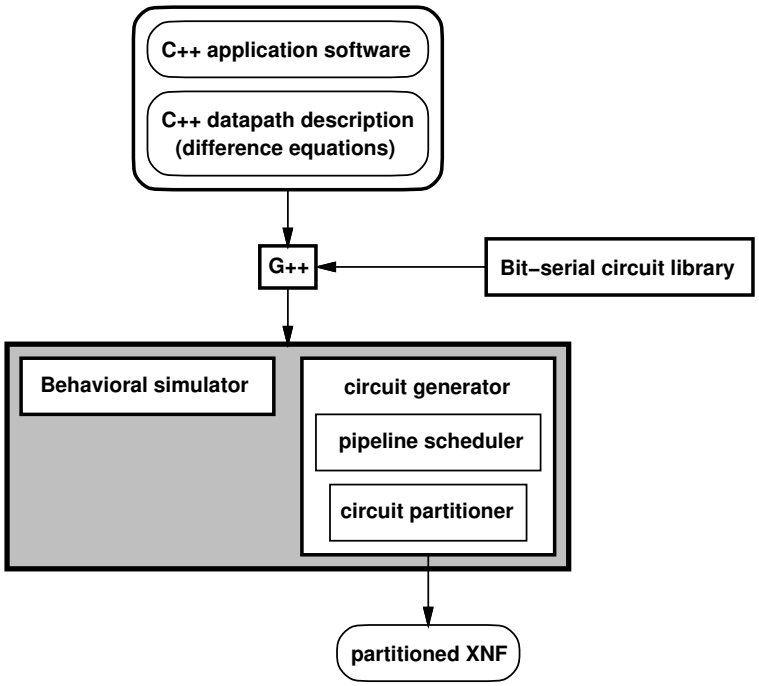



Figure 4.1: C++ capture system compilation flow developed in this thesis.

and any other user-defined data objects. Hardware objects are defined by the separate include files and object codes developed by the author which is used to describe *signals* which connect the hardware modules which are to be mapped onto the configurable hardware. A set of overloaded operators are provided for these hardware objects to let the programmer declare *hardware operators* in the same way he or she would express operations with ordinary data types. The hardware descriptions are automatically extracted by the C++ compiler and appropriate function calls are assigned to each overloaded functions. The executable file becomes the circuit generator and also the behavioral simulator. The compilation flow is illustrated in Fig.4.1. This compilation flow is quite similar to the `wirec` circuit design capture system [71] where they generate C++ codes from their schematic capture and the executable file compiled from these codes generates a circuit netlist which can be used by the logic simulator.

## 4.2 Datapath Description Model

As the datapath design description model, *difference equations* is employed here which often used to specify time-discrete systems. This form of design specification is widely used in many of the high-level synthesis tools as the design capture scheme for applications in digital signal processing such as SILAGE [91]. The computation is modeled such that each *signal* has an implicit time index which increments when the whole computation sequence (although does not mean that the computations occur *in sequence*) specified by the difference equations is executed. This computation sequence is assumed to iterate indefinitely. This indefinite process iteration gives us the opportunity of increasing the throughput by means of functional pipelining even if there are is a small degree of parallelism in the computation threads. Since bit-serial circuits utilize high-speed bit-level pipelining, it is only natural for us to extend the pipelining to the functional-level.

The computations can have if-else branches and for-loops for describing the control-flow as well. By allowing

if-else control-flow statements, also for-loops are allowed to express straightforward state machines for generating multidimensional memory addresses. Furthermore, although this is not yet supported, it is possible to implement complex state machines by these control-flow statements. IO interface are supported to make communication between the hardware datapath and the software, and also between the datapath and memories. In this implementation, all branches will be mapped onto the hardware and therefore exhaustively executed. This approach may be inefficient for implementing computations with large amount of complex control-flow. However, for many of the DSP applications tend to have very simplistic control-flow structures and therefore the overhead for these exhaustive branch execution can be expected to be small.

The task for the designer is to come up with a set of difference equations which correctly reflects the algorithm and which produces high throughput (sometimes low latency) pipeline structure with minimum hardware size. There are a number of factors which determines the throughput of the datapath and its size, such as the data word size, number of operations, types of operations, structure of computation (recursive or non-recursive). The designer will try out many algorithms (sets of difference equations) trading off the quality of the computation outputs, speed and size. These different algorithms are first verified, corrected if necessary, and then sent on to the hardware synthesis for speed and size evaluation.

## 4.3 Components of C++ Datapath Description

There are four major components used to describe the behavior of the computations on the hardware:

1. *Signal variable*
   Hardware signal variables are the means to calculate intermediate and the final results of the computations. The following statement sets the size of the data word size:

   - `void set_signal_precision(int N);` : Sets single precision data word size to N.

   There are four types of signal variables expressed in 2's complement form: *Single, Double, Extended Single* and *Extended Double*.

   - `class Single` : Single precision word ($N$ bit data).
   - `class Double` : Double precision word ($2N$ bit data).
   - `class Ext_Single` : Extended Single precision word ($N$ bit overflow extension, $N$ bit data).
   - `class Ext_Double` : Extended Double precision word ($N$ bit overflow extention, $2N$ bit data).

2. *Control variable*

   - `class Control` : Control variables are means to express sequential processes on the datapath. Current implementation of a state machine is limited to cascaded counters aimed to implement for-loops, memory address generation and conditional signal multiplexing.

3. *IO interface and data storage*
   The datapath circuit on the FPGA hardware have access to three devices: *system bus* which provides connection to the host CPU, *RAM* which is used to store intermediate results whose address is generated by control variables, and *ROM* which is used to implement a large lookup table (LUT) on external memory device whose address is generated by a signal variable.

- `class Interface` : This provides connection to the system bus. Parallel-serial converter enables the communication between bit-parallel system bus and bit-serial FPGA datapath. User can specify multiple system buses, although the current tool does not have the necessary features to specify which bus is mapped to which peripheral devices. This should be supported in the future versions.
- `class Ram` : This provides connection to the bit-parallel SRAMs in which the datapath can either read or write every clock cycle whose address is generated by control variables. Data are stored in bit-parallel format at each address, and therefore a parallel-serial interface is provided for the interface.
- `class Rom` : This provides connection to the bit-parallel SRAMs in which the datapath can read every clock cycle whose address is generated by a single precision signal variable. Data are stored in bit-parallel format at each address, and therefore a parallel-serial interface is provided for the interface.

4. *Signal operator declaration*
There are two ways to declare hardware operations which are *declaration by overloaded arithmetic operators* and *declaration by dedicated function calls*. There are several overloaded arithmetic operators currently supported:

   - Addition (`+`): Takes in all four signal types and produces all four signal type accordingly.
   - Subtraction (`-`): Takes in all four signal types and produces all four signal type accordingly.
   - Multiplication (`*`): Takes two single precision signal inputs or one single precision signal input and a constant floating point input (this is also called a *scaler* since it scales the input signal by a constant factor), and produces a double precision signal output. For the scaler declaration, there are two types of hardware implementation, one with a redefinable unified cell structure and the other with a fixed structure as described in Section 3.2.5. If the scaling operation is with a `float` variable or a `double` variable, the fixed structure is chosen. If the designer wishes to implement the scaler with a redefinable structure so that the scaling constant can be redefined using the same circuit layout, a dedicated class object `class Wildcard` is provided to replace the `float` or `double` variable. By using this `Wildcard` variables in the expression, the tool will chose the redefinable structure implementation.
   - Assignment (`=`): If the right-hand side of the equal sign is a constant, this represents a value assignment to the variable on the left-hand side. Currently, this value is only allowed to be 0, therefore a reset operation. Non-zero assignments should be supported in the future. If the right-hand side is an expression using signal variables, this simply expresses an assignment to the left signal variable. left.

   Also, there are a number of dedicated function calls designated for declaring hardware operations:

   - `delay(*)` : Takes in all four signal types and delays it by one sampling period. This is the basic element for implementing digital filters.
   - `line_delay(Single &)` : Takes in a single precision signal and delays it by $L$ sampling periods. $L$ is explicitly defined by the designer by the statement `LINE_SIZE = L` previous to the difference equations. This is used to implement line buffers for two-dimensional image processing using an external RAM.
   - `round(Double &)` : Takes in a double precision signal and converts into a single precision signal by round-to-nearest-even scheme.
   - `truncate(Double &)` : Takes in a double precision signal and converts into a single precision signal by truncation.
   - `saturate(Ext_Single &)` : Takes in a extended-single precision signal and converts into a single precision signal by saturation.

- `saturate(Single &)` : The data word size $N$ is declared by the statement `set_signal_precision(int N)`. For acyclic computations where there are no cycles in the data dependency graph, the sampling period $T$ automatically becomes $T = N$. However, for cyclic computations, the sampling period $T$ may need to be longer than $N$ because of the long critical path along the cycles. Therefore, the effective word size becomes $T$ ($T \geq N$), and the extra $T - N$ bits are added to the left of the MSB. This saturation function takes in a $T$-bit single precision signal $x$ and performs the saturation described below.

$$y = \begin{cases} 2^{N-T} - 2^{1-T} & (x \geq 2^{N-T} - 2^{1-T}) \\ x & (-2^{N-T} \leq x < 2^{N-T} - 2^{1-T}) \\ -2^{N-T} & (x < -2^{N-T}) \end{cases} \tag{4.1}$$

  The $T - N$ most significant bits are sign-extended.

5. *Control variable operator declarations*
   Declaration of the control variable operators are done using the overloaded arithmetic operators.

   - Arithmetic operations (`+`, `-`, `*`) : The integer values recorded in control variables are used as indices for data arrays on both the hardware and the software. These operations are not implemented on the hardware. The return values of these operations are therefore integers.
   - Logic expressions (`<`, `==`) : These two logic expressions are applied to control variables in order to generate the *select* signals for the multiplexers implementing the branches, or the *reset* signals to reset the counter. Other expressions are not supported in the current tool. The return values for these expressions are integers (0 or 1).
   - Increment (`++`) : This expression is used inside the for-loop with the control variable.

## 4.4 Difference Equation Description Using Overloaded Arithmetic Functions

Difference equation can be described using a set of operations provided in the module library. Following code is an simple example of a signal convolution shown in Fig.4.2.

```
Interface bus;
Single x[8], y;
Double a[8];
extern double coef[8];
// Predefined filter coefficient floating point data
set_signal_precision(16); // Setting the signal word length to 16 bits

for(int j = 0; j < 8; j ++){
    if(j == 0){
        x[j] = bus.read();
        a[j] = x[j] * coef[j];
    }
    else {
        x[j] = delay(x[j - 1]);
        a[j] = x[j] * coef[j] + a[j - 1];
    }
}
y = round(a[7]);
bus.write() = y;
```

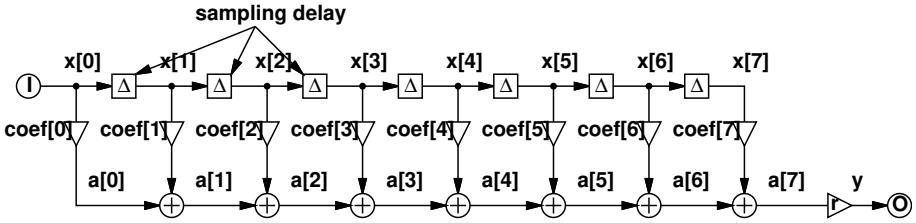



Figure 4.2: FIR filter

- `Interface bus;` : Declares an IO interface between the host CPU.
- `Single x[8], y; Double a[8];` : Declare signal variables. Since these signal variables are merely data classes, they can be declared as vectors using array statement.
- `x[j] = bus.read();` : Declares a data input module from the interface bus to the datapath hardware. The input data is stored in the single precision signal `x[0]`.
- `a[j] = x[j] * coef[j];` : Declares a scaler whose scaling factor is `coef[j]` (must be a fractional value), whose input is a single precision signal `x[j]` and whose output is a double precision signal `a[j]`.
- `x[j] = delay(x[j - 1]);` : Declares a sample delay whose input is `x[j]` and whose output is `x[j - 1]` (variable precision of the input and the output must match). Using the *Z-transform*, their relationship is $x[j] = z^{-1} \cdot x[j-1]$.
- `y = round(a[7]);` : Declares a rounder (round to nearest even integer) whose input is double precision signal `a[7]` and whose output is single precision signal `y`.
- `bus.write() = y;` : Declares a data output module to the interface bus from the datapath hardware.

Note that here, if-else statements and for-loop statements does not describe sequential processes on the hardware since the arguments used in the expressions are not control variables, but merely to describe the datapath structure efficiently. One of the advantage in this function-based design capture method we can see here is that the conventional C++ expressions are still allowed within the hardware description code segments which greatly facilitates the coding effort.

## 4.5 Behavioral Design Verification

While the code segment shown above declares the hardware modules which implement the computations, the very same code serves as a mean to verify the behavior of the datapath.

```
void 1D_FIR_filter(int in[MAX], int out[MAX], double a[], int m)
{
    Interface bus;
    Single x[m + 1], y;
    Double w[m + 1];
    set_signal_precision(16);

    for(int t = 0; t < MAX; t ++){
        for(int k = 0; k <= m; k ++){
            if(k == 0){
                x[k] = bus.read(in[t]);
                w[k] = x[k] * a[k];
            }
            else {
                x[k] = delay(x[k - 1]);
                w[k] = x[k] * a[k] + w[k - 1];
            }
        }
        y = round(w[m]);
        bus.write(out[i]) = y;
        clocker();
    }
}
```

- `for(int i = 0; i < TMAX; i ++){` : This is the application software part of the code where `i` provides the index for the input and output data arrays which reside on the host computer.
- `clocker();` : Indicates that the whole computation sequence has been executed and the sample clock is incremented.

This code segment actually calculate the convolution value of the input signal data `input[i]` and stores in `output[i]` while actually generating each hardware module. As we can see here, verifying the functionality of the datapath is straightforward and designer has the full advantage of C++ features to aid the generation of the test vectors and processing of the simulated output data.

## 4.6 Sequential Process Description

As we have described earlier, a simple form of sequential process can be described using *control* variables. Following is an example of a sequential process expressed by a nested for-loop (Fig.4.3).

```
Single x[8];
Interface bus;
Ram ram;
Control s0, s1, s2;
extern int in[];

for(s0 = 0; s0 < 2; s0 ++){ // width of s0 is 1 bit.
    for(s1 = 0; s1 < 8; s1 ++){ // width of s1 is 3 bits.
        for(s2 = 0; s2 < 8; s2 ++){ // width of s2 is 3 bits.
            if(s0 == 0){
                x[s2 + 0] = bus.read(in[s1 * 8 + s0]);
                // s2 + 0 is a dummy expression which is needed to
                // return the value of s2
            }
            else{
                x[s2 + 0] = ram.read(s0, s1, s2);
                // The address of the RAM read operations are
                // generated by concatenation of the three control
                // variables s0, s1 and s2.
            }
        }
        .
        .
        .
        clocker();
    }
}
```

- `Ram ram;` : Declares an IO interface between the local SRAM.
- `Control s0, s1, s2;` : Declares three control variables.
- `for(s0 = 0; s0 < 2; s0 ++){` : Declares a counter with a maximum value of 2. `s0` is implicitly assigned 1 bit as the word size. This *for*-loop is emulated as a normal for-loop. In other words, the code inside the bracket is repeated twice. Let us call this `s0`-loop.
- `for(s1 = 0; s1 < 8; s1 ++){` : Declares a counter with a maximum value of 8. `s1` is implicitly assigned 3 bits as the word size. The code inside the bracket is repeared 8 times. Let us call this `s1`-loop.
- `for(s2 = 0; s2 < 8; s2 ++){` : Declares a counter with a maximum value of 8. `s2` is implicitly assigned 3 bits as the word size. The code inside the bracket is repeared 8 times. Let us call this `s2`-loop.
- `if(s0 == 0){, else{` : Declares a conditional signal multiplexer. This *if* statement is emulated as a normal *if* statement. In other words, the code inside the bracket is executed if the expression `s0 == 0` is true or otherwise ignored.
- `x[s2 + 0] = bus.read(in[s1 * 8 + s0]);` : Data array input from the system bus. When the control variables are used as array indices, it is treated as integers. Technically speaking, the overloaded functions `+, -, *` with control variable operands returns an integer value which is to be used as the array index. `s2 + 0` is a dummy expression which returns the value of s2.
- `x[s2 + 0] = ram.read(s0, s1, s2);` : Data array input from the SRAM whose address is calculated by the three control variables. The address has 7 bits: $[s0_0, s1_2, s1_1, s1_0, s2_2, s2_1, s2_0]$. As discussed earlier, this statement is only executed when the expression `(s0 == 0)` is false.

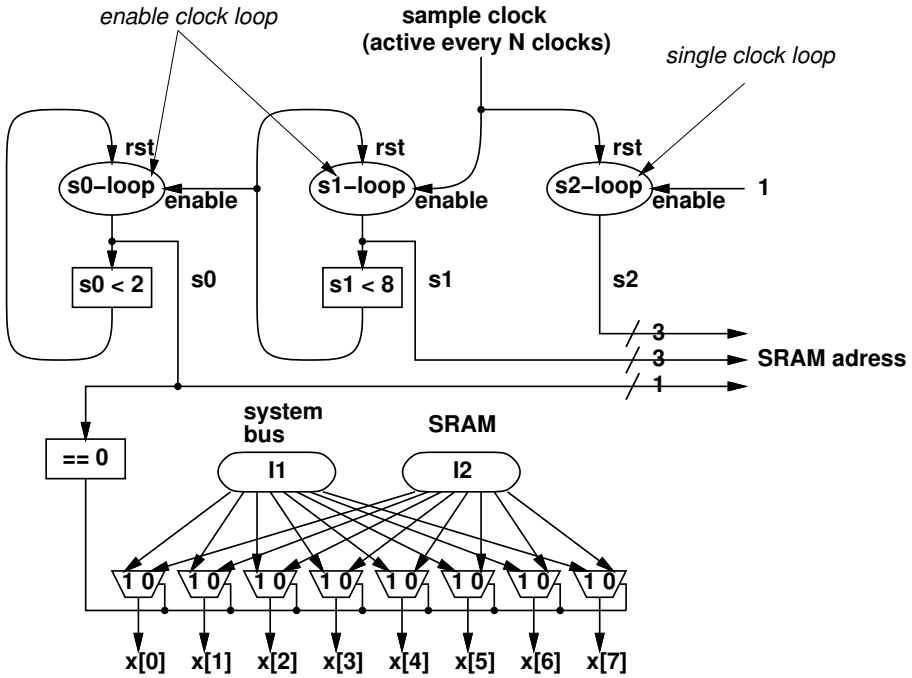



Figure 4.3: Example of a sequential process on the datapath. `s1`-loop is nested inside `s0`-loop. When `s1`-loop reaches the maximum value, it will reset itself and increments `s0`-loop. `s0`-loop and `s1`-loop called *enable clock loops* where the counter is incremented when the enable input is active. `s2`-loop is a `single clock loop` where the counter is incremented every cycle and is reset every $N$ cycles. 8 input signals from the system bus and 8 input signals from the SRAM are multiplexed by the expression `s0 == 0`.

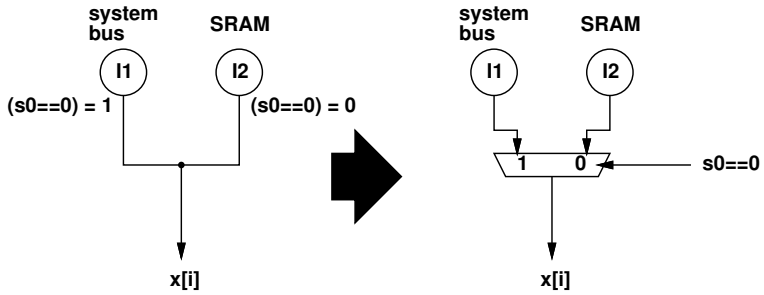



Figure 4.4: A Signal with multiple sources has to be transformed into multiple single-sourced signals feeding into a multiplexer.

When the clocking statement (`clocker()`) is outside the *for*-loop as see in the inner-most `s2`-loop, the loop is called *single clock loop* where the loop counter is incremented every clock cycle and is reset every $N$ clock cycles where $N$ is the data sample period. When the clocking statement is inside the *for*-loop as seen in the outer loops (`s1`-loop, `s0`-loop), the loop is called *enable clock loop* where the loop counter is incremented each time the inner loop is reset, as `s0`-loop whose inner loop is `s1`-loop, or is incremented every $N$ clock cycles if there are no inner *enable clock loops* as `s1`-loop (Fig.4.3).

The tool keeps track of logic expressions of control variables (currently, *equality* is the only available logic expression) and their values (true or false) for each signal. This is needed since the tool cannot recognize whether or not *if-else* statement has been executed. It only knows that the overloaded logic expression is executed. For each output pin of each module, such as those input modules, an active logic expression and its value is recorded. Since the tool does not recognize that if-else branch exists, the above code will initially generate a circuit where the signals can have multiple sources (Fig.4.4). Those signals are then transformed into multiple single-sourced signals feeding into multiplexers.

The high-speed circuit implementation of the counter which implements the for-loops is given in Section 4.10.

## 4.7 Hardware Module Generator

Here we will look into the details of hardware operator declarations. As we have seen, declaring a hardware operator invokes two events in the program:

1. Emulates the actual arithmetic operation result for behavioral simulation.

2. Generates a new operator module if no redundancy is detected. For instance, consider the code shown in Fig.4.5. The first line will generate a scaler whose scaling factor is 0.3 and whose input is `a0`, a scaler whose scaling factor is 0.4 and whose input is `a1`, and an adder whose inputs are the outputs of the two scalers and whose output is `f0`. However at the second line, only a subtracter is newly generated since the two scalers have already been generated (Fig.4.5).

Table 4.1 shows the currently available module generators.

- Functions whose names start with `operator`$\cdots$ are overloaded functions in which they can be called by merely typing '+', '-', or '*'.

- Module generator `Double & operator*(Single &, Wildcard &);` is a function which generates a scaler with a unified cell structure (refer to Chapter 3).

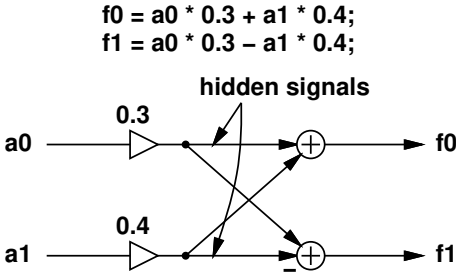



Figure 4.5: Instead of generating four scaler modules, the module generator recognizes the redundancy and generates only two scaler modules.

- Module generator `Single & line_delay(Single &);` is a function which generates a line buffer for image processing. The length of a line, or the number of pixels in the horizontal direction has to be specified by assigning a value to the external variable `LINE_SIZE`. This line delay is implemented on off-chip SRAM, and the necessary address generator is synthesized automatically.
- Module generator `Single & saturate(Ext_Single &);` is a function which generates a saturation circuit for converting an extended single precision signal to a single precision signal.
- When the data word size $N$ is shorter than the sample period $P$, this can occur with cyclic datapath circuits where the critical loop length determines the sampling period, and when the signal have to be bounded within $N$ bits, module generator `Single & saturate(Single &);` can be used which generates a saturation circuit for converting $P$-bit data into $N$-bit data.

## 4.8 Bit-Serial Circuit Library

### 4.8.1 Hierarchical Data Structure of Bit-Serial Pipeline Network

In this section, some terminologies about the components are introduced which constitutes a bit-serial pipeline network.

### 4.8.2 Directed Hyper-Graph Representation

Any type of digital circuit can be represented as a directed hyper-graph $HG(\overline{E}, V)$ consisting of hyper-edges ($\overline{e} \in \overline{E}$) and vertices ($v \in V$). A hyper-edge $\overline{e} = (\{u_0, \cdots, u_{j-1}\}, \{v_0, \cdots, v_{k-1}\})$ can have multiple sources $\{u_0, \cdots, u_{j-1}\}$ and multiple destinations $\{v_0, \cdots, v_{k-1}\}$, although multiple-source hyper-edges are created only when there is an if-else branch and a signal is calculated with a multiple expressions, in which one one of them is selected at a time. These hyper-edges are immediately converted into single-source hyper-edges by inserting multiplexors as shown in Fig.4.4. The word *netlist* will be used to refer to this directed hyper-graph representation.

There are three levels of netlists which represents the bit-serial pipeline network: *module-level netlist*, *cell-level netlist* and *logic-level netlist* (Fig.4.8).

| |
|---|
| Single & operator+(Single &, Single &); |
| Double & operator+(Double &, Double &); |
| Double & operator+(Single &, Double &); |
| Double & operator+(Double &, Single &); |
| Ext_Single & operator+(Ext_Single &, Ext_Single &); |
| Ext_Double & operator+(Ext_Double &, Ext_Double &); |
| Ext_Double & operator+(Ext_Single &, Ext_Double &); |
| Ext_Double & operator+(Ext_Double &, Ext_Single &); |
| Ext_Single & operator+(Ext_Single &, Single &); |
| Ext_Double & operator+(Ext_Double &, Double &); |
| Ext_Double & operator+(Ext_Single &, Double &); |
| Ext_Double & operator+(Ext_Double &, Single &); |
| Ext_Single & operator+(Single &, Ext_Single &); |
| Ext_Double & operator+(Double &, Ext_Double &); |
| Ext_Double & operator+(Single &, Ext_Double &); |
| Ext_Double & operator+(Double &, Ext_Single &); |
| |
| *Functions with the same argument types and same return value types exist for* operator-$(\cdots)$ |
| |
| Double & operator*(Single &, Single &); |
| Double & operator*(Single &, double); |
| Double & operator*(double, Single &); |
| Double & operator*(Single &, float); |
| Double & operator*(float, Single &); |
| Single & delay(Single &); |
| Single & line_delay(Single &); |
| Double & delay(Double &); |
| Single & round(Double &); |
| Single & truncate(Double &); |
| Single & saturate(Ext_Single &); |
| Single & saturate(Single &); |

Table 4.1: Hardware module declaration.

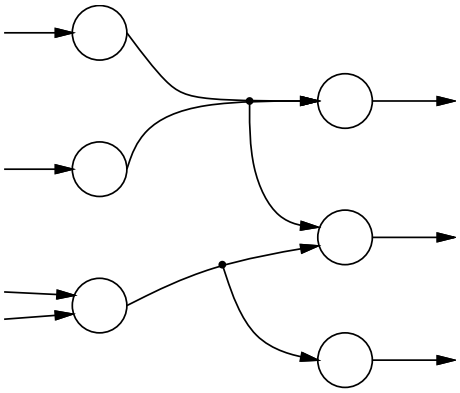


Figure 4.6: A directed hyper-graph. Multiply-sourced edges are allowed in the representation.

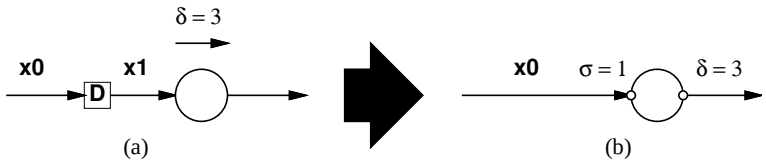



Figure 4.7: Module-level netlist representation of a sampling delay. Instead of representing each sampling delay operator with a node, the amount of sampling delays ($\sigma$) to each module input is recorded in the input pin. Each output pin has a latency value ($\delta$) attached which denote the processing latency of that module.

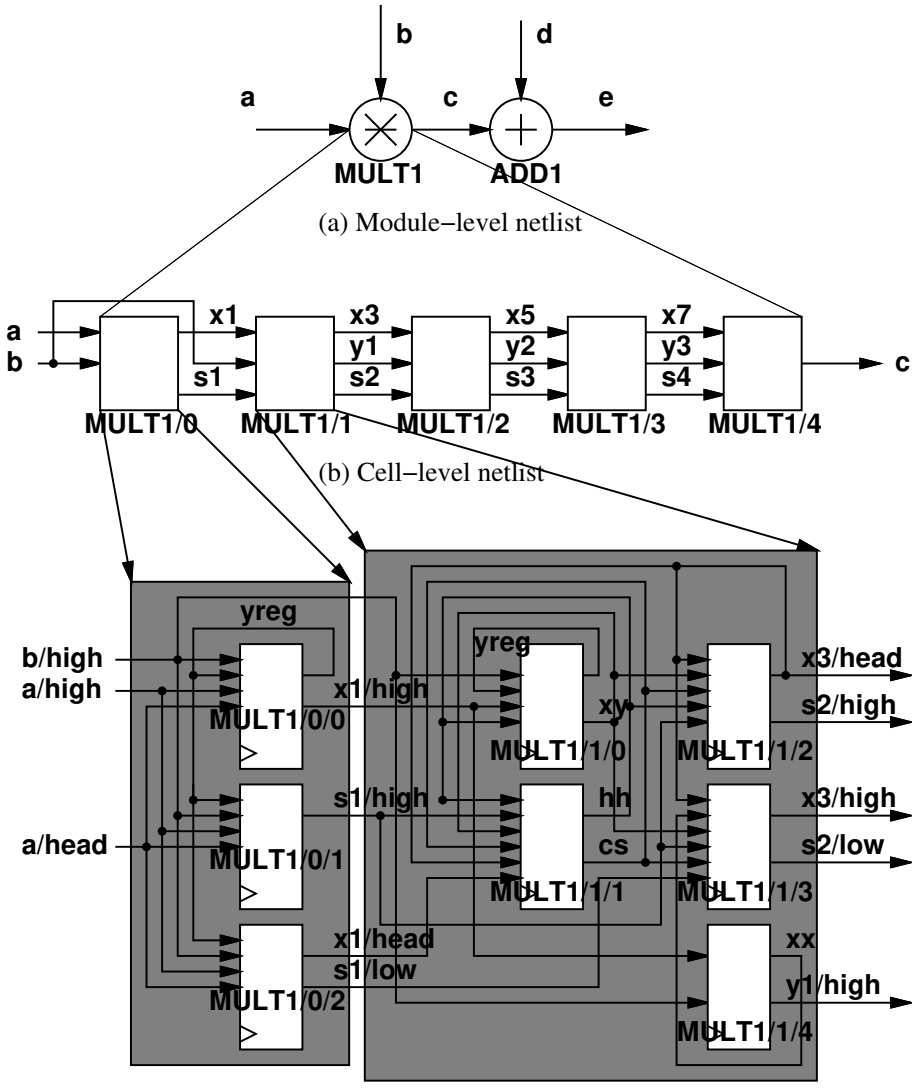



Figure 4.8: Three levels of datapath netlist.

### 4.8.3 Module-level netlist $HG(\overline{E_m}, V_m)$

This representation is the highest level in the hierarchy of the bit-serial pipeline network representation whose vertices denote bit-serial *modules* such as adder, subtracter, multiplier, rounder, etc., and whose edges denote *signals*.

- `class Module` is an abstract object class which represents a vertex in the module-level netlist. This class has a number of *custom module* child classes which represent different types of operators. Pipeline synchronization, bit-serial control signal routing and pipeline register minimization is done on this level. Also, input and output pins of the modules each holds some important timing information:
  - Input pins which connect to input signals have an integer sampling delay values. For instance, if a scaler module has an input $x_1$ which is a delayed sample of $x_0$, i.e.,

    $$x_1 = delay(x_0), \tag{4.2}$$

    then instead of representing the sampling delay operator with a node in the hyper-graph, the scaler module input is connected to $x_0$ with a sampling delay value of 1 (Fig.4.7).
  - Output pins which connect to output signals have an integer latency value, the same one defined in chapter 5, which represents the processing latency of the module.

  These input sampling delay values and output latency values are used later on to determined the weight of an edge which determines the scheduling of the pipeline.
- `class Signal` is a class which represents a hyper-edge in the module-level netlist. It has four types (Single, Double, Extended-Single, Extended-Double) reflecting the signal precision.

### 4.8.4 Cell-level netlist $HG(\overline{E_c}, V_c)$

This representation is the next level in the hierarchy. Each *module* is subdivided into *cells* connected by *buses*. Cells are bit-serial operators which constitute the bit-serial modules. Most of the bit-serial modules such as adder, subtracter, and rounder consist of only a single cell. Also, the input parallel-serial converter and the output serial-parallel converter are regarded as one cell, respectively. Multipliers and scalers consist of multiple cells. Circuit partitioning is done on this level.

- `class Cell` is an abstract class which represents a vertex in the cell-level netlist. This class has a number of *custom cell* child classes which represent different type of cells in each *custom module*.
- `class Bus` is a class which represents a hyper-edge in the cell-level netlist.

### 4.8.5 Logic-level netlist $HG(\overline{E_l}, V_l)$

This is the lowest level in the hierarchy which is composed of *logic elements* connected by *nets*. Logic element represents either a configurable logic block (CLB), an IO block (IOB), a tristate buffer (TBUF) or a pullup resistor (PULLUP) where are the programmable logic elements on Xilinx XC3000 FPGA family. The circuits are finally synthesized at this level.

- `class Logic` is an abstract class which represents a vertex in the logic-level netlist. This class has four *custom logic* child classes (`class Clb, class Iob, class Tbuf, class Pullup`) which represent different types of logic elements on the Xilinx XC3000 FPGAs.
- `class Net` is a class which represents a hyper-edge in the logic-level netlist.

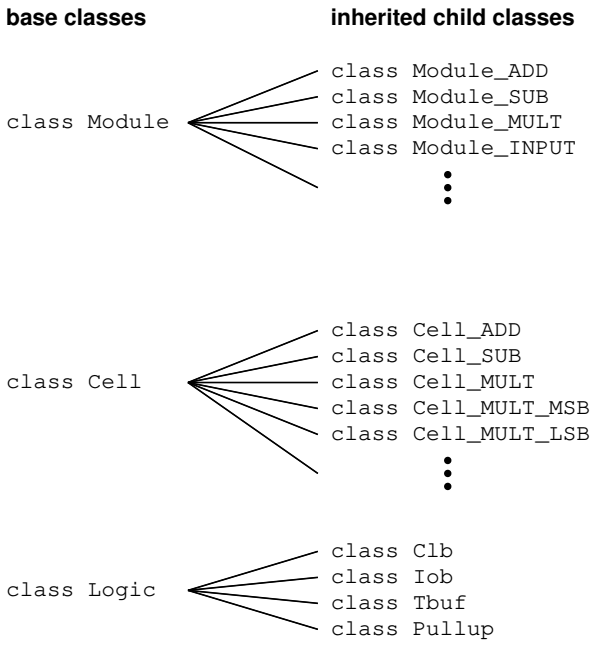



Figure 4.9: Hardware module classes. A variety of hardware module objects at the module-level and cell-level are inherited from their base object class.

### 4.8.6 Netlist Hierarchy and Circuit Element Labeling

The netlists are generated from top to bottom. In order to provide a means to traverse the netlist vertically between hierarchies, the following data structure is provided:

1. *Vertex hierarchy*
   - `Module` has a linked list of `Cell`. Each `Cell` has an attribute to `Module`.
   - `Cell` has a linked list of `Logic`. Each `Logic` has an attribute to `Cell`.
2. *Hyper-edge hierarchy*
   - `Signal` has a pointer to `Bus`. Each `Bus` has an attribute to `Signal` which may be *null*.
   - `Bus` has a list of `Net`. Each `Net` has an attribute to `Bus` which may be *null*. If the bus attribute of a net is not null, it means that the net represents either the head bit, high bit, low bit or the overflow bit.

Circuit element labeling is done using this netlist hierarchy.

1. *Vertex labeling*
   - Each custom module is labeled with a type name followed by a unique index within the same module type(`ADD1`, `MULT1`).
   - Each custom cell is labeled with the label of its attributed module followed by a unique index within the same module (`ADD1/0`, `MULT1/0`).

- Each logic element is labeled with the label of its attributed cell followed by a unique index within the same cell (`ADD1/0/0, ADD1/0/1`). As an option, a logic element can have a local name concatenated to the label (`MULT1/0/0/carry`).

2. *Edge labeling*

   - Each signal is initially labeled with a string "SIG" followed by a unique number (`SIG0, SIG1`). After declaring each signal (as shown in the previous section), the designer can choose to label each signal to make the netlist more comprehensible as in the following.

```
Single x, y, z;
x.name("x"); // Labels signal x as "x"
y.name("y"); // Labels signal y as "y"
z.name("z"); // Labels signal z as "z"
```

   - Each bus has its own label. However if its signal attribute is not null, it is labeled the same as its attributed signal label.
   - Each net has its own label. However if its signal attribute is not null, it is labeled with the attributed bus label followed by either "head", "high", "low" or "over" according to the type of the net.

### 4.8.7 Hierarchical Netlist Generator

The bit-serial module library is composed of a set of functions which generates the hierarchical netlist data structure.

1. Each hardware operator functions in Table 4.1 includes the following functionality.

   - Emulates the actual arithmetic operation result for behavioral simulation.
   - Checks if a module with identical type and identical set of input signals already exists. If there is no such module, the appropriate *custom module constructor* is called. A *custom module* is a child class inherited from class `Module` which represents one of the bit-serial modules having a unique constructor and a cell generator.

2. For each individual module, following functions are provided:

   - *Custom module constructor* : A function which initializes various data such as number of inputs, number of outputs, module type, module name, latency of the output pins, and most important of all, connecting input and output pins to the corresponding input and output signals.
   - *Cell generator* : A function which generates cells by generating the necessary buses which connects the cells and calling the various cell constructors.

3. For each individual cell, following functions are provided:

   - *Cell constructor* : A function which initializes various data such as number of inputs, number of outputs, cell type, cell name, and connecting input and output pins to the corresponding input and output buses.
   - *Circuit generator* : A function which generates logic elements such as CLBs and IOBs by generating the necessary nets which connects the logic elements, calling the proper logic element constructor, naming the logic element if desired (not required), connecting the input and output pins of the logic element to the corresponding nets, and providing information about the configuration of the logic element such as logic equations for the LUT. Currently, all library circuits are constructed for Xilinx 3000 series FPGA architecture.

```
_      inherited from Module
{

    _         module constructor
       _   generates module name
     _   generates cells
}

 --
{

    --
     -
}

 -
{

    -
     -
}

 --
{

    --
     -
}
```

Figure 4.10: Definitions of class `Module_MULT` and its cell classes (`Cell_MULT_LSB`, `Cell_MULT`, `Cell_MULT_MSB`).

Here, let us describe the details by showing an example. It should be note that the following example is a slightly modified version of the actual code containing some portions which are irrelevant for the purpose of the explanation or unnecessarily confusing, and therefore are omitted (also see Fig.4.9).

- *Custom module class* (`class Module_MULT`, Table 4.10): This is a child class of `Module`. It has its own constructor (`Module_MULT()`), name generator (`module_name()`) and its own cell generator (`generate_cell()`);
- *Custom cell class* (`class Cell_MULT_LSB, class Cell_MULT, class Cell_MULT_MSB`, Table 4.10): These are the three cell classes which construct one bit-serial multiplier. Each cell class, which is a child class of `Cell`, has its own constructor and a circuit generator.

```
--
  _    Calling base constructor.
 There are 2 inputs and 1 output. Type is "MULT"
{
       Connecting the input signal to input pin
       Connecting the input signal to input pin
       Connecting the output signal to output pin
            Setting the latency of the output pin
}
```

Figure 4.11: Constructor for `Module_MULT`.

```
 --
{

        _  x[0] is the bus of the first input signal
                  {

                  is a member function of a class  which returns a  pointer.
                x[i] is a newly created bus
     }

        _
        _  y[0] is the bus of the second input signal

              {

                sum[i] is a newly created bus

                y[i] is a newly created bus
     }
      __       LSB cell
                {
            _
          Middle cells
     }
      __                MSB cell
}
```

Figure 4.12: Cell generator of `Module_MULT`.

```
----
      Calling the base constructor.
 There are 2 inputs and 2 outputs. Type is "MMLSB".
{



}
```

Figure 4.13: Constructor for `Cell_MULT_LSB`.

```
---
{
        -
        -
        -
        -
        -
        -
        -
```

*new CLB created*
*CLB is in FG mode (two 4-input LUTs)*
*Connecting nets to input pins*
*Connecting nets to output pins*
*Connecting clock net to clock input pin.*
*The two DFF outputs (QX, QY) are connected to CLB outputs X and Y.*
*LUT output is connected to the first DFF (DX)*
*LUT output is connected to the second DFF (DY)*

Figure 4.14: Circuit generator for `Cell_MULT_LSB` on Xilinx 3000 FPGA architecture.

```
}
```

Figure 4.15: Circuit generator for `Cell_MULT_LSB` on Xilinx 3000 FPGA architecture. (continued)

- *Custom module constructor*, (`Module_MULT::Module_MULT(···)`, Table 4.11): The functions of the custom module constructor are:
  1. Call the constructor of its base class `Module(···)`.
  2. Connect each input signal to each input pin (`in[0].insert(in0), in[1].insert(in1)`).
  3. Connect each output signal to each output pin (`out[0].insert(out0)`).
  4. Assign the latency of each output pin which is specified by the circuit designer. In the case of a two-input multiplier, the latency is $2N-1$ where $N$ is the width of the input signal `in0`.
- *Cell generator* (`Module_MULT::generate_cell()`, Table 4.12): The functions of the cell generator are
  1. Generate *buses* which connect the cells (`x[width * 2 - 2], sum[width + 1], y[width]`).
  2. Call custom cell constructors (`new Cell_MULT_LSB(this, x[0], y[0], x[1], sum[1]);`).
- *Custom cell constructor*, (`Cell_MULT_LSB::Cell_MULT_LSB(Module * mod, Bus * din, Bus * yin, Bus * dout, Bus * sout)`, Table 4.13): Like the module constructor, the cell constructor calls the constructor of its base class `Cell` and connects input and output signals to input and output pins.
- *Circuit generator* (`Cell_MULT_LSB::generate_circuit()`, Table 4.14) : The functions of the circuit generator are
  1. Generate nets which connect the logic elements (`Net * xin = din->search_net("high");`).
     Call custom logic constructors (`clb = new Clb(this, "YREG");`).
  2. Connect input nets to input pins of each logic element (`clb->input(yin, yreg, hout, xin, 0);`).
  3. Connect output nets to output pins of each logic element (`clb->output(yreg, xout);`).
  4. Define configuration of each logic element:
     (a) Functional mode (`clb->base = "FG";`).
     (b) D-flip-flop configuration (`clb->DFF(clk, 0, 0, X, Y);`).
     (c) LUT configuration (`clb->equateF("A*C+B*C̃", DX);`).

## 4.9 Memory Interface Model

Here let us consider the SRAM interface model for the coprocessor to access the local memories. In a conventional digital hardware design, designing SRAM interface circuit requires special attention because of the asynchronous nature of the device, especially for the write cycle. To be more specific, during the period where the write enable signal is active, the address signals have to be stable. In Fig.4.16, the address setup time ($t_{AS}$) and the write recovery time ($t_{WR}$) cannot be negative (for recent fast SRAMs, the setup time and the write recovery time can be 0 ns, but nevertheless cannot be less than 0). It is extremely difficult to implement a single cycle write operation since the the write enable signal is subjected to a two-sided timing constraints relative to *all* the address signals. If the write enable signal and the address signals were generated from the circuits using the same clock, in order to implement a single cycle clock operation, the following constraints have to be met:

$$\begin{array}{rcl} D_{WE} - D_{addr}^{max} & \geq & t_{AS} \\ D_{addr}^{min} - D_{WE} & \geq & t_{WR} \end{array} \tag{4.3}$$

where $D_{WE}$ is the propagation delay from the clock input to the write enable input on the SRAM, $D_{addr}^{max}$ is the largest propagation delay from the clock input to the address input on the SRAM, and $D_{addr}^{min}$ is the smallest

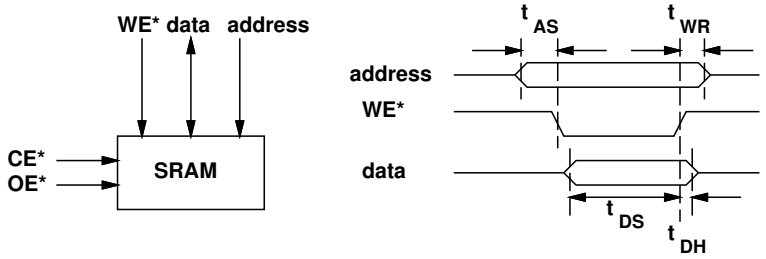



Figure 4.16: SRAM write cycle timing. Asynchronous SRAM imposes a double-sided timing constraint on write enable signal ($WE*$) relative to the address signals, making it almost impossible to implement a single-cycle write operation.

propagation delay. Since $D_{addr}^{max} \geq D_{addr}^{min}$, these constraints imply that the the address setup time and the write recovery time both have to be 0, delays of all address lines and the write enable signal have to be exactly the same. This timing constraint is not practical for FPGA designs. The conventional solution is to use three clock cycles: first cycle for activating the address lines, second cycle for activating the write enable signal, and the third cycle for deactivating the write signal. Our solution for implementing a single cycle write operation is to generate the write enable signal at the coprocessor controller which also generates the event clock for the datapath. This is illustrated in Fig.4.17. The actual write enable signal is generated at the coprocessor controller by taking the NOR of the event clock signal and the write enable signal generated by the datapath. Each SRAM needs separate write enable signal, and therefore multiple NOR gates are needed for multiple SRAMs. In order to analyze the timing of the write operation, let us define the delay values as in the figure:

- $D_{clk}$ is the delay of the event clock signal ($e\_clk$) from the clock generator output (inside the controller chip) to the memory control logic inside the FPGA. This delay includes the output buffer delay of the controller chip, the parasitic capacitance of the event clock line, and the clock buffer delay on the FPGA.
- $D_{WE}$ is the delay of the write enable signal ($WE*$) from the event clock generator to the SRAM. This delay includes the NOR-gate delay, output buffer delay of the controller chip, and the parasitic capacitance of the write enable line to the SRAM.
- $D_{WE_{fpga}}$ is the delay of the write enable signal from the memory control logic inside the FPGA to the SRAM through the controller chip. This delay includes the clock-to-output delay of the flip-flop of the memory control logic, output buffer delay on the datapath, input buffer delay of the controller chip, NOR-gate delay, output buffer delay of the controller chip, and the parasitic capacitance of the write enable line to the SRAM.
- $D_{addr}^{max}$ is the largest delay of the address lines from the memory control logic. This delay includes the clock-to-output delay of the flip-flop of the memory control logic, output buffer delay on the datapath, and the parasitic capacitance of the write enable line to the SRAM.
- $D_{addr}^{min}$ is the smallest delay fo the address lines from the memory control logic.

Also, let $T_{clk}$ be the minimum event clock cycle period. Using these delay values, the timing constraints are:

1. Address setup time ($t_{AS}$):

$$\max(D_{WE} + T_{clk}/2, D_{WE_{fpga}} + D_{clk}) - (D_{addr}^{max} + D_{clk}) \geq t_{AS} \tag{4.4}$$

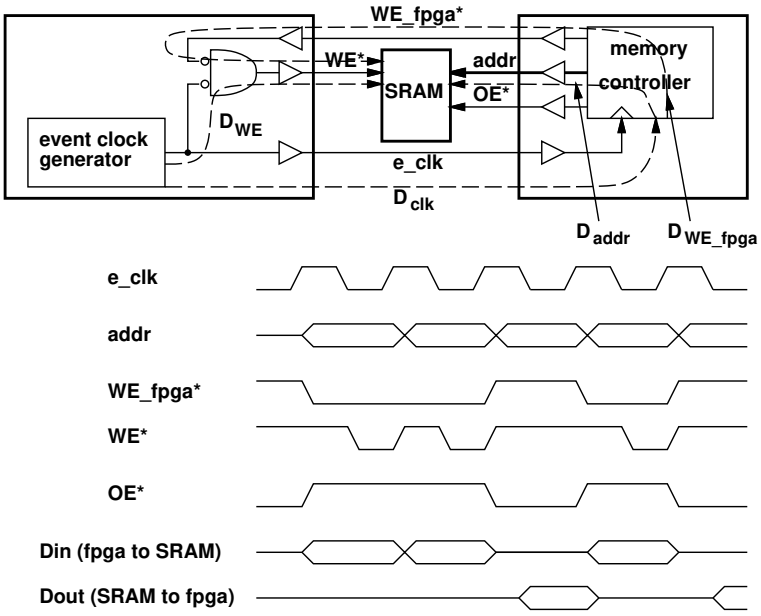



Figure 4.17: Single cycle write operation by taking the NOR of the event clock and the write enable signal from the datapath.

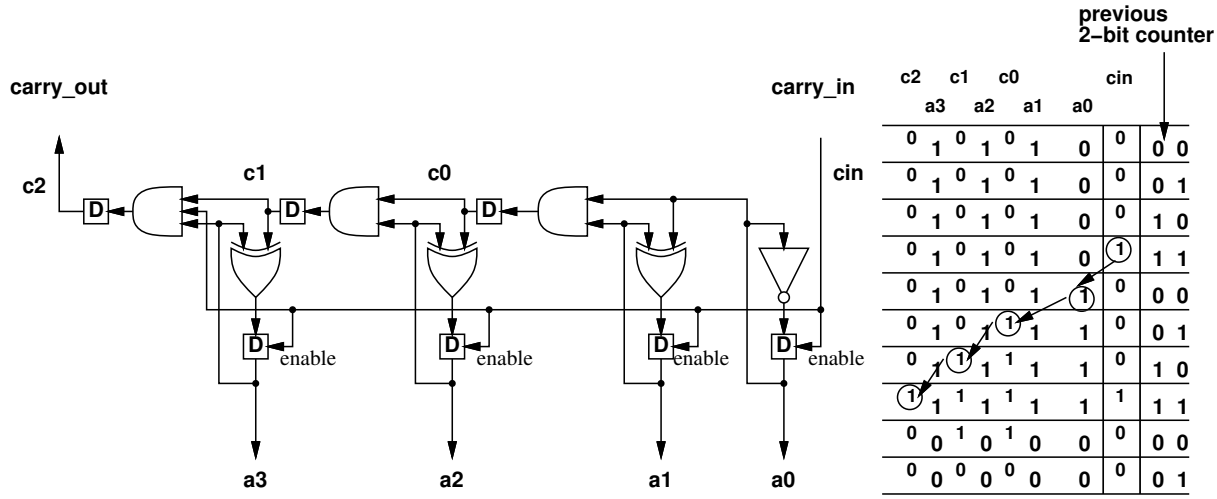
2. Write recovery time ($t_{WR}$):

$$(D_{addr}^{min} + D_{clk}) - \min(D_{WE}, D_{WE_{fpga}} + D_{clk}) \geq t_{WR}. \tag{4.5}$$

Let us first assume that $\max(D_{WE} + T_{clk}/2, D_{WE_{fpga}} + D_{clk}) = D_{WE} + T_{clk}/2$ and $\min(D_{WE}, D_{WE_{fpga}} + D_{clk}) = D_{WE}$. By setting the minimum event clock cycle period to be sufficiently long, such that $T_{clk} \geq 2 \cdot (t_{AS} + D_{addr}^{max} + D_{clk} - D_{WE})$, the address setup time constraint can be met. Next, by making the write enable signal delay from the controller ($D_{WE}$) sufficiently smaller than the address line delay ($D_{addr}^{min} + D_{clk}$), the write recovery time constraint can be satisfied. We can use a fast device for the controller chip to make $D_{WE}$ sufficiently small, or we can simply delay the event clock line from the clock generator so that $D_{clk}$ becomes sufficiently large. If $\max(D_{WE} + T_{clk}/2, D_{WE_{fpga}} + D_{clk}) = D_{WE_{fpga}} + D_{clk}$ or $\min(D_{WE}, D_{WE_{fpga}} + D_{clk}) = D_{WE_{fpga}} + D_{clk}$, the above scheme can still be applyed to meet the constraints since the constraints will only be relaxed by this instead of being tightened.

## 4.10 Ultra-Fast Counter for Simple Control Flow and Memory Address Generation

Here, let us consider the implementation of the control-flow. The major problem in implementing the state machine for implementing the control flow is that the state machine circuits are usually implemented in multi-level logic in which the circuit becomes slower than our high-speed bit-serial circuits having only single-level logic. A new type of counter structure have been developed which is implemented exclusively with a single-level logic structure for *arbitrary length* with one CLB per bit. This counter can be cascaded to implement also an arbitrary radix counter, which in return can be used to implement a simple but very fast state machine.




(a) Single–level logic counter

(b) Behavior

Figure 4.18: A single-level logic counter structure. There is a two-bit counter before this counter stage. With this single-level logic structure, this counter successfully computes the correct carry output for the next stage.

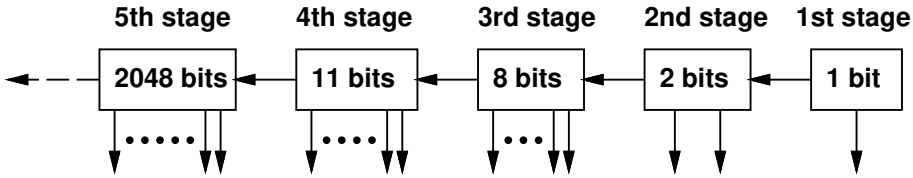



Figure 4.19: Cascaded structure of the arbitrary size single-level counter

The ultimate problem in implementing a counter is how to propagate the carry bits from the least significant bit to the most significant bit. A simple $N$-bit ripple-carry counter has $N$ carry-logic chains. By taking advantage of the LUT, 2-bits can be incremented at the same time, reducing the length of the array chain to $N/2$. Several high-speed counter designs are provided in Xilinx databooks which uses prescaler technique. The basic idea of the prescaler technique is to implement a small $n$-bit counter where $n$ is 3 or less with a single-level logic. The carry output of this mini-counter only becomes active every $2^n$ cycles, therefore the effective clock rate reduces by $2^n$ for the subsequent counter stages. Therefore, roughly $2^n$ ripple-carry stages can be added and still achieve the speed which is equivalent to single-level logic. With this technique, a 16-bit counter can operate as fast as 200 MHz. While this counter design is sufficient for many of the applications, a new technique in the counter design was discovered in the course of this thesis research which has a single-level logic structure which can be cascaded to arbitrary size using minimum number of logic blocks.

### 4.10.1 Arbitrary size single-level logic counter

The counter is decomposed into $K$ stages in which each counter stage has a carry input from the left stage and a carry output to the right stage. Carry output is set to 1 if all the output bits in the counter stage and the carry input is 1. This is no different from the ordinary counter. If we depend on the carry input value to compute the carry output, this still forms a ripple-carry chain and therefore a single-level logic implementation is not possible.

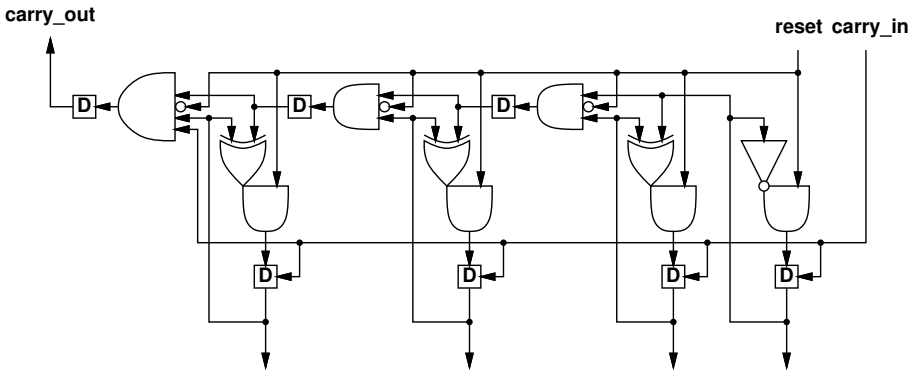



Figure 4.20: Single-level logic counter with synchronous reset.

The only way to break this carry chain is to compute the carry output without the carry input. By recognizing the fact that the $i$-th bit becomes active every $2^i$ cycles periodically, we can see how this is possible. At $k$-th counter stage, let its least significant bit be the $i$-th bit of the whole counter. Since its carry input only becomes active every $2^i$ cycles, it only needs to compute whether or not to set a 1 on the carry output every $2^i$ cycles as well. With this in mind, the counter stage is designed so that it automatically sets a 1 on the carry output precisely $2^i$ cycles after its value becomes $11\cdots11$. Actually, this can be achieved with a surprisingly simple circuit which is the carry-save counter. Fig.4.18 shows the circuit of the single-level logic counter stage. We assume that there is a 2-bit counter stage before this. Fig.4.18(b) shows the actual counting procedure of this counter stage. As we can see, the pipelined carry chain evaluates if the output bits are all ones which takes precisely 4 clock cycles. Therefore, by the time it computes that all of the bits are 1's and sets the carry output to 1, the carry input becomes 1 at the precisely same moment. At the $k$-th counter stage where its least significant bit is the $i$-th bit of the whole counter, the length of this stage becomes $2^i$ bits. Therefore, the $k+1$-th counter stage starts from $(2^i+i)$-th stage. Calculating the number of bits for each stage reveals an interesting mathematics, where for any practical application, we don't need more than four counter stages since the fourth counter stage grows so large (Fig.4.19):

| | | |
|---|---|---|
| First stage | : | $2^0 = 1$ bit |
| Second stage | : | $2^1 = 2$ bits |
| Third stage | : | $2^{1+2} = 8$ bits |
| Fourth stage | : | $2^{1+2+8} = 2048$ bits |
| Fifth stage | : | $2^{1+2+8+2048}$ bits |
| Sixth stage | : | $2^{1+2+8+2048+2^{2059}}$ bits |

Each bit can be easily implemented with 1 CLB on Xilinx FPGA. Therefore our counter design is the *fastest* of any counters. This simple but a powerful design can be expanded to have synchronous reset (Fig.4.20), or can even be loadable (Fig.4.21), although the carry values ($c0, c1, c2$ in Fig.4.18) also have to be loaded.

### 4.10.2 Arbitrary Radix Counter

Now, using this high-speed counter, the control flow implementation scheme can be constructed. As I have described, the intention is not to implement an arbitrary state machine on the bit-serial datapath since that will cause the circuit to slow down significantly. Here, we focus on the fact that most of the control flows in many of the DSP applications are for-loops which can be implemented with counters. For nested for-loops (Fig.4.22(a)), we can

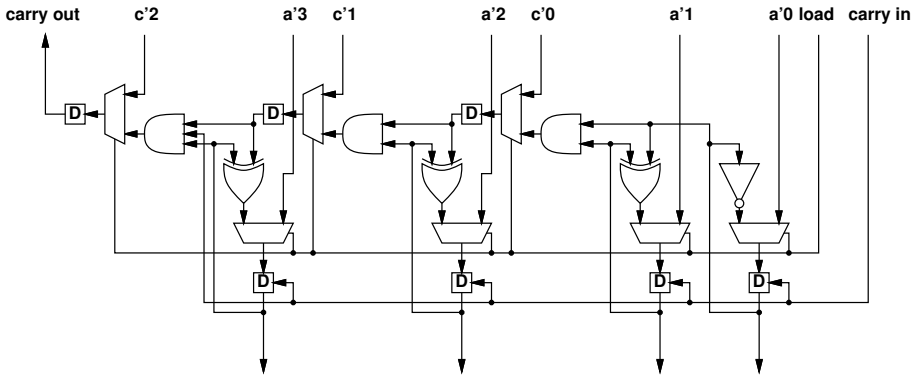
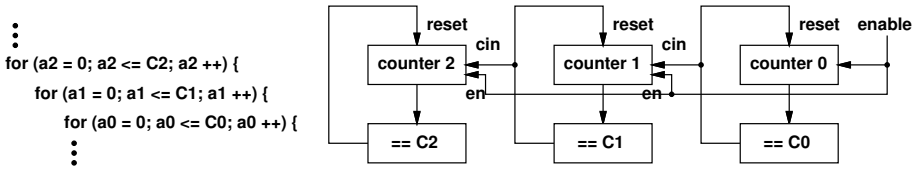



Figure 4.21: A loadable single-level logic counter.




(a) Nested for−loops.

(b) Cascaded counters implementing nested for−loops.

Figure 4.22: Cascaded counters implementing a nested for-loops.

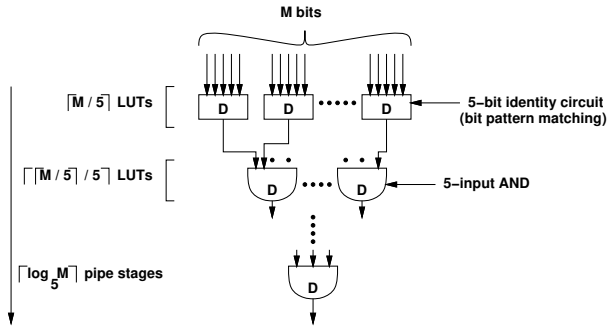



Figure 4.23: A pipelined magnitude comparator.

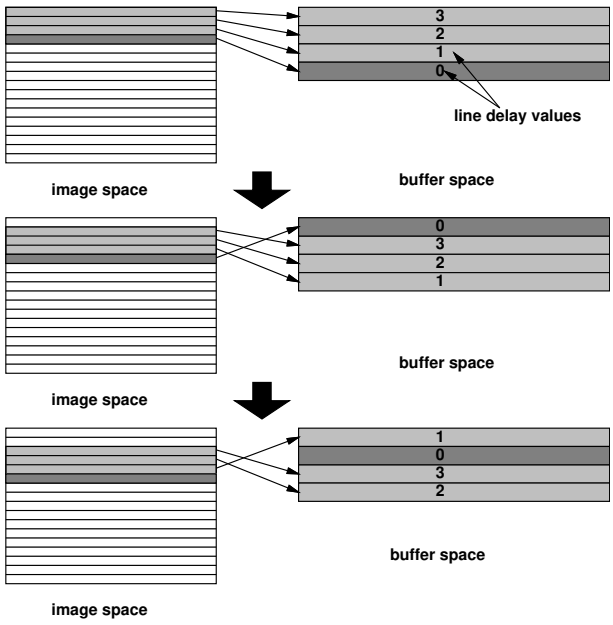



Figure 4.24: Line delay implementation using local RAM.

use a series of cascaded counters. Fig.4.22(b) illustrates the basic structure of the cascaded counters. Each counter is accompanied with a comparator which checks whether or not the counter reaches the assigned maximum value ($C_i$), and resets the counter if it does. A comparator is also implemented in a single-level logic using a pipelined tree structure shown in Fig.4.23. The first pipe stage is an identity circuit where the input bits are compared with a specific bit pattern which is created from the maximum value of each counter. Subsequent stages are AND-gates. Each logic has 5 inputs to fit into one LUT. The number of pipe stage is $\lceil \log_5 M \rceil$. This reset signal essentially becomes the carry input to the next counter in the cascaded chain.

### 4.10.3 Line Buffer Implementation Using High-Speed Counters

For 2D image processing, line buffers are needed to implement the line delays in the vertical direction perpendicular to the horizontal pixel scanning direction. Let $k$ be the number of line delays needed on the datapath ($k$ pixels in the vertical direction). This $k$ line delays are implemented merely by buffering $k$ lines of image pixels in the RAM (Fig.4.24). We only need to assign $2^{\lceil \log_2 k \rceil}$ lines worth of address space in the RAM and wrap around the addressing. Address calculation involves three counters (Fig.4.25):

1. Head line pointer: This points to the vertical position of the current pixel. This is incremented every $M$ sampling cycles where $M$ is the size of the image in the horizontal direction.

2. Pixel pointer: This points to the horizontal position of the current pixel. This is incremented every sampling cycle.

3. Line pointer: This points to the vertical position of the RAM address space. This is calculated taking the head line pointer as the initial value and incrementing every clock cycle. This is implemented by the loadable counter which is described in the earlier section.

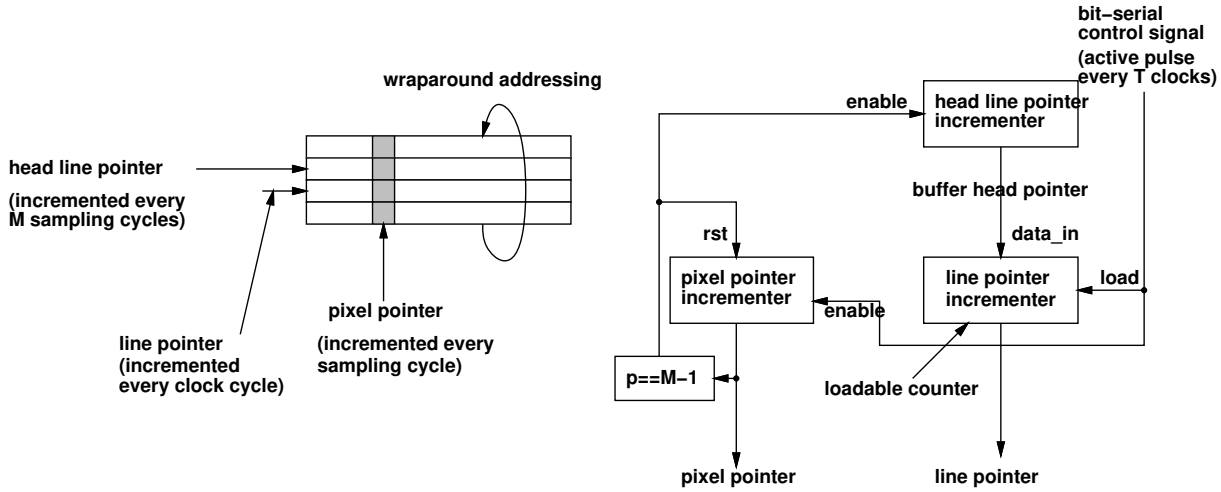



Figure 4.25: Address calculation of line delay RAM.

At the first clock cycle of the sampling period, the value of the line pointer is the same as the value of the head line pointer. At this cycle, the pixel data is written on the RAM. For the subsequent cycles, the line pointer is incremented and the pixel values stored in the RAM is read back from the RAM which are the delayed pixel values in the vertical direction.

## 4.11 High Throughput Pipelined Datapath Structure

In the first part of this thesis, a number of bit-serial operator designs have been described which operates as a pipeline on bit-level which exhibits high throughput with small hardware resource. In order to take full advantage of this bit-level pipelined circuits, the datapath is constructed so that each module is also pipelined at the functional-level as well. This approach is also taken in the early bit-serial silicon compilers as well [73], [58]. With this approach, the architecture synthesis program will produce a datapath structure where each hardware functional unit operates at identical speed (unified logic structure using 1-LUT sequential logic) with identical throughput which produces minimal area overhead for flow control and busing, and no idle cycles on acyclic datapath structures, therefore an area efficient and high throughput circuit.

This section goes over the basic procedures of deriving this bit-serial pipeline datapath from a behavioral description input. CATHEDRAL-I addresses most of the problems [58], and therefore the main effort here was to adopt their methods and make necessary modifications to apply on configurable coprocessing paradigm.

**Behavioral description capture** As described eaclier in this chapter, C++ is used to describe the algorithm. It is described in terms of *difference equations* which is used to describe time-discrete systems. Also, control flow description such as for-loops and if-else statement are allowed as well. In order to generate a pipelined datapath structure, each distinct operation is simply assigned to a physical hardware module. Here, an operation being *distinct* is defined by the following:

1. Two operations are *indistinct* if the functionality of the two together with the set of input signals are both identical among the two. Here, *functionality* is the type of operation.
2. Two operations are *distinct* if either the functionality of the two or the set of input signals are not identical.

**Minimum data sampling period evaluation** The data sampling period is initially determined by the signal word length $N$. However, if the datapath includes a cycle, the sampling period may need to be longer in order for the feedback computations to terminate before the next sampling cycle. This minimum data sampling period evaluation involves analyzing the critical loop and calculating the latency of each loop.

**Pipeline Synchronization** After employing a one-to-one mapping of distinct operations to hardware functional modules, the pipeline network needs to be synchronized so that the arrival time of all inputs to each functional modules are identical. In other words, the LSB of all inputs have to arrive at the same clock instance. To achieve this, shift registers are added to delay the arrival time of signals which reaches a module earlier than the other inputs. This pipeline synchronization problem can be formulated as a single-source longest paths problem, a variation of a well-known single-source shortest paths problem [24].

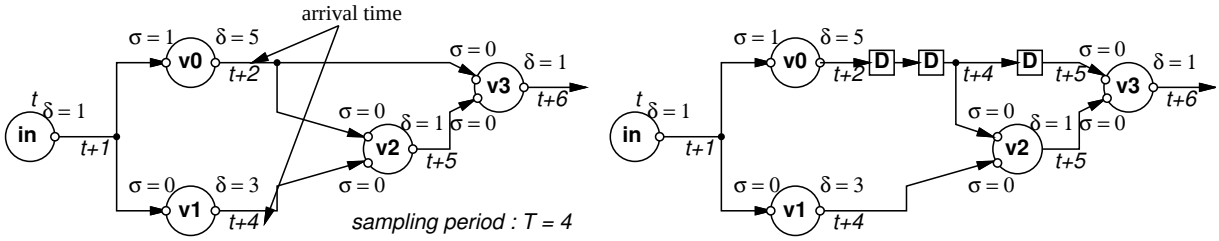
**Pipeline Register Minimization** The number of pipeline registers needed to synchronize the whole pipeline network can grow to a large number where the pipeline registers alone can dominate the total hardware size. Therefore, we need to minimize the number of pipeline registers in a synchronized pipeline network. Note that there can be a number ways in which a pipeline network is synchronized. Solving the single-source longest paths problem obtains a feasible solution, but not necessary an optimized one. This pipeline register minimization problem is studied by many and it is known that simplex method for linear programming can be applied [49], [66]. In this pipeline synthesis system, the simplex method is chosen because of its simplicity.

**Interface IO scheduling** There are two types of interface IOs connecting to the datapath, which are system bus interface and RAM interface. These share the same data lines for reads and writes. The data communications on these interfaces have to be scheduled to avoid bus conflict among these communications. This is done by statically scheduling the interface IOs so that none of the communications on the same interface overlap each other.

**Library-Based Logic Synthesis and Technology Mapping** As presented in Chapter 3, the bit-serial operators all have the same 1-LUT logic structure whose outputs are all latched by a D-flip-flip (DFF). It is therefore natural for us to establish a bit-serial module library at the logic block level rather than gate level in order to assure that the critical path length is always 1 LUT deep by bypassing logic minimization and technology mapping. These circuit libraries are also coded in C++ as a set of circuit generators which are called by each functional module.

## 4.12 Pipeline Synchronization

This bit-serial datapath synthesis program generates a bit-level pipeline network where each bit-serial module computes the data continuously and produces the result every $T$ clock cycles where $T$ is the *sampling period*. Each module starts computing a new data as soon as it sees the least significant bit (LSB) of all the input signals and produces the output some cycles later (Fig.4.26(a)). Recall the definition of operation latency in Chapter 3 where the latency $\delta$ is defined to be the interval between the arrival time of the $T-1$-th LSB after the decimal position of the input and the output. Here, let us call the $T-1$-th LSB after the decimal position *the high-LSB* (since it is the LSB of the *high bit*), and the time of arrival of the high-LSB inputs the *activation time* of a module. In order for the bit-serial circuits to operate properly, we have to assure that the LSBs of all the input signals arrives at the same module simultaneously. We assume that both $T$ and $\delta$ for all modules are constant, therefore the behavior of the pipeline network is static. Because of this static nature of the pipeline, the arrival time slots of the LSB of any signal is strictly periodical with a period of $T$ cycles. Therefore merely by adding shift registers as delay elements, we can control the arrival time of the signals to a module to assure input synchronization (Fig.4.26(b)). This pipeline synchronization which is done on the module-level netlist $HG(\overline{E_m}, V_m)$ involves two steps:




(a) A pipeline network before synchronization.

(b) A synchronized pipeline network.

Figure 4.26: Pipeline synchronization. $\sigma$ is the input sampling delay value and $\delta$ is the output latency value. Sampling period $T = 4$. The arrival time of the signal $\overline{e}_0 = (\{v_0\}, \{v_2, v_3\})$ is calculated as $t - T \cdot \sigma + \delta = t - 3 + 5 = t + 2$. At vertex $v_2$, the two inputs have a different activation time of $t + 2$ and $t + 3$. Therefore, 2 register elements have to be inserted on the first input so that the they become the same activation time. Same with vertex $v_3$.

**ASAP pipeline scheduling on $HG(\overline{E_m}, V_m)$**

1. Set the activation time of the primary input modules to 0. A *primary input module* is a module without any inputs.
2. For each module, calculate the longest path from all primary *input modules*. This multiple-source longest path problem can be converted into a single-source longest path problem by inserting a dummy source node which connects to all the primary input modules and calculate the longest path from that dummy source node.

**Pipeline register insertion**

1. For each module, insert shift registers in order to equalize the distance from the primary input nodes. One register has a latency of 1.

The reason why we call this scheduling the "ASAP (As-Soon-As-Possible) pipeline scheduling" is that calculating the longest distance of each module from the primary input modules is equivalent of calculating the earliest possible activation time after the activation of all primary input modules. Here, the assumption is that all primary inputs are activated at $t = 0$. Note here that calculation of the longest path of each module can be done by modifying the well-known algorithm for the single source shortest paths problem by Bellman and Ford [24]. The algorithm takes in a weighted graph $G(E, V, W)$ where $W$ is the set of weight values assigned to each edge in $E$ (Table 4.27). This algorithm permit loops in the graph as long as all the loop weights are *non-positive*. Otherwise, the algorithm outputs an "infeasible" flag. This algorithm runs in $O(|E||V|)$. Note that the original single-source shortest paths problem permits loops with *non-negative* loop weights.

### 4.12.1 Conversion from Directed Hyper-Graph to Directed Graph

Since the algorithm described above works on weighted graph $G(E, V, W)$, we have to convert our hyper-graph circuit representation $HG(\overline{E_m}, V_m)$ a directed graph $G(E_m, V_m)$. Also the notion of *weight* on edges to calculate the length of a path has to be redefined for our module-level netlist (Fig.4.28).

1. A hyper-edge $\overline{e} \in \overline{E_m}$ with multiple destinations

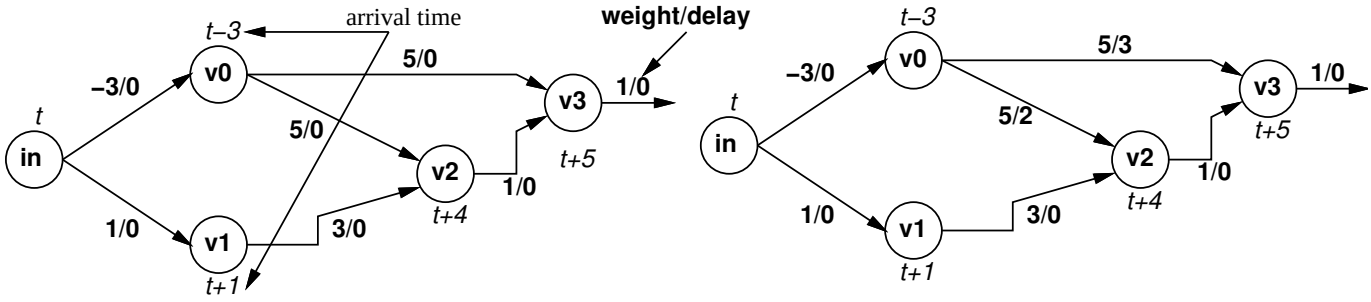
$$\begin{aligned} \overline{e} \quad = \quad & (\{u\}, \{v_0, \cdots, v_{k-1}\}) \\ & u : \text{source vertex} \\ & \{v_0, \cdots, v_{k-1}\} : \text{destination vertices} \end{aligned} \tag{4.6}$$

   is decomposed into single fanout edges

$$\{e_i = (u, v_i) | i = 0, \cdots, k-1, e_i \in E_m\} \tag{4.7}$$

```
Calculate_longest_input_distance(Vertex v){
    d = −∞;
    For each edge e = (u, v) ∈ E
        if d < distance_in(u) + weight(e);
            d = distance_in(u) + weight(e);
    return d;
}
Longest_path(G(E, V, W)){
    For each vertex v ∈ V
        If v is a primary input
            distance_in(v) = 0;
        Else
            distance_in(v) = −∞;
    For i = 0 to |V| − 1
        For each vertex v ∈ V
            d = calculate_longest_input_distance(v);
            if distance_in(v) < d
                distance_in(v) = d;
    For each vertex v ∈ V
        d = Calculate_longest_input_distance(v);
        if distance_(v) < d
            return "infeasible";
    return "feasible";
}
```

Figure 4.27: Single-source longest path problem based on Bellman and Ford




(a) Graph representation of an unsynchronized pipeline network.

(b) Synchronized pipeline network graph..

Figure 4.28: Hyper-graph decomposition from the previous figure. The numbers attached to each edge represents the weight (left) and the delay (right). In (a), the pipeline network is not synchronized, and therefore all the delay values on the edges are 0s. In (b), the pipeline network is synchronized by inserting 2 delays on the edge $(v_0, v_2)$ and 3 delays on the edge $(v_0, v_3)$.

2. The *weight* of an edge $e = (u, v) \in E_m$, now having exactly one source vertex $u$ and one destination vertex $v$, is calculated by the following equation:

$$weight(e) = latency(e) - cycle(e) \cdot T \tag{4.8}$$

where $latency(e)$ is the latency of the output pin of $u$ ($\delta(u, v)$), $cycle(e)$ is the sampling delay of the input pin of $v$ ($\sigma(u, v)$), and $T$ is the sampling period.

By this graph decomposition, we are able to apply the longest path problem.

### 4.12.2 Critical Path Analysis and Minimum Sampling Period Calculation

As pointed out earlier, it has to guarantee that there are no loops with positive weight. A loop is defined as

1. A *loop* is a path whose source vertex and the destination vertex are identical, and no intermediate vertices are identical. A loop is expressed as a sequence of edges $(e_0, e_1, \cdots, e_{k-1})$ where the source vertex of $e_0$ and the destination vertex $e_{k-1}$ are the same ($e_0 = (u, v_i), e_{k-1} = (v_j, u)$). A self-loop, a loop with only one vertex, has only one edge whose source and destination are the same.
2. The weight of a loop is the sum of the weight of all edges which constitutes the loop.
3. A *critical loop* is a loop with the largest weight.

Note that by the definition of the edge weight in eq.(4.8), loop weights are determined not only by the accumulative latency along the loop, but also the accumulative sample delays and the sampling period $T$ itself. We need to determine the minimum sampling period $T$ which guarantees non-positive loop weights.

The weight of a loop $l = (e_0, e_1, \cdots, e_{k-1})$ is calculated by the following equation:

$$\begin{aligned} weight(l) &= \sum_{i=0}^{k-1} weight(e_i) \\ &= \sum_{i=0}^{k-1} latency(e_i) - T \cdot \sum_{i=0}^{k-1} cycle(e_i) \\ &= latency(l) - T \cdot cycle(l) \end{aligned} \tag{4.9}$$

where $latency(l)$ is the total sum of the latency values of the edges along the loop, and $cycle(l)$ is the total sum of sampling delay values of the edges along the loop. In order to fulfill the non-positive loop weight constraint, the following equation must be satisfied for all existing loops:

$$\begin{aligned} latency(l) - T \cdot cycle(l) &\leq 0 \\ latency(l) &\leq T \cdot cycle(l) \\ T &\geq latency(l)/cycle(l) \end{aligned} \tag{4.10}$$

where $l \in L$ and $L$ is the set of loops existing in the datapath. Let us define the critical loop period $LP_{max}$ as

$$LP_{max} = max\{latency(l)/cycle(l) | l \in L\}. \tag{4.11}$$

The sampling period $T$ is determined by the following equations to assure there are no loops with non-positive weight in the datapath:

$$T = \max(N, LP_{max}) \tag{4.12}$$

A straightforward way of calculating $LP_{max}$ is described below which takes $O(|E||V| \log |V|)$ time.

**Critical Loop Period Computation Algorithm** 1. Convert the directed hyper-graph $HG(E_m, V_m, P_m)$ (module-level netlist) into a weighted directed graph $G(E, V, W)$.

2. As shown below, do a binary search on $T$ by recursively applying the FB algorithm to determine $LP_{max}$ (Fig.4.29).

```
Calculate_critical_loop_period(G(E_m, V_m, W_m))
{
    latency_max := max({latency(e)|e ∈ E});
    LP := ⌈latency_max · |V_m|/2⌉;
    T := LP; (note that T determines the edge weights)
    δ = LP;
    repeat
        Update edge weight matrix W_m with the new sampling period T;
        if Longest_path(G(E_m, V_m, W_m)) = 'feasible'
            LP = T;
            T := T − δ;
        else
            T := T + δ;
        if δ = 1
            return LP (LP = LP_max);
        δ := ⌈δ/2⌉;
}
```

**Proof of correctness** 1. $latency\_max \cdot |V|$ is the maximum possible loop delay:
*Proof*: Given a graph $G(E, V, W)$, longest possible loop path by only counting the edge latency is when all the vertices in $V$ are connected as a single long loop and the edge weights along the loop all the largest weight $latency\_max$. Therefore, for any loop $l \in L$,

$$latency(l) \leq latency\_max \cdot |V|. \tag{4.13}$$

Also, a loop has to include at least one sampling delay, or else the computation includes mutually dependent calculations which causes a dependency deadlock. Therefore,

$$cycle(l) \geq 1. \tag{4.14}$$

And thus cycle periods of all loops are less than $latency\_max \cdot |V|$:

$$\begin{aligned} latency(l) &\leq latency\_max \cdot |V| \cdot cycle(l) \\ latency(l)/cycle(l) &\leq latency\_max \cdot |V|. \end{aligned} \tag{4.15}$$

2. The algorithm takes $O(|E_m||V_m| \log |V_m|)$ time.
*Proof*: It takes $O(|E_m||V_m|)$ time to execute the routine Longest_path(). Since it takes $O(\log(latency\_max \cdot |V_m|)) = O(\log(|V_m|))$ to do a binary search in the range of 0 and $latency\_max \cdot |V_m|$, the total running time is $O(|E_m||V_m| \log |V_m|)$.

the algorithm takes $O(|E_m||V_m| \log(|V_m|))$. When this algorithm terminates, it returns the minimum sampling period which guarantees a feasible pipeline scheduling.

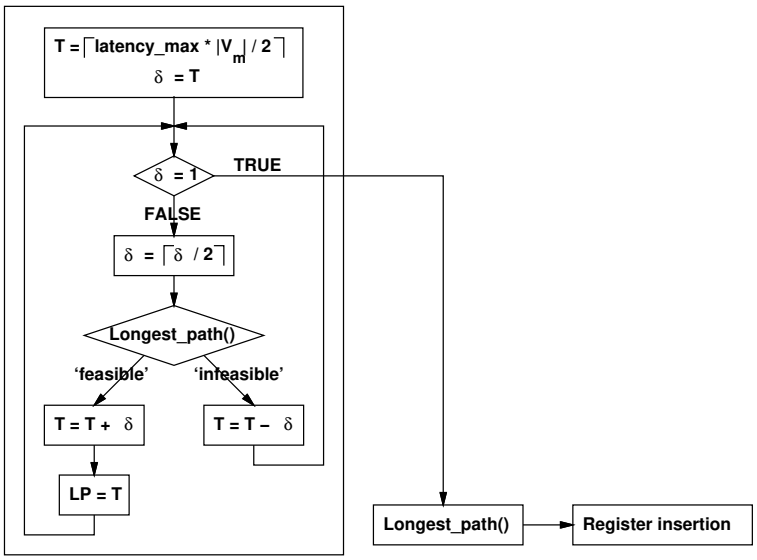



Figure 4.29: Pipeline synchronization method. First, a binary search is performed to determine the minimum feasible sampling period $T$. Then the pipeline is synchronized by applying FB longest paths problem and pipeline registers are inserted.

### 4.12.3 Pipeline Register Insertion

After the pipeline scheduling under the minimum sampling period, the number of pipeline registers for each of the module input signals is then calculated and recorded as $delay(e)$.

```
Insert_Pipeline_Register
{
    For each vertex v ∈ V_m
        For each edge e = (u, v) ∈ E_m
            delay(e) = distance(v) − (distance(u) + weight(e));
}
```

Fig.4.29 shows the whole flow of our pipeline synchronization method.

## 4.13 Bit-Serial Control Signal Routing

Bit-serial control signals indicate the word boundaries for the bit-serial circuits to reset the various signals in the circuits. Here, we say a control signal is *active* if it presents a 1-state (active) at one clock cycle before the arrival of the first active `high` data bit. The shift-register network on the pipeline which carries this bit-serial control signals are initialized to 0s after the power up of the system (flip-flops are reset on FPGAs). Control signal pulses are then generated at the primary inputs and propagated through the shift-register network as the clock cycles (refer to Section 3.2.1). Each location of the pipeline network first receives the active control signal at different times. During this system startup phase, the data signals are first inactive at all locations, and then active data signals propagate gradually from the primary inputs. Recall that our bit-serial circuits are designed to output 0s

when the inputs are 0s as well. This is guaranteed by designing the circuits so that all the outputs are active-high. Therefore, as long as the input data signals are inactive (0s), the output data signal is inactive regardless of the status of the control signals. However, if the input signals are active while the control signals are not, this will lead to an incorrect behavior since the circuit does not recognize the word boundary and thus executes the calculation as though the consecutive input signals were long data words. Therefore, active data signals have to be accompanied with active control signals.

Next, let us look into how the data signals, initially inactive, becomes active throughout the pipeline network. After the pipeline synchronization, all the paths from the primary inputs to a vertex have equal distance. At vertex $v \in V_m$, let $\{p_0, p_1, \cdots, p_{k-1}\}$ be the paths connecting the primary inputs and $v$ in a synchronized pipeline network graph $G(E_m, V_m, W_m, D_m)$ ($D_m$ is the matrix which is composed of the edge delay values calculated in the pipeline register insertion). A path $p_i$ is expressed as a sequence of edges $p_i = (e_{0,i}, e_{1,i}, \cdots, e_{n-1,i})$. Note that in this case, the source vertex of $e_{0,i}$ is one of the primary input vertices, and the destination vertex of $e_{n-1,i}$ is $v$. Let $distance(p_i)$ be the total distance of the path $p_i$:

$$\begin{aligned} distance(p_i) &= \sum_{e \in p_i} (weight(e) + delay(e)) \\ &= \sum_{e \in p_i} (latency(e) - cycle(e) \cdot T + delay(e)) \\ &= latency(p_i) - cycle(p_i) \cdot T + delay(p_i) \end{aligned} \tag{4.16}$$

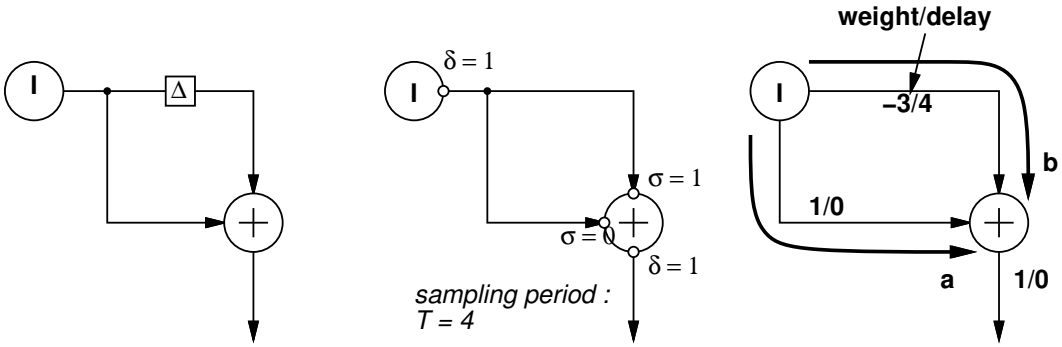
where $latency(p_i)$ is the total sum of the latencies along $p_i$, $delay(p_i)$ is the total sum of the pipeline registers along $p_i$, and $cycle(p_i)$ is the total sum of the sampling delays along $p_i$. Since all paths have equal distance, the following equations hold:

$$\begin{aligned} distance(p_0) &= distance(p_1) \\ &= \cdots \\ &= distance(p_{k-1}) \\ &= distance(v). \end{aligned} \tag{4.17}$$

However, note here that the distance includes the artificial negative delay values $cycle(p_i) \cdot T$ which was introduced to *logically* delay the data to the subsequent sampling periods but does not reflect the physical delay values. Let us define $physical\_distance(p_i)$ as the distance of path $p_i$ disregarding the sampling delay value $cycle(p_i)$:

$$\begin{aligned} physical\_distance(p_i) &= latency(p_i) + delay(p_i) \\ &= distance(v) + cycle(e) \cdot T. \end{aligned} \tag{4.18}$$

The physical distance actually indicates the signal activation time from the primary input activation, and therefore the signal with the smallest physical distance activates the earliest among the input signals at $v$. Consider the example shown in Fig.4.30(a) where one of the input to the adder is a delayed sample of the other. After the pipeline synchronization, four delay elements are added on $b$ in order to equalize the distances of the two paths. The four delay elements in fact correspond to the sampling period of four clock cycles. However, by this pipeline register insertion, the physical distance of $b$ is 4 cycles longer than $a$. This means that signal on $a$ becomes active one sampling period before $b$ does. In order to assure that the output data signal of a module, once active, is accompanied with an active output control signal, this output control signal has to be generated from the earliest activated input control signal, where in this case, the earliest activated input control signal is on $a$. In the case of Fig.4.30, simply initializing the pipeline network after the power up of the system, which is to feed the primary inputs with zeroes for a sufficient number of sampling periods in order to fill the pipeline network with active control signal pulses, will guarantee that the active data signals will be accompanied by active control signals, even if the

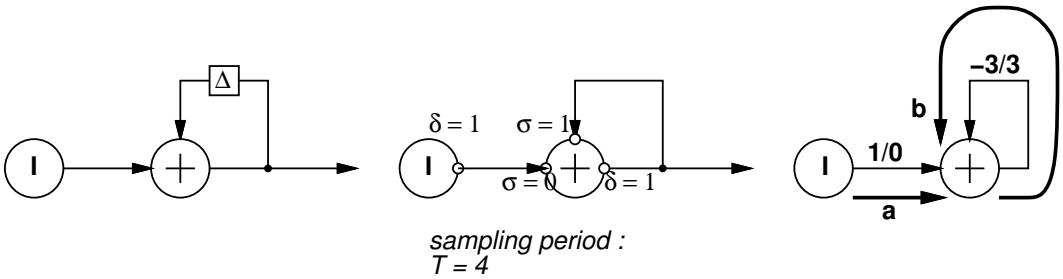



(a) Addition of an input data and its delayed sample data.

(b) Module–level netlist description of (a).

(c) Synchronized graph description of (b).

Figure 4.30: A simple example of an adder with a delayed sample input.




(a) An accumulator circuit.

(b) Module–level netlist description of (a).

(c) Synchronized graph description of (b).

Figure 4.31: An accumulator circuit requiring proper control signal routing.

earliest activated input control signal is not used at the adder. However, there are cases where an improper control signal routing to a module makes it impossible to distribute the active control signal pulses throughout the whole pipeline network. Consider the circuit in Fig.4.31. If the control signal from $b$ was used to generate the output control signal of the adder, signal $b$ will never have an active control signal since the control signal routing forms a loop. The control signal in $b$ is the source for the output control signal of the adder which is also $b$, and therefore $b$ is referring to itself for the control signal generation. In such situation, no matter how much initialization cycles is executed on the pipeline network, these loops in the control signal routing will never be filled with active control signal pulses. Therefore, in routing the control signals on the pipeline network, we have to avoid forming loops on the routing.

**Bit-Serial Control Signal Routing Algorithm** 1. Convert the module-level netlist $HG(\overline{E_m}, V_m)$ into a weighted directed graph $G(E_m, V_m, W'_m)$ with a modification on the weight calculation as follows:

- Instead of calculating the weight of each edge $e \in E_m$ as in eq(4.8), calculate the weight as

$$weight'(e) = cycle(e) \tag{4.19}$$

2. Solve the single-source *shortest path problem*. The shortest distance from the primary inputs to vertex

$v$ is calculated as

$$distance\_shortest(v) \quad = \quad min(\{distance\_shortest(u) + weight'(u,v) | (u,v) \in E_m\}) \tag{4.20}$$

3. For each bit-serial operator module $v$, route the control signal from one of the input signals such that routed control signal $e = (u,v) \in E_m$ satisfies

$$distance\_shortest(u) + weight'(u,v) = distance\_shortest(v) \tag{4.21}$$

**Proof of correctness** 1. The algorithm described above does not form a loop in the control signal routing.
*Proof* :

(a) There are only non-negative weight edges since $cycle(e) \geq 0$ for $e \in E_m$.

(b) Every loop in the pipeline network has at least one sampling delay, or otherwise this will create a mutual computation dependency among operations along the loop (delay-free loops). Therefore, every loop has a positive weight, i.e., $weight'(l) > 0$ for any loop $l$ in the pipeline network.

(c) Let $u$ and $v$ be any two vertices included in the same loop $l$. Let $p_u$ and $p_v$ be the shortest paths from the primary inputs to $u$ and $v$, respectively. Assume that there exists a path $p_u$ which includes $v$. Then,

$$cycle(p_u) \geq cycle(p_v), \tag{4.22}$$

since all edge weights along $p_u$ is non-positive. Then $p_u$ can be decomposed into $p_v$ and $p_{v,u}$ which is the shortest path from $v$ to $u$. Therefore,

$$cycle(p_u) = cycle(p_v) + cycle(p_{v,u}). \tag{4.23}$$

Under the assumption that there exists a path $p_u$ which includes $v$, let us consider the possibility of $u$ being included in $p_v$. If $u$ is included in $p_v$, $p_v$ can be decomposed into $p_u$ and $p_{u,v}$ which is the shortest path from $u$ to $v$, and therefore

$$cycle(p_v) = cycle(p_u) + cycle(p_{u,v}). \tag{4.24}$$

Since $cycle(p_{v,u}) > 0$ and $cycle(p_{u,v}) \geq 0$, from eq.(4.23) and eq.(4.24),

$$cycle(p_{v,u}) = cycle(p_{u,v}) = 0. \tag{4.25}$$

However, this implies that loop $l$ is a delay-free loop, and therefore this equation creates a contradiction to the property described in 1.(b). Thus, for any $u$ and $v$ who are included in loop $l$ of the pipeline network where $p_u$ and $p_v$ are the shortest paths from the primary inputs to $u$ and $v$, respectively, $u$ cannot be included in $p_v$ if $v$ is included in $p_u$, and vice versa.

(d) Since there does not exist a vertex pair in a loop whose shortest paths include each other from the argument in 1.(c), the bit-serial control signal routing does not form a loop.

2. Any vertex other than the primary inputs has a control signal routing path to one of the primary inputs. Here, it is assumed that there is at least one primary input node in the graph, and each node can is can be reached at least one of the primary inputs. This assumption is true for most of the signal processing algorithms where there is at least one signal input.
*Proof* :

(a) All vertices have a path to at least one of the primary inputs on the pipeline network. Otherwise, the vertex will be isolated from the input data to the pipeline which will have no influence in the computation.

(b) Each vertex other than the primary inputs have a control signal routed from another vertex due to step 3 in the algorithm.

(c) Since the control signal routing does not form a loop, each control signal originates from some terminal in the routing. From 2.(b), this terminal cannot be other than the primary inputs.

3. Control signal routing to a vertex $v$ is the shortest path of $v$.
*Proof* :

(a) There exists a control signal routing path from one of the primary inputs to $v$ ($p_v$= $((u_0, u_1)$,$(u_1, u_2)$, $\cdots$,$(u_{k-1}, v))$, $u_0$ : primary input) by the arguments in 2.

(b) At vertex $u_1$ which is the first vertex after the primary input vertex $u_0$ on the path $p_v$,

$$distance\_shortest(u_0) + weight'(u_0, u_1) = distance\_shortest(u_1). \tag{4.26}$$

Therefore, the path $p_{u_1} = ((u_0, u_1))$ which is included in $p_v$ ($p_{u_1} \subset p_v$) is the shortest path from the primary inputs to $u_1$.

(c) At vertex $u_i$, assume that the path $p_{u_i} = ((u_0, u_1), (u_1, u_2), \cdots, (u_{i-1}, u_i))$ is the shortest path from the primary inputs to $u_i$. Since

$$distance\_shortest(u_i) + weight'(u_i, u_{i+1}) = distance\_shortest(u_{i+1}), \tag{4.27}$$

the path $p_{u_{i+1}} = (p_{u_i}, (u_i, u_{i+1}))$ is the shortest path from the primary inputs to vertex $u_{i+1}$.

(d) By induction from 3.(b) and 3.(c), $p_v = ((u_0, u_1), (u_1, u_2), \cdots, (u_{k-1}, v))$ is the shortest path from the primary inputs to vertex $v$.

By this algorithm, the control signals are properly routed on the pipeline network such that no initialization cycles are needed before the actual computation starts in order to guarantee correct behavior of the pipeline network.

## 4.14 Pipeline Synchronization Register Minimization

In Section 4.11, the methodology of pipeline synchronization have been described which is one of the essential part of the bit-serial datapath synthesis. The single-source longest paths problem (primary inputs being the source) was used to formulate the pipeline scheduling scheme which can be viewed as *As-Soon-As-Possible* (ASAP) scheduling since each bit-serial module is activated at the earliest possible time after the activation of the primary inputs. This is, however, only one out of many pipeline synchronization solutions. One can formulate *As-Late-As-Possible* (ALAP) scheduling scheme by making the primary outputs as the source and inverting the direction of the edges in the hyper-graph $HG(E_m, V_m, P_m)$.

**ALAP pipeline scheduling** 1. Set the activation time of the *primary output modules* to 0. A *primary output module* is a module without any outputs.

2. For each module, calculate the longest path from all primary *output module*. This multiple-source longest path problem can be converted into a single-source longest path problem, similarly to the ASAP pipeline scheduling case, by inserting a dummy sink node which connects to all the primary output modules and calculate the longest path from that dummy sink node. In solving the single-source longest path problem in this case, invert the direction of all edges and negate the weight values of all edges.

3. Let $distance\_out(v)$ be the longest distance from the primary output modules, and $distance\_input = min(\{distance\_out(v) | v \in V\ v$ : primary input$\})$. Increment the distance of each module by $-distance\_out\_min$:

$$\begin{aligned} distance(v) \quad := \quad & distance(v) - distance\_out\_min \\ & (v \in V), \end{aligned} \tag{4.28}$$

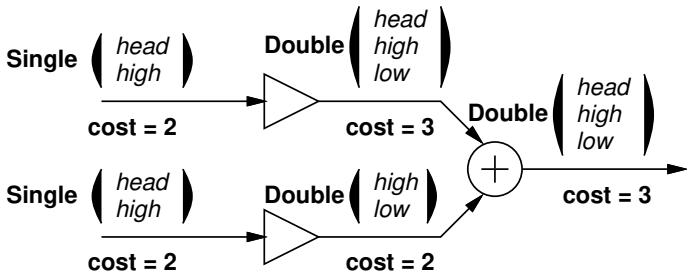



Figure 4.32: Edge cost of each signal.

thereby adjusting the activation time of the farthest primary input from the primary output, or in other words the earliest primary input, to 0 , or $distance\_input = 0$.

With this ALAP scheduling scheme, each bit-serial module is activated at the latest possible time before the activation of the primary outputs. Given that the activation time of the earliest primary input module $distance\_input$ and the activation time of the latest primary output module $distance\_output = max(\{distance\_in(v)|v \in V, v : \text{primary output}\})$ is fixed, $distance\_in(v)$ and $distance\_out(v)$ gives the lower bound and the upper bound, respectively, of the possible scheduling at module $v \in V$. Since pipeline synchronization registers increases the hardware size, we need to find a optimal scheduling which minimizes the number of pipeline synchronization registers.

A popular approach for the register minimization problem is to formulate a linear programming on a synchronized pipeline network and finding the optimal scheduling by moving each module's activation time locally [66]. It is known that this linear programming formulation is an LP-dual of the minimum cost flow problem and therefore can be solved in polynomial time [49]. In practice, this linear programming, although technically an integer linear programming, is usually solvable in low order polynomial time using the classical simplex method which guarantees an optimal solution [51]. Expantion of this formulation can deal with multiple fanout signals where pipeline register sharing occurs which will be discussed later. More recent work by Hu et al. [49] formulated a linear programming from an unsynchronized pipeline network. However, this pipeline network has to be guaranteed a feasible pipeline scheduling, i.e., sampling period has to be predefined such that positive weight loop does not exist in the network. The former approach of linear programming formulation of a synchronized pipeline network has been chosen for implementation in the bit-serial pipeline synthesis tool for the following reasons:

1. Linear programming formulation is simpler.
2. Since the minimum sampling delay is calculated by means of pipeline scheduling and therefore pipeline scheduling comes virtually for free, the latter method of solving the pipeline scheduling and the register minimization problem is not much of an advantage.

The problem formulation for the pipeline register minimization is described as follows:

1. From a module-level netlist $HG(E_m, V_m, P_m)$, generate a weighted directed graph $G(E, V, W, D, C)$. Here, it is assumed that there are only two point edges in $E_m$, therefore $E = E_m, V = V_m$. Multiple fanout hyper-edges will be dealt with later.
   (a) For each edge $e \in E$, assign a *cost* which indicates how many physical nets are included in the edge (Fig.4.32). Recall that each signal have at most four types of nets (head, high, low, over).
   (b) For each edge $e = (u, v) \in E$, assign a *delay* value as follows (Fig.4.33):

$$\begin{aligned} delay(e) &= distance(v) - distance(u) - weight(e) \\ &\quad (e = (u, v) \in E) \end{aligned} \tag{4.29}$$

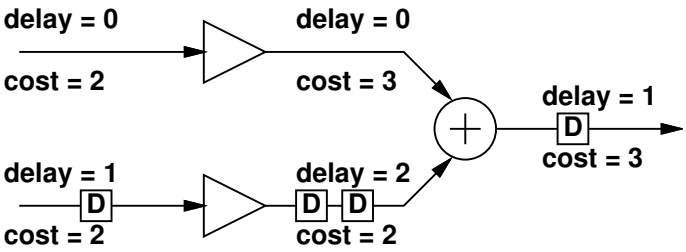



Figure 4.33: Edge delay of each signal.

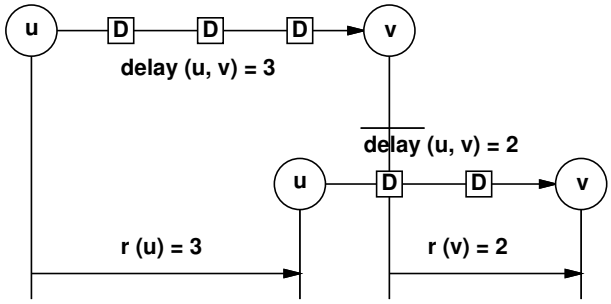



Figure 4.34: Pipeline retiming. After node $u$ and $v$ have been delayed their activation times by 3 clocks and 2 clocks, respectively, the edge delay is reduced by 1.

Here, $distance(v)$ is the longest distance from the primary inputs which is calculated in Longest_path() described in the previous section.

2. From the weighted graph $G(E, V, W, D, C)$, rearrange the pipeline registers and generate a modified weighted graph $G(E, V, W, \overline{D}, C)$ while maintaining the pipeline synchronism so as to minimize the number of pipeline registers.

   (a) Let $r(v)$ denote the difference between the activation time (retiming) of the modified graph $G(E, V, W, \overline{D}, C)$ from the original graph $G(E, V, W, D, C)$ at vertex $v$. For an edge $e = (u, v)$, the following equation holds between the delay values of the original graph ($delay(e)$) and the modified one ($\overline{delay}(e)$) (Fig.4.34):

$$\begin{aligned} \overline{delay}(e) &= delay(e) + r(v) - r(u) \\ & \quad (e = (u, v) \in E). \end{aligned} \tag{4.30}$$

   (b) The total number of pipeline registers is calculated as

$$\begin{aligned} z &= \sum_{e \in E} \overline{delay}(e) \cdot cost(e) \\ &= \sum_{e \in E} (delay(e) + r(v) - r(u)) \cdot cost(e) \\ &= PR_{init} + \sum_{v \in V} r(v) \cdot f(v) \end{aligned} \tag{4.31}$$

where $PR_{init} = \sum_{e \in E} delay(e)$ which is the total number of pipeline registers in the original graph, and $f(v)$ is

$$f(v) = (\sum_{(u,v) \in E} cost(u,v) - \sum_{(v,u) \in E} cost(v,u)). \tag{4.32}$$

As seen here, $z$ is a function of the retiming value of each vertex ($r(v)$). Therefore minimizing $z$ is equivalent to minimizing

$$z' = \sum_{v \in V} r(v) \cdot f(v) \tag{4.33}$$

3. Equations (4.31), (4.32) and (4.33) form a linear programming problem :

- Given a graph $G(E,V)$, minimize

$$z' = \sum_{v \in V} r(v) \cdot f(v) \tag{4.34}$$

with subject to

$$\begin{aligned} &\overline{delay}(e) + r(u) - r(v) = delay(e) \quad (e = (u,v) \in E), \\ &\overline{delay}(e) \geq 0 \qquad\qquad\qquad\quad (e \in E) \\ &r(v) \geq 0 \qquad\qquad\qquad\qquad (v \in V) \\ &\overline{delay}(e), r(v) : \text{ integer} \end{aligned} \tag{4.35}$$

Note that the condition $r(v) \geq 0 (v \in V)$ which is needed to solve this problem using the simplex method can be guaranteed because the delay value of each edge is calculated from ASAP pipeline scheduling. The activation time of each module derived by the ASAP scheduling is the earliest possible time, and therefore the activation time derived by any other scheduling cannot be earlier. Thus the retiming values of all modules are non-negative.

It has been pointed out that by taking the linear programming dual of this problem, a minimum-cost flow problem is obtained in which a polynomial time algorithm exists. However, it was chosen to apply the classical simplex method directly to solve the pipeline synchronization optimization problem for its simplicity. Following describes a basic concept of the simplex algorithm.

**Simplex Algorithm** 1. *Input* :

$$\begin{aligned} \boldsymbol{A} &: M \times N \quad (N \geq M) \\ \boldsymbol{b} &: 1 \times M \quad \text{(non-negative constants)} \\ \boldsymbol{c} &: 1 \times N \\ \boldsymbol{x} &: 1 \times N \quad \text{(non-negative variables)} \end{aligned} \tag{4.36}$$

2. *Objective* : Minimize

$$z = \boldsymbol{c}^T \boldsymbol{x} \tag{4.37}$$

with subject to

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} &= \boldsymbol{b}, \\ x_i &\geq 0 \quad (i = 0, 1, \cdots, N-1). \end{aligned} \tag{4.38}$$

3. *Method* :

(a) *Find a basic feasible solution $\boldsymbol{x}'$ where $N-M$ elements of $\boldsymbol{x}'$ are 0.*
This solution is given by transforming the matrix equations by means of pivot operations to

$$\begin{array}{rcl} \boldsymbol{c}'^T\boldsymbol{x}' & = & z-\bar{z} \\ \boldsymbol{A}'\boldsymbol{x}' & = & \boldsymbol{b}' \end{array} \tag{4.39}$$

where

$$\begin{aligned} \boldsymbol{A}' &= \begin{pmatrix} 1 & 0 & \cdots & 0 & a'_{0,M} & \cdots & a'_{0,N-1} \\ 0 & 1 & \cdots & 0 & a'_{1,M} & \cdots & a'_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & a'_{M-1,M} & \cdots & a'_{M-1,N-1} \end{pmatrix} \\ &= (\boldsymbol{I}_M|\boldsymbol{a}'), \\ \boldsymbol{c}' &= \begin{pmatrix} 0 & \cdots & 0 & c'_M & \cdots & c'_{N-1} \end{pmatrix}, \end{aligned} \tag{4.40}$$

and all elements in $\boldsymbol{b}'$ are non-negative, and $\boldsymbol{x}'$ is same as $\boldsymbol{x}$ but the ordering of the elements is changed. From these equations, the basic feasible solution $\boldsymbol{x}'$ is calculated as

$$x_i = \begin{cases} b'_i & (i=0,1,\cdots,M-1) \\ 0 & (i=M,M+1,\cdots,N-1) \end{cases} . \tag{4.41}$$

Because of the constraint that all elements in $\boldsymbol{b}'$ must be non-negative, this transformation is generally not trivial. Simply applying pivot operations may lead to some basic solution but a non-feasible one. A general solution to this problem is to apply yet another simplex method to obtain the initial feasible basic solution.

(b) *Find a new basic feasible solution which makes the cost $z$ equal to or smaller than the current cost. Continue this step until $z$ can no longer be decreased.*
This is done by examining the values of elements in $\boldsymbol{c}$, exchanging column $m$ $(0 \le m \le M-1)$ and column $n$ $(0 \le n \le N-1)$ of $\boldsymbol{A}'$, $\boldsymbol{c}^T$, and $\boldsymbol{x}^T$, and maintaining the structures of eq.(xx) and (yy) by means of pivot operations which takes $O(MN)$ operations. The columns $m$ and $n$ are picked so that the amount of decrease in $z$ is maximized while the feasibility ($b'_i \ge 0$ for $i=0,1,\cdots,M-1$) is maintained after the pivot operations.

Besides its simplicity, the simplex method can solve our pipeline synchronization optimization problem quite efficiently:

1. In general case, the simplex method first have to be given a *feasible basic form* as a starting point of the optimization as described earlier. In our case, equations (xx) and (xx) is already a feasible basic form:

$$\begin{array}{rcl} \boldsymbol{A} & = & (I_{|E|}|J) \\ \boldsymbol{b}^T & = & (\ delay(e_0) \quad delay(e_1) \quad \cdots \quad delay(e_{|E|-1})\ ) \\ \boldsymbol{c}^T & = & (\ 0 \quad \cdots \quad 0 \quad f(v_0) \quad \cdots \quad f(v_{|V|-1})\ ), \end{array} \tag{4.42}$$

which derives a feasible basic solution

$$x_i = \begin{cases} \overline{delay}(e_i) = delay(e_i) & (i=0,1,\cdots|E|-1, e_i \in E) \\ r(v_{i-|E|}) = 0 & (i=|E|,|E|+1,\cdots,|E|+|V|-1, v_{i-|E|} \in V) \end{cases} \tag{4.43}$$

and therefore preprocessing for finding a feasible basic form is not required.

2. Simplex method is known to run in low order polynomial time for most cases (cases where it takes exponential time have been reported but rarely occurs in real problems [51]).

3. In general case, some modifications are needed on the simplex method to solve integer linear programming problem. However, if matrix $\boldsymbol{A}$ is *unimodular*, direct application of simplex method guarantees an optimal integer solution.

   **Matrix Unimodularity** A matrix $\boldsymbol{A}$ is said to be *unimodular* if $\boldsymbol{A}$ satisfies the following conditions [51]:
   (a) All elements of $\boldsymbol{A}$ is either 0, 1 or -1.
   (b) Determinant of any square submatrix of $\boldsymbol{A}$ is either 0, 1 or -1.

   It is known that unimodularity applies to our case.

**Multiple Fanout Hyper-Edge Modeling**

In the above discussion, only the pipeline network with two-point edges (one source, one destination) have been considered. Hyper-edges with multiple fanout have to be treated in a special way in order to model the register sharing among the fanout nodes [49][66]:

**Multiple fanout hyper-edge modeling** For each hyper-edge $e \in E_m$ with a fanout of $k$ ($k > 1$)

$$\begin{aligned} e &= (u, v_0, v_1, \cdots, v_{k-1}) \\ & \quad u : \text{source vertex} \\ & \quad \{v_i | i = 0, 1, \cdots, k-1\} : \text{destination vertices.} \end{aligned} \tag{4.44}$$

1. Divide $e$ into $k$ single fanout edges $\{e_i = (u, v_i) | i = 0, 1, \cdots, k-1\}$.
2. Assign a cost of $cost(e_i)/k$ to each edge $e_i$ ($i = 0, 1, \cdots, k-1$).
3. Assign a delay value to each edge as

$$\begin{aligned} delay(e_i) &= distance(v_i) - distance(u) - weight(e_i) \\ & \quad (i = 0, 1, \cdots, k-1). \end{aligned} \tag{4.45}$$

4. Add a dummy sink node $t$ and connect the $k$ fanout nodes $(e_0, e_1, \cdots, e_{k-1})$ to $t$ with $k$ additional dummy edges $(\bar{e}_i = (v_i, t)$, $i = 0, 1, \cdots, k-1)$. Assign a cost of $cost(e)/k$ and a weight of 0 to each of these $k$ edges. Consequently, the delay values of these dummy edges, assuming that the dummy sink node is ASAP scheduled, are

$$\begin{aligned} delay(\bar{e}_i) &= delay_{max}(e) - delay(e_i) \\ & \quad (i = 0, 1, \cdots, k-1) \end{aligned} \tag{4.46}$$

   where $delay_{max}(e) = \max(delay(e_i) | i = 0, 1, \cdots, k-1)$.

**Accuracy of the model** Because of the register sharing of the multiple fanout nodes, the number of registers needed on the hyper-edge $e$ to synchronize $v_0, v_1, \cdots, v_{k-1}$ is $C(e) = delay_{max}(e) \cdot cost(e)$. With the conversion method described above, total number of registers is calculated as

$$\begin{aligned} C'(e) &= \sum_{i=0}^{k-1} delay(e_i) \cdot cost(e)/k + \sum_{i=0}^{k-1} delay(\bar{e}_i) \cdot cost(e)/k \\ &= cost(e) \cdot \sum_{i=0}^{k-1} (delay(e_i) + delay(\bar{e}_i)) \\ &= cost(e) \cdot delay_{max}(e) \\ &= C(e). \end{aligned} \tag{4.47}$$

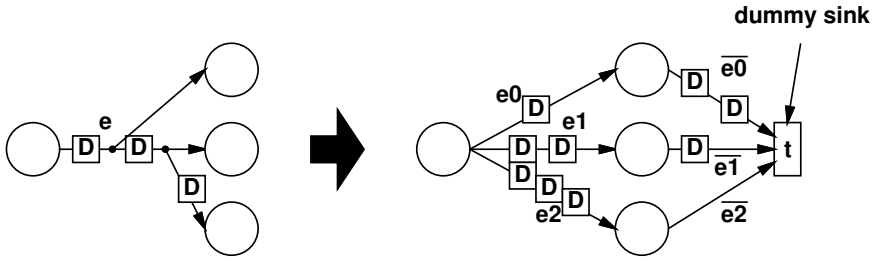



Figure 4.35: Conversions of hyper-edges into edges.

After retiming by synchronization optimization, the dummy node $t$ will be scheduled at the earliest possible time since it has no output edges and minimizing the register count for synchronizing $t$ leads to the earliest possible scheduling. Therefore, after the retiming, the new delay values for the dummy edges $(\bar{e}_0, \bar{e}_1, \cdots, \bar{e}_{k-1})$ is given as

$$\begin{aligned} \overline{delay}(\bar{e}_i) &= \overline{delay}_{max}(e) - \overline{delay}(e_i) \\ & (i = 0, 1, \cdots, k-1) \end{aligned} \tag{4.48}$$

where $\overline{delay}_{max}(e) = \max(delay(e_i)|i = 0, 1, \cdots, k-1)$. Therefore, the total number of registers after the retiming $\overline{C}'(e)$ is still the same as $\overline{C}(e)$:

$$\begin{aligned} \overline{C}'(e) &= \sum_{i=0}^{k-1} \overline{delay}(e_i) \cdot cost(e)/k + \sum_{i=0}^{k-1} \overline{delay}(\bar{e}_i) \cdot cost(e)/k \\ &= cost(e) \cdot \sum_{i=0}^{k-1} (\overline{delay}(e_i) + \overline{delay}(\bar{e}_i)) \\ &= cost(e) \cdot \overline{delay}_{max}(e) \\ &= \overline{C}(e). \end{aligned} \tag{4.49}$$

This multiple fanout hyper-edge modeling was first proposed by Leiserson and Saxe [66] and recognized to be a powerful tool for general pipeline synchronization problems by others [49, 58, 74].

## 4.15 IO Scheduling

Earlier, three types of IO interface were defined: system bus interface, RAM interface and ROM interface. The same data lines are used for input and output on system bus interface as well as RAM (ROM interface use separate input and output data lines since the input is connected to the address pins of the SRAM, and the output is connected to the data pins). For the static nature of the bit-serial pipeline network behavior where input data are fed into the system at a constant rate while the system produces the output data also at constant rate, we need to schedule these interface IOs to avoid read/write conflicts. Again, we assume that the behavior of the datapath circuit is static which repeats the same behavior every $T$ clock cycles where $T$ is the sampling period, and therefore we use static scheduling to avoid bus conflicts.

**IO scheduling for system bus interface and RAM interface** 1. Assume there are $M_{in}$ data inputs and $M_{out}$ data outputs in the datapath from and to the bus. These read and write transactions have to

occur within the sampling period $T$. Assuming that each read operation and write operation take one clock cycle, the following condition has to hold:

$$M_{in} + M_{out} \leq T. \tag{4.50}$$

If this condition cannot be met, it means that the bus bandwidth is not sufficient to sustain the computation on the datapath, and therefore either the sampling period $T$ has to increase or additional buses are needed. In the current implementation, if the above condition is not met, the pipeline synthesis program will indicate as such, but does not automatically increase the sampling period or add extra system buses. The designer is responsible for specifying the number of buses and the signal precision which determines the minimum sampling period (but not necessarily feasible if loop exists) and meet the above condition.

2. It is assumed that multiple read operations or multiple write operations are done in burst mode. Therefore, $M_{in}$ read operations occur consecutively in $M_{in}$ cycles, and $M_{out}$ write operations occur consecutively in $M_{out}$ cycles. This is partly imposed by our parallel-serial interface circuit design. Our circuit design is not yet flexible enough to accommodate multiple discrete read operations and write operations. This also needs to be addressed in the future.

3. Due to the above constraint, multiple $M_{in}$ read operations from the same bus is handled as one large operation. Graph representation of the operation is a vertex ($v_{in}$) having no inputs (therefore a primary input node) and $M_{in}$ output edges having different latencies. The first output edge has the latency of 1, and $i$-th output edge ($i = 1, 2, \cdots, M_{in}$) has the latency of $i$.

4. Likewise, multiple $M_{out}$ write operations to the same bus is handled as one large operation as well. Graph representation of the operation is a vertex($v_{out}$) having no outputs (therefore a primary output node) and $M_{out}$ input edges. Like other multiple input modules, these inputs have to be synchronized in order to implement the consecutive write operations.

5. Let $t(v_{in})$ and $t(v_{out})$ be the time the primary input vertex $v_{in}$ and the primary output vertex $v_{out}$ whom both share the same bus are scheduled after the pipeline synchronization optimization. Note that the pipeline synchronization optimization scheme described earlier does not take into account the bus conflict problem. Then the $M_{in}$ read operations occur between $t(v_{in})$ and $t(v_{in}) + M_{in}$. Similarly, the $M_{out}$ write operations occur between $t(v_{out})$ and $t(v_{out}) + M_{out}$. To avoid bus conflict, delay the primary output node $v_{out}$ by $r'(v_{out})$ so that

$$\begin{aligned} \mathrm{mod}(t(v_{in}) + i, T) \quad \neq \quad & \mathrm{mod}(t(v_{out}) + r'(v_{out}) + j, T) \\ & (i = 0, 1, \cdots, M_{in} - 1) \\ & (j = 0, 1, \cdots, M_{out} - 1). \end{aligned} \tag{4.51}$$

This condition guarantees that none of the read and write operations cause a bus conflict (Fig.4.36).

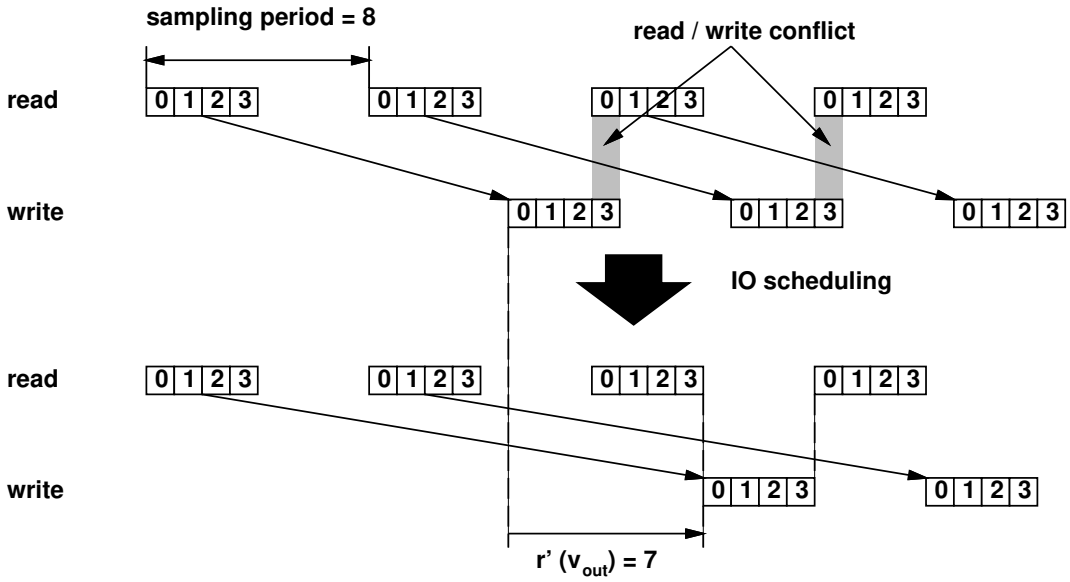



Figure 4.36: IO scheduling for resolving bus conflicts. The above shows a bus conflict at the first read operation and the last write operation. In order to avoid this read/write conflict, we have to delay the write operations by 7 clocks.

# Chapter 5

# Design Examples Using the Bit-Serial Datapath Synthesis System

The following applications were chosen to demonstrate the ability of the C++ bit-serial pipeline synthesis tool.

1. 1D FIR filter : This application is one of the most basic functions in digital signal processing.
2. 2D FIR filter : This is also a basic function in image processing. This application demonstrates the use of the line delays implemented on the external SRAM.
3. 1D IIR filter : This is also a basic DSP application. Because the algorithm contains loops, the throughput is limited to a certain value which may be considerably smaller than the maximum throughput allowed by the given signal precision. This application demonstrates the importance of algorithm selection where the throughput varies significantly amoung different algorithm implementations.
4. 1D adaptive filter : This application demonstrates yet another interesting trade-off between performance, hardware size, and the quality of the algorithm behavior.
5. Inverse discrete cosine transform : This application is often used in image and video compression. Two different algorithm implementation is compared where one implementation consumes 20 multipliers whereas the other consumes 14 multipliers. The interesting result is that the 20 multiplier implementation required less hardware because of the regularity in the datapath structure. This is the indication that the merely comparing the number of operations may lead to an inaccurate analysis of the quality of the algorithm implementation.
6. Digital neural network : This application is particularly interesting in terms of layout problems. The computations of this application presents a highly connected structure (a complete connection between layers) therefore considered to be very hard to partition and route. Because of the limitation of the current implementation of the C++ bit-serial synthesis tool, some of the operations implemented in the original work by Cox et al. [25] were not implemented in this study. However, this application was an ideal benchmark for demonstrating the efficiency of the bit-serial circuits in terms of FPGA layout.

## 5.1 Digital Filters

In this section, we present several design examples of one of the most basic operations in digital signal processing [43]. *Signal convolution* is an operation where the output signals are calculated from the weighted sum of the input signals:

$$y(t) = \sum_{k=0}^{m} x(t-k) \cdot a(k) \tag{5.1}$$

where $y(t)$ is the output signal at time $t$ and $x(t-k)$ is the input signal at time $t-k$. Linear shift-invariant systems, a time-discrete linear system where its characteristic does not change over time, are often characterized by the *impulse response* which is the output signal sequence corresponding to an impulse input. An impulse signal, written as $\delta(t)$, is a signal whose value is 1 if $t = 0$, and 0 otherwise. The impulse response of a convolution can then be calculated as

$$\begin{aligned} y(t) &= \sum_{k=0}^{m} \delta(t-k) \cdot a(k) \\ &= a(t). \end{aligned} \tag{5.2}$$

Therefore the convolution weights, which are also called filter coefficients, are the actual impulse response. In reality, the number of filter coefficients ($m$) are limited to a finite number, or else it will take an infinite amount of time to calculate even one output. Therefore, this signal convolution is also called a *finite impulse response* (FIR) filter. The above example is for one-dimensional (1-D) signals, where $t$ usually denotes the time. This FIR filter can be expanded into multidimensional signals. A two-dimensional (2-D) FIR filter is expressed as

$$y(i,j) = \sum_{k=0}^{m} \sum_{l=0}^{n} x(i-k, j-l) \cdot a(k,l). \tag{5.3}$$

This 2-D filter is also called the *mask filter* which is one of the basic operations in image processing.

There is another type of signal convolution where the output is calculated by the weighted sum of the input signals plus the weighted sum of the past output signals:

$$y(t) = \sum_{k=0}^{m} x(t-k) \cdot a(k) + \sum_{k=1}^{m} y(t-k) \cdot b(k) \tag{5.4}$$

The recursive nature of this system has a very different characteristic from the FIR filter. First, the impulse response continues to infinity because of the recursive calculation. Thus this recursive filter is also called an *infinite impulse response* (IIR) filter. Because of this long impulse response, IIR filter can have a sharper cutoff frequency response compared to FIR filter under the same filter order, i.e. the number of multiplications. However, IIR filter has a non-linear phase response, i.e. the latency of the signal output differs with the signal frequency. As a result, the waveform of the output signal tends to be distorted. IIR filters are popularly used in audio signal processing because of its sharp cutoff frequency response. Since the human ear captures the audio signals at the frequency domain and insensitive to the waveform itself, the non-linear phase response is not a significant problem. FIR filters are still dominantly used in image processing because the human vision is sensitive to the phase response and the linear phase response of the FIR filter is essential in preserving the basic waveform of the image data.

### 5.1.1 1-D FIR Filters

The C++ description of the 1-D FIR filter is already given in the section 4.4. Table 5.1 describes the design parameters, performance, and the size of the bit-serial pipeline circuits. Scalers are implemented with a unified

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 1D-FIR5-I | 5 | 354 | 6395 | 17.48 | 5.75 | 695 | 842 |
| 1D-FIR10-I | 10 | 684 | 12245 | 17.60 | 5.83 | 1340 | 1487 |
| 1D-FIR20-I | 20 | 1344 | 23737 | 17.66 | 5.87 | 2630 | 2777 |
| 1D-FIR30-I | 30 | 2004 | 35645 | 17.68 | 5.88 | 3920 | 4067 |
| 1D-FIR40-I | 40 | 2664 | 47345 | 17.69 | 5.89 | 5210 | 5357 |
| 1D-FIR50-I | 50 | 3324 | 59045 | 17.70 | 5.89 | 6500 | 6647 |
| 1D-FIR100-I | 100 | 6624 | 117337 | 17.71 | 5.90 | 12950 | 13097 |

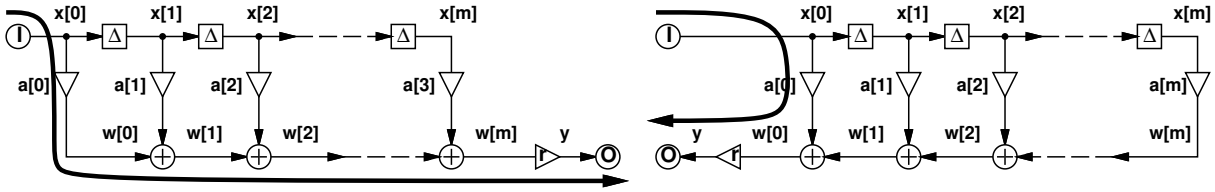
| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 1D-FIR5-I | (5, 4) | 22.5 | 3518 | 16 | 54 | 2.5 MHz |
| 1D-FIR10-I | (10,9) | 47.5 | 3879 | 16 | 59 | 2.5 MHz |
| 1D-FIR20-I | (20,19) | 97.5 | 4108 | 16 | 69 | 2.5 MHz |
| 1D-FIR30-I | (30,29) | 147.5 | 4138 | 16 | 79 | 2.5 MHz |
| 1D-FIR40-I | (40,39) | 197.5 | 4172 | 16 | 89 | 2.5 MHz |
| 1D-FIR50-I | (50,49) | 247.5 | 4192 | 16 | 99 | 2.5 MHz |
| 1D-FIR100-I | (100,99) | 497.5 | 4240 | 16 | 149 | 2.5 MHz |

Table 5.1: Type-I 1D FIR filter designs. "# taps" denotes the filter order $m$. "# gates" denotes the equivalent gate count of the datapath hardware which is counted by two-input gates. "gates/CLB" denotes the average gate count per CLB, and "gates/LUT" denotes the average gate count of the logic implemented in one LUT. "# FF" denotes the number of flip-flops used in the CLBs, flip-flops on the IOBs are excluded. "(# mult., # add)" denotes the number of multiplications and number of additions implemented on the datapath. "MOPS @ 40MHz" denote the number of integer operations per second at 40MHz clock. "OPS/gate" denote the number of integer operations per second divided by the gate count. "period" denotes the data sampling period in terms of clock cycles. "frequency" denotes the data sampling frequency at 40MHz clock. Signal word length is 16 bits.

cell structure whose scaling factor (multiplication coefficient) can be programmed after the physical placement and routing are fixed (refer to Chapter 3).

MOPS are calculated by the number of multiplications and additions performed on the datapath per second at 40 MHz clock. This clock rate, as we will see later in the layout results, is feasible in most cases. Number of gates are measured by the number of two-input gates in the logic equations implemented by the CLBs:

- NOT is not counted as gates.
- AND and OR is each counted as 1 gate.
- EXOR is counted as 3 gates.
- A Flip-flop is counted as 6 gates.




(a) Type–I FIR filter.

(b) Type–II FIR filter.

Figure 5.1: Two types of 1D FIR filters. While type-I has a pipeline latency proportional to the filter size, type-II has a constant latency regardless of the filter size.

From these gate calculations, we can evaluate how many "gates" are implemented in each LUT table and each CLB. Xilinx have evaluated they gate capacity of their XC3000 series FPGAs assuming 30 gates per CLB. The lower two digits of their product number, (3090, 3064, 3042, 3030, 3020) correspond to the maximum gate capacity of each chip, multiplied by 100 (3090: 9000 gates, 3064: 6400 gates, 3042: 4200 gates, 3030: 3000 gates, 3020: 2000 gates). However, in their recent databook [97], they have reduced the estimated gate capacity by 40 to 50%, perhaps after observing that the older gate capacity estimated were often not satisfied in real designs. 30 gates per CLB can be derived by dividing the old gate capacity number by the number of CLBs for each device.

1. XC3090 : 9000 gates, 320 CLBs → 28.125 gates/CLB
2. XC3064 : 6400 gates, 224 CLBs → 28.571 gates/CLB
3. XC3042 : 4200 gates, 144 CLBs → 29.167 gates/CLB
4. XC3030 : 3000 gates, 100 CLBs → 30.000 gates/CLB
5. XC3020 : 2000 gates, 64 CLBs → 31.25 gates/CLB

The true device utilization is not only governed by the number of CLBs used, but also the number of gates implemented on the CLB. In the case of 1D FIR filter, we see that the "gate utilization" of each CLB does not quite reach 30. Therefore, even if the logic resource utilization reaches 100%, as in many cases on bit-serial circuits, the effective gate count implemented on FPGA can be quite low.

One of the major components in this 1D FIR filter and also in other filters in general is the sampling delay element. In bit-serial pipeline architecture, these sampling delays are automatically converted into shift-registers with guaranteed pipeline synchronism. Recall from Section 3.2.4 and 3.2.5 that these shift-registers are available inside the scaler modules. The bit-serial synthesis program automatically detects these hidden register resources and uses to synchronize the pipeline.

The circuit size grows proportionally to the filter size while the data throughput remains unchanged. However, the latency grows proportionally to the filter size. This is caused by the pipelined accumulation latency (Fig.5.1(a)). By changing the direction of the accumulation, we are able to make the latency to be a constant value. Fig.5.1(b) shows this filter structure. The C++ description of this filter is given in the Appendix (Fig.D.1). Table 5.2 shows the synthesis results of the second FIR structure. Fig.5.3 shows the comparison of latency values between the two types of filter. The designer can easily try out different FIR designs using our C++ design capture system and instantaneously get the performance measure and the hardware estimate for the evaluation of the algorithm implementation.

### 5.1.2 2-D FIR filters

Fig.5.4 describe the SFG of the 2-D FIR filter. The corresponding C++ description can be found in the appendix (Fig.D.2). The function `delay()` denotes the *pixel delay* and the function `line_delay()` denotes the *line delay*. A pixel delay is used to access the pixel data which is to the left of the current pixel location. A line delay is used to access the pixel data which is to the upper of the current pixel location. If the pixel data are fed into the system with a raster scan format, the pixel delay essentially becomes the sampling delay and the line delay is implemented by cascading `MAX2` sampling delays where `MAX2` is the number of pixels per scan line. In our datapath synthesis system, the line delay is implemented by an SRAM and the memory control circuit is automatically generated. Up to 7 line delays are implemented in one SRAM when the signal precision is 8 bits since there are 8 clock cycles per sampling period and one clock is needed to write the data to the SRAM, which leaves 7 clock cycles to read in the data from the SRAM. Synthesis results of different filter orders are given in Table 5.3. Here, the clock speed is assumed to be 40 MHz. However, since the off-chip SRAM is used, the clock rate may need to decrease, depending on the hardware setting and the electrical parasitics at the MCM-level and board-level.

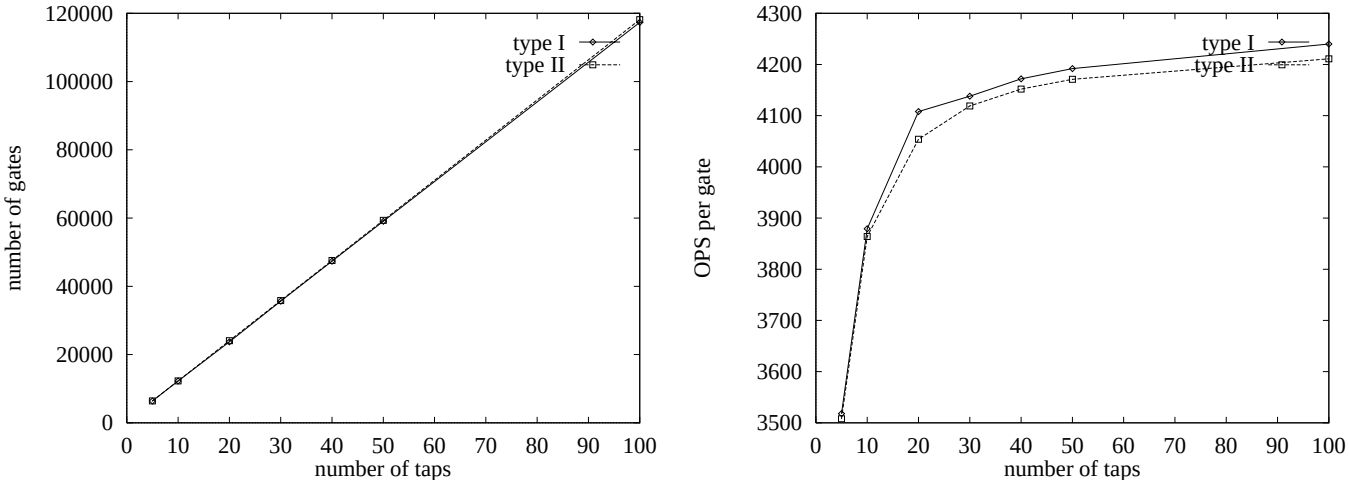



Figure 5.2: 1D FIR filter synthesis results for the two filter structures. X-axis is the number of filter taps and y-axes are gate count (left), and OPS per gate (right).

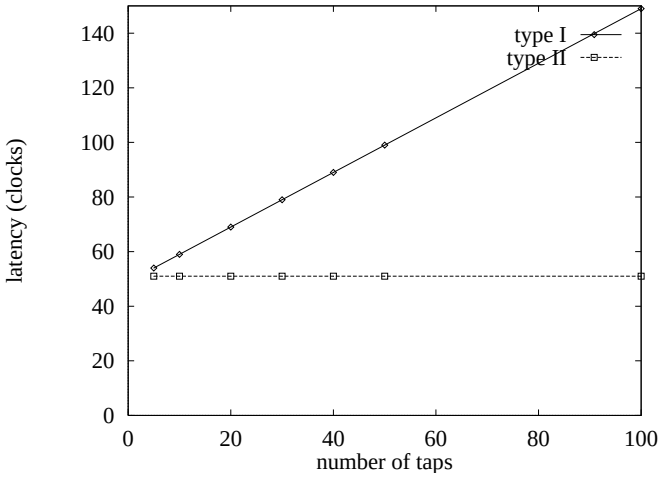



Figure 5.3: Latency comparison between the two types of 1D FIR filter designs. The latency of type-I scales linearly with the filter size, whereas the latency of type-II remains the same.

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 1D-FIR5-II | 5 | 354 | 6413 | 17.53 | 5.75 | 698 | 845 |
| 1D-FIR10-II | 10 | 684 | 12293 | 17.67 | 5.83 | 1348 | 1495 |
| 1D-FIR20-II | 20 | 1344 | 24053 | 17.74 | 5.87 | 2648 | 2795 |
| 1D-FIR30-II | 30 | 2004 | 35813 | 17.78 | 5.88 | 3948 | 4095 |
| 1D-FIR40-II | 40 | 2664 | 47573 | 17.78 | 5.89 | 5248 | 5395 |
| 1D-FIR50-II | 50 | 3324 | 59333 | 17.79 | 5.89 | 6548 | 6647 |
| 1D-FIR100-II | 100 | 6624 | 118133 | 17.80 | 5.90 | 13048 | 13195 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 1D-FIR5-II | (5,4) | 22.5 | 3508 | 16 | 51 | 2.5 MHz |
| 1D-FIR10-II | (10,9) | 47.5 | 3864 | 16 | 51 | 2.5 MHz |
| 1D-FIR20-II | (20,19) | 97.5 | 4054 | 16 | 51 | 2.5 MHz |
| 1D-FIR30-II | (30,29) | 147.5 | 4119 | 16 | 51 | 2.5 MHz |
| 1D-FIR40-II | (40,39) | 197.5 | 4152 | 16 | 51 | 2.5 MHz |
| 1D-FIR50-II | (50,49) | 247.5 | 4171 | 16 | 51 | 2.5 MHz |
| 1D-FIR100-II | (100,99) | 497.5 | 4211 | 16 | 51 | 2.5 MHz |

Table 5.2: Type-II 1D FIR filter designs. Compared to type-I, the latency remains constant regardless of the filter size. Signal word length is 16 bits.

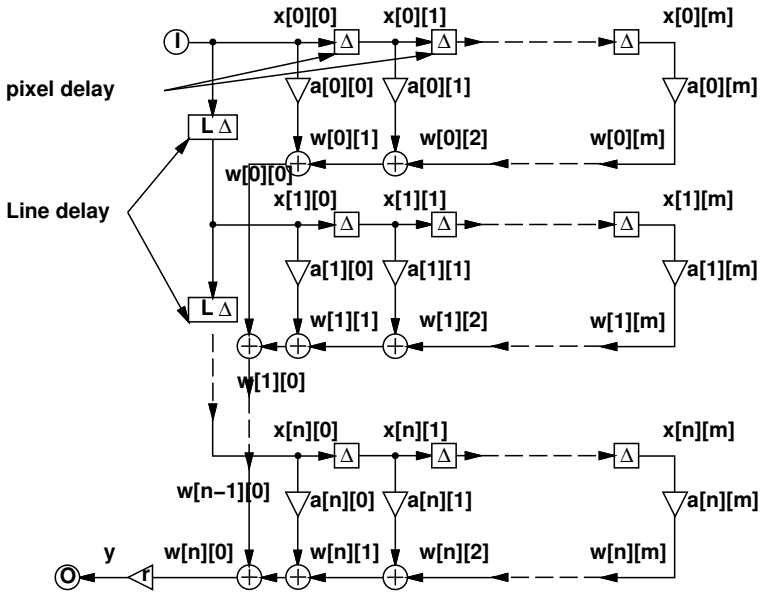



Figure 5.4: 2D FIR filter

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 2D-FIR3x3 | 9 | 374 | 6527 | 16.97 | 5.384 | 707 | 856 |
| 2D-FIR4x4 | 16 | 620 | 10837 | 17.19 | 5.52 | 1180 | 1329 |
| 2D-FIR5x5 | 25 | 930 | 16303 | 17.33 | 5.60 | 1781 | 1930 |
| 2D-FIR6x6 | 36 | 1312 | 23025 | 17.41 | 5.65 | 2518 | 2667 |
| 2D-FIR7x7 | 49 | 1758 | 30891 | 17.47 | 5.69 | 3383 | 3532 |
| 2D-FIR8x8 | 64 | 2272 | 39957 | 17.51 | 5.71 | 4380 | 4529 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 2D-FIR3x3 | (9,8) | 85.0 | 13023 | 8 | 30 | 5.0 MHz |
| 2D-FIR4x4 | (16,15) | 155.0 | 14303 | 8 | 31 | 5.0 MHz |
| 2D-FIR5x5 | (25,24) | 245.0 | 15028 | 8 | 32 | 5.0 MHz |
| 2D-FIR6x6 | (36,35) | 355.0 | 15418 | 8 | 33 | 5.0 MHz |
| 2D-FIR7x7 | (49,48) | 485.0 | 15700 | 8 | 34 | 5.0 MHz |
| 2D-FIR8x8 | (64,63) | 635.0 | 15892 | 8 | 35 | 5.0 MHz |

Table 5.3: 2D FIR filter designs. Signal word length is 8 bits.

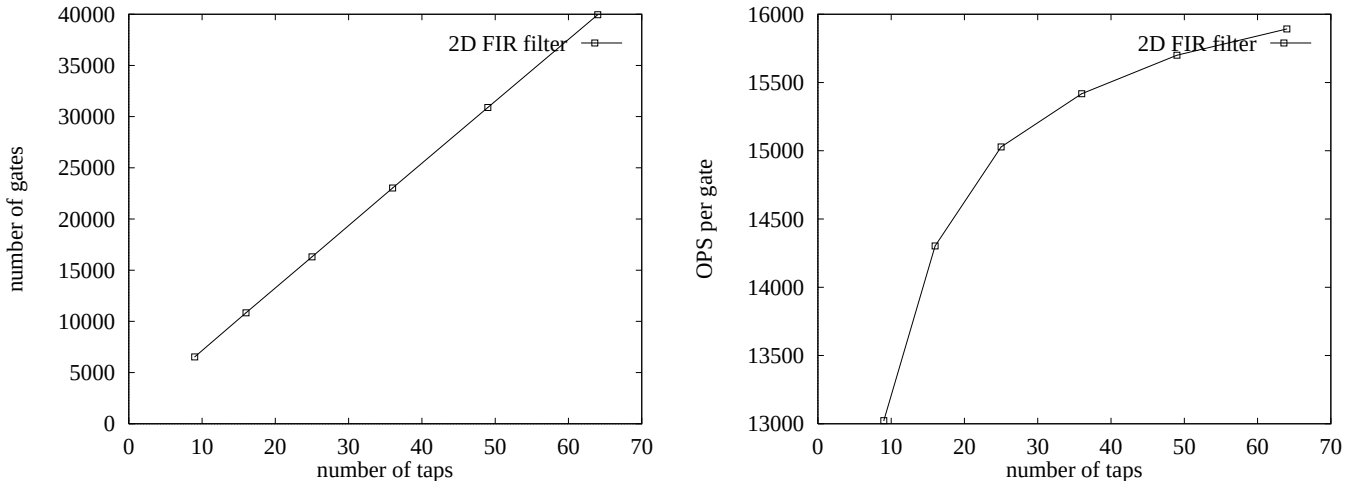
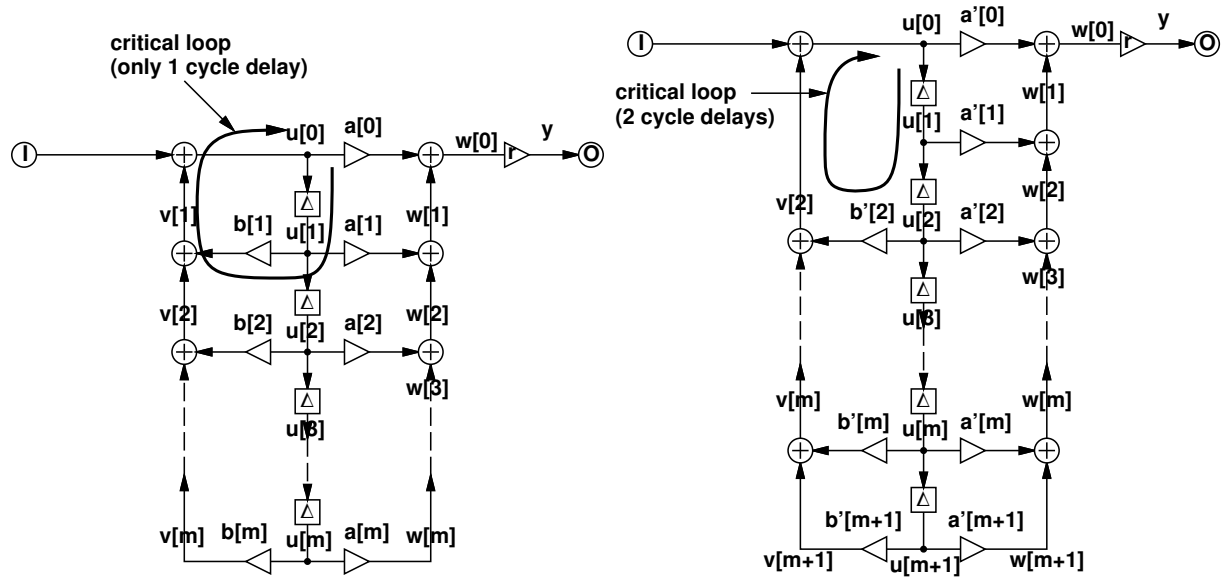



Figure 5.5: 2D FIR filter synthesis results. Signal word length is 8 bits.




(a) Direct IIR filter (type–I).

(b) Lookahead IIR filter (type – II).

Figure 5.6: 1D IIR filters. In (a), there is a loop whose cycle delay is one, indicating that the operations along the loop (one scaling, two addition) are required to complete within one sampling period. In (b), the cycle delay of the loop is two, which indicates that the operations along the loop are required to complete within two sampling periods. Therefore, the latter implementation has a shorter feasible minimum sampling period.

### 5.1.3 1-D Recursive Filters

One type of implementation of a 1-D IIR filter is illustrated in Fig.5.6(a) and the corresponding C++ code is shown in the appendix (Fig.D.3. The synthesis results of this IIR filter structure is given in Table 5.4. Here, the bit-serial scalers are also implemented using the unified cell structure. As we can see here, the sampling period is more than twice as long as the signal word length which is 16 bits. This is because there is a critical loop in the datapath which includes one scaler whose latency is $2N = 32$ clock cycles, two adders and a rounder, each of which having one clock cycle latency, with a total of 35 clock cycle latency. Since there are only one sampling delay along this loop (this is called a *cycle delay*), all these computations have to be completed in one sampling cycle. If the filter coefficients are fixed, we can use custom cell structure for the scaler modules which not only decreases the hardware size, but also decreases the latency and therefore increasing the throughput. Another approach in increasing the throughput is by unfolding the recursion to relax the critical loop computation requirement. The

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 1D-IIR5-I | 5 | 558 | 10379 | 18.26 | 5.81 | 1096 | 1115 |
| 1D-IIR6-I | 6 | 664 | 12419 | 18.41 | 5.82 | 1304 | 1323 |
| 1D-IIR8-I | 8 | 876 | 16499 | 18.62 | 5.85 | 1720 | 1739 |
| 1D-IIR10-I | 10 | 1088 | 20579 | 18.74 | 5.86 | 2136 | 2155 |
| 1D-IIR12-I | 12 | 1300 | 24659 | 18.82 | 5.88 | 2552 | 2571 |
| 1D-IIR15-I | 15 | 1618 | 30779 | 18.90 | 5.88 | 3176 | 3195 |
| 1D-IIR20-I | 20 | 2148 | 40787 | 18.99 | 5.89 | 4216 | 4235 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 1D-IIR5-I | (9,8) | 19.4 | 1872 | 35 | 53 | 1.14MHz |
| 1D-IIR6-I | (11,10) | 24.0 | 1933 | 35 | 53 | 1.14MHz |
| 1D-IIR8-I | (15,14) | 33.1 | 2009 | 35 | 53 | 1.14MHz |
| 1D-IIR10-I | (19,18) | 42.3 | 2055 | 35 | 53 | 1.14MHz |
| 1D-IIR12-I | (23,22) | 51.4 | 2086 | 35 | 53 | 1.14MHz |
| 1D-IIR15-I | (29,28) | 65.1 | 2117 | 35 | 53 | 1.14MHz |
| 1D-IIR20-I | (39,38) | 88.0 | 2147 | 35 | 53 | 1.14MHz |

Table 5.4: Type-I 1D IIR filter designs. Due to the feedback loop having a single cycle delay, the sampling period is more than twice as long as the signal word length. Signal word length is 16 bits.

recursion of eq.(5.4) can be unfolded one time as follows:

$$\begin{aligned} y(t) &= \sum_{k=0}^{m} x(t-k)\cdot a(k) + \sum_{k=2}^{m} y(t-k)\cdot b(k) \\ &+ y(t-1)\cdot b(1) \\ &= \sum_{k=0}^{m} x(t-k)\cdot a(k) + \sum_{k=2}^{m} y(t-k)\cdot b(k) \\ &+ b(1)\cdot(\sum_{k=0}^{m} x(t-k-1)\cdot a(k) + \sum_{k=1}^{m} y(t-k-1)\cdot b(k)) \\ &= \sum_{k=0}^{m+1} x(t-k)\cdot a'(k) + \sum_{k=2}^{m+1} y(t-k)\cdot b'(k) \end{aligned} \tag{5.5}$$

where

$$\begin{aligned} a'(k) &= \begin{cases} a(k) & (k=0) \\ b(1)\cdot a(k-1) & (k=m+1) \\ a(k)+b(1)\cdot a(k-1) & (\text{otherwise}) \end{cases} \\ b'(k) &= \begin{cases} b(1)\cdot b(k-1) & (k=m+1) \\ b(k)+b(1)\cdot b(k-1) & (\text{otherwise}). \end{cases} \end{aligned} \tag{5.6}$$

This recursion transformation is possible for any type of recursive structure and is often a very effective method in increasing the throughput of the feedback system. The new filter structure have the critical loop which has two cycle delays instead of one. Therefore, the 35 latency cycles can be dispersed into two sampling periods. This recursive filter has a sampling period of $\lceil 35/2 \rceil = 18$ clock cycles. It has one extra scaler and one adder, but the throughput is effectively doubled. This *recursion lookahead* technique can be applied a multiple of times as well. By unfolding the recursion two times, the sampling period becomes the same as the signal word size which is 16. The synthesis results of the 1-lookahead IIR filters are shown in Table 5.5 and the 2-lookahead IIR filters are shown in Table 5.6. From the synthesis results, we can see that the hardware does not grow as much for lookahead filters even though the number of scalers and adders grow. This is because we need less number of pipeline synchronization registers as the sampling period decreases.

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 1D-IIR5-II | 5 | 602 | 11285 | 18.43 | 5.82 | 1181 | 1200 |
| 1D-IIR6-II | 6 | 704 | 13277 | 18.59 | 5.83 | 1381 | 1400 |
| 1D-IIR8-II | 8 | 908 | 17261 | 18.80 | 5.85 | 1781 | 1800 |
| 1D-IIR10-II | 10 | 1112 | 21245 | 18.93 | 5.86 | 2181 | 2200 |
| 1D-IIR12-II | 12 | 1316 | 25229 | 19.03 | 5.87 | 2581 | 2600 |
| 1D-IIR15-II | 15 | 1622 | 31205 | 19.12 | 5.88 | 3181 | 3200 |
| 1D-IIR20-II | 20 | 2132 | 41165 | 19.22 | 5.89 | 4181 | 4200 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 1D-IIR5-II | (10,9) | 42.2 | 3741 | 18 | 53 | 2.22MHz |
| 1D-IIR6-II | (12,11) | 51.1 | 3850 | 18 | 53 | 2.22MHz |
| 1D-IIR8-II | (16,15) | 68.9 | 3991 | 18 | 53 | 2.22MHz |
| 1D-IIR10-II | (20,19) | 86.7 | 4079 | 18 | 53 | 2.22MHz |
| 1D-IIR12-II | (24,23) | 104.4 | 4140 | 18 | 53 | 2.22MHz |
| 1D-IIR15-II | (30,29) | 131.1 | 4201 | 18 | 53 | 2.22MHz |
| 1D-IIR20-II | (40,39) | 175.6 | 4265 | 18 | 53 | 2.22MHz |

Table 5.5: Type-II lookahead 1D IIR filter designs. Since the feedback is unroled one sampling cycle, the sampling period becomes sufficiently close to the signal word length. Signal word length is 16 bits.

With our bit-serial synthesis system, the designer does not need to worry about the pipeline synchronization issues since this is taken care of by our synthesis tool. The designer is thus able to simply change the structure of the filter with extreme ease by C++ coding, and immediately is able to see the tradeoffs on hardware size and performance for those different filter structures.

### 5.1.4 Comparison With Other Filter Designs on FPGAs

Several groups have reported their digital filter implementations over the years [84][54][93][65].

#### 1D FIR Filters

Isoaho et al. [54] has implement digital filters using bit-serial adders and shifters. They have limited the filter coefficients to be only two bits. Their data shows that up to 25 taps with 16-bit word can be implemented using one XC3090 which has 320 CLBs. Comparison with our implementation is shown in Table 5.7. In our implementation, all the coefficients are set to 0.75 to match they coefficient precision of 2 bits. As we can see here, their design is more compact then ours. This is because they only use a single line bit-serial data for internal signals which needs more precision than the external signals. They have increased the sampling period in order to accommodate longer words. On the other hand, we use dual lines for the internal word which does not increase the sampling period. And therefore we need twice as much hardware to implement dual line bit-serial circuits for exchange for speed.

#### 1D IIR Filters

Wahab et al. [93] has implement digital filters using a combination of bit-parallel ALU and bit-serial transfer function block. The performance of their filter implementation decreases as the filter order increase. Their architecture does not allow the hardware to completely absorb the filter order increase in order to maintain constant performance such as in our case. They report a second order IIR filter with 8-bit signals using 69 CLBs which have the sampling frequency of little less than 500kHz, and a 16-bit 15th order IIR filter using 317 CLBs where the sampling frequency is less than 50kHz. With our bit-serial implementation, a second order IIR filter with 8-bit word requires 156 CLBs and has a sampling frequency of 4.4MHz, and a 15th order IIR filter with 16-bit word

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| 1D-IIR5-III | 5 | 667 | 12437 | 18.36 | 5.82 | 1307 | 1326 |
| 1D-IIR6-III | 6 | 769 | 14429 | 18.51 | 5.84 | 1507 | 1526 |
| 1D-IIR8-III | 8 | 973 | 18413 | 18.73 | 5.85 | 1907 | 1926 |
| 1D-IIR10-III | 10 | 1177 | 22397 | 18.87 | 5.87 | 2307 | 2326 |
| 1D-IIR12-III | 12 | 1381 | 26381 | 18.96 | 5.87 | 2707 | 2726 |
| 1D-IIR15-III | 15 | 1687 | 32357 | 19.07 | 5.88 | 3307 | 3326 |
| 1D-IIR20-III | 20 | 2197 | 42317 | 19.17 | 5.89 | 4307 | 4326 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| 1D-IIR5-III | (11,10) | 52.5 | 4221 | 16 | 53 | 2.5MHz |
| 1D-IIR6-III | (13,12) | 62.5 | 4332 | 16 | 53 | 2.5MHz |
| 1D-IIR8-III | (17,16) | 82.5 | 4481 | 16 | 53 | 2.5MHz |
| 1D-IIR10-III | (21,20) | 102.5 | 4577 | 16 | 53 | 2.5MHz |
| 1D-IIR12-III | (25,24) | 122.5 | 4644 | 16 | 53 | 2.5MHz |
| 1D-IIR15-III | (31,30) | 152.5 | 4713 | 16 | 53 | 2.5MHz |
| 1D-IIR20-III | (41,40) | 102.5 | 4785 | 16 | 53 | 2.5MHz |

Table 5.6: Type-III lookahead 1D IIR filter designs. Since the feedback is unroled two sampling cycles, the sampling period becomes the same as the signal word length. Signal word length is 16 bits.

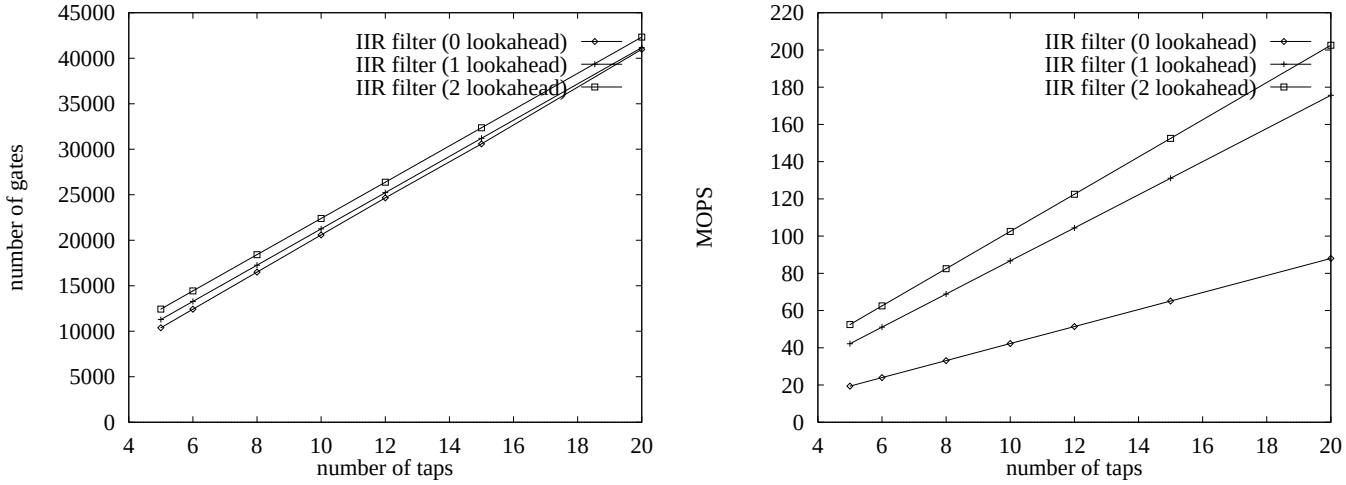



Figure 5.7: 1D IIR filter synthesis results for the three filter structures. The left graph shows the gate count for different filter orders, and the right graph shows the number of operations per second.

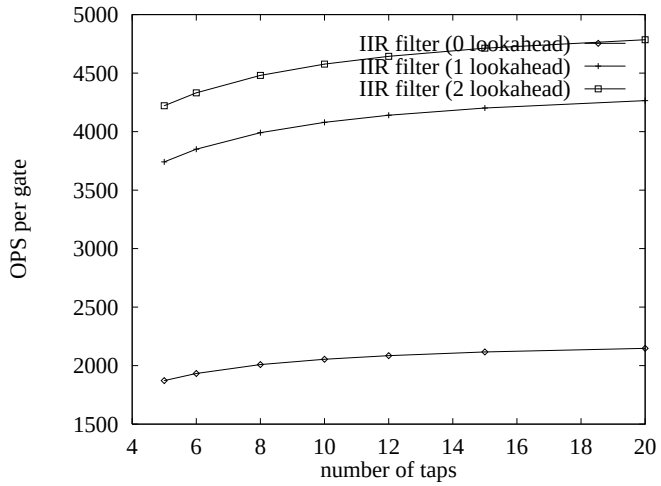



Figure 5.8: 1D IIR filter synthesis results for the three filter structures (OPS/gate).

| Filter | external word size | internal word size | max # of taps on XC3090 | gates/CLB | sampling period |
|---|---|---|---|---|---|
| Ours | 8 bits | 16 bits | 19 | 14.28 | 8 cycles |
| | 12 bits | 24 bits | 15 | 13.87 | 12 cycles |
| | 16 bits | 32 bits | 12 | 13.61 | 16 cycles |
| Ioaho et al. | 8 bits | 13 bits | 35-36 | 12.8 | 13 cycles |
| | 12 bits | 17 bits | 29-31 | 12.8 | 17 cycles |
| | 16 bits | 21 bits | 23-25 | 12.8 | 21 cycles |

Table 5.7: Comparison between our FIR filter implementation and the implementation by Isoaho et al. (1993)

| Filter | word size ($w$) | filter order ($n$) | # CLBs ($c$) | sampling frequency ($f$) | circuit efficiency ($e = w \cdot n \cdot f/c$) |
|---|---|---|---|---|---|
| Bit-serial | 8 bits | 2 | 156 | 4.4MHz | $4.51 \times 10^5$ |
| | 16 bits | 15 | 1730 | 2.2MHz | $3.05 \times 10^5$ |
| Wahab et al. | 8 bits | 2 | 69 | $\sim$ 500kHz | $1.16 \times 10^5$ |
| | 16 bits | 15 | 317 | $\sim$ 50kHz | $0.38 \times 10^5$ |

Table 5.8: Comparison between our IIR filter implementation and the implementation by Wahab et al. (1993). "Circuit efficiency" is a number which is derived by multiplying the signal word length, filter order and sampling frequency, and further dividing it with the number of CLBs.

| FPGA | # CLBs | # IOs | FF-FF | pad-FF | FF-pad |
|---|---|---|---|---|---|
| FPGA1 | 214 | 34 | 12.8ns | 9.7ns | 8.5ns |
| FPGA2 | 164 | 33 | 10.9ns | 9.7ns | 9.8ns |

Table 5.9: Bit-serial implementation of a fifth-order wave elliptic filter. The circuit functions on a 50MHz clock assuming that the there are no extra delays on MCM-level or board-level. XC3164A-3 is used. "FF-FF" denotes the maximum delay from the flip-flop output to the flip-flop input. "pad-FF" denotes the maximum delay from the input pad to the FF input. "FF-pad" denotes the maximum delay from the FF output to the output pad.

requires 1730 CLBs and has the sampling frequency of 2.2MHz (Table 5.8). In terms of circuit efficiency which we calculate here as the product of sampling frequency, word size and filter order divided by the number of CLBs, we have an advantage by a factor of 3.89 over their implementation of 8-bit second order IIR filter and 16.12 over their implementation of 16-bit 15th order IIR filter.

**Wave Elliptic Filter**

Leeser et al. designed a fifth-order wave elliptic filter using their *Bedroc* high-level synthesis system. The filter structure is illustrated in Fig.5.9. This filter architecture is derived from the analogue filter design methodology. The advantage of this filter structure is that the filter coefficients can be shorter compared to other filter structures. Due to the long critical loop, this filter is commonly used in evaluating the performance of the high-level synthesis tools. They have compared their synthesized design with a direct implementation using bit-parallel circuits. We have implemented the 5th order elliptic filter with the very same specifications they used: 6-bit signal word length, 3-bit coefficients. We have mapped the circuits onto two XC3164A Xilinx FPGAs (Table 5.9). The routing results show that our bit-serial circuit is functional at 50MHz clock.

It is hard to to a direct comparison since they used Altera MAX5000 series FPLD which has a different routing architecture similar to PLDs, but a fairly similar LUT-based logic block architecture. Here we simply compare by the estimated gate counts versus performance (Table 5.10). Here, we see a huge advantage of our bit-serial synthesis system compared to their Bedroc system by a factor of 73 in terms of circuit efficiency. Our design is only one third the size in terms of estimated gate counts compared to their synthesized design, and is 25 times faster. This illustrates the large degree of inefficiency introduced by the high-level synthesis approach. Also, compared to their direct implementation using bit-parallel circuits, our design is 52 times more efficient. Our design is 32% smaller and 38.4 times faster. This comparison strongly backs up our claim in Chapter 2 about the superior efficiency compared to bit-parallel circuits.

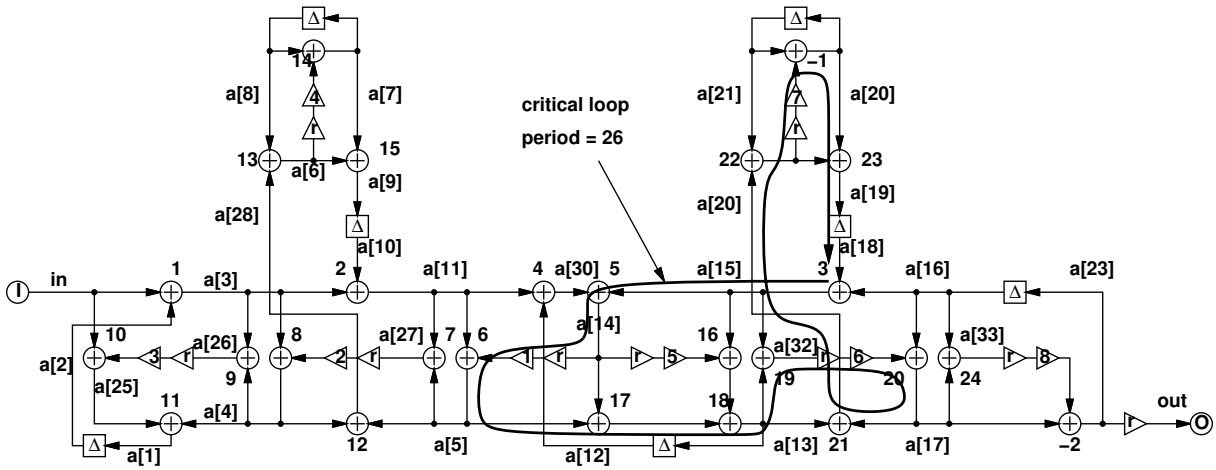
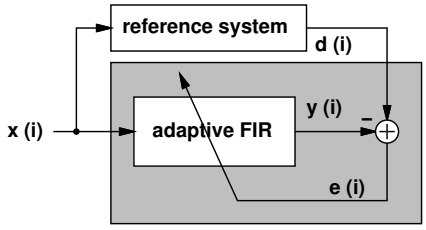


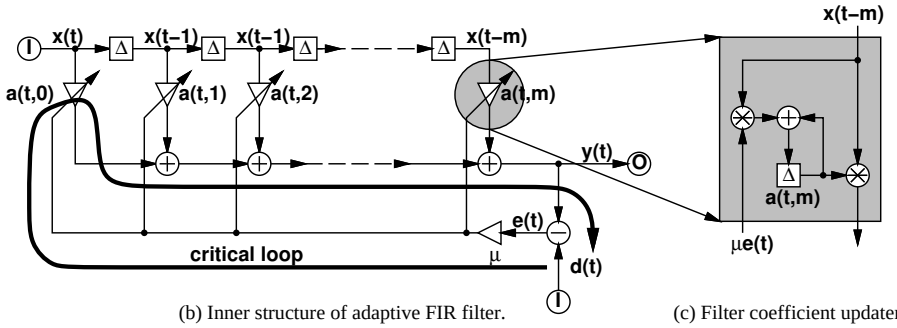

Figure 5.9: 5th order wave elliptic filter. This filter architecture is derived from the analogue filter design methodology. The advantage of this filter structure is that the filter coefficients can be shorter compared to other filter structures. Due to the long critical loop, this filter is commonly used in evaluating the performance of the high-level synthesis tools.

| Implementation | gate count ($g$) | clock frequency | sampling period | sampling frequency ($f$) | efficiency ($f/g$) |
|---|---|---|---|---|---|
| Bit-serial pipeline | 5312 | 50.0MHz | 26 | 1.92 MHz | 361.4 |
| Direct bit-parallel (Leeser) | 7820 | 50KHz | 1 | 50KHz | 6.93 |
| High-level Bedroc (Leeser) | 15520 | 1.54MHz | 20 | 77KHz | 4.96 |

Table 5.10: Comparison of different implementations of fifth-order wave elliptic filter between our synthesis system and by Leeser et al. (1993). Signal word size is 6 bits and coefficients are 3 bits or less.




(a) Diagram of adaptive FIR filter.




(b) Inner structure of adaptive FIR filter.

(c) Filter coefficient updater.

Figure 5.10: Adaptive filter structure.

#### 2D FIR Filters

Shoup [84] designed a 3x3 2D FIR filter on CLi6000 FPGAs from Amtel. He also uses bit-serial multipliers and adders, and implements the line delay with an external memory. Their 8-bit multiplier requires 16 cycles to compute the product whereas we only need 8 cycles. Therefore, their design is simpler and more compact than ours, but with the cost of lower performance. In terms of efficiency, our design and Shoup's is nearly the same, since we are using the same architecture. Hardware size and performance figures for Shoup's design was not available.

## 5.2 Adaptive Filter

In the previous section, we presented several digital filter designs where the filter coefficients are known in advance, and therefore can be implemented with one-input scalers. One of the basic forms of *adaptive filters* have the same FIR or IIR filter structure, or other filter structures such as lattice filter, but the filter coefficients are updated by the outcome of the past output signals. Fig.5.10 illustrates the basic structure of the adaptive FIR filter. The adaptive FIR filter has two signal inputs: an input signal $x(t)$ and a reference signal $d(t)$. The reference signal $d(t)$ is assumed to be generated from $x(t)$ by some linear shift-invariant system $H(z)$, however, this reference system may change its characteristics over time. The adaptive FIR filter tries to get closer to $H(z)$ by observing the error $(e(t))$ between the current output signal and the reference signal

$$e(t) = d(t) - y(t), \tag{5.7}$$

and updating the filter coefficients as

$$a(t+1,k) = \mu \cdot e(t) \cdot x(t-k) + a(t,k) \tag{5.8}$$

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| adapt5 | 5 | 712 | 11400 | 15.88 | 6.40 | 1382 | 1393 |
| adapt10 | 10 | 1363 | 21916 | 16.01 | 6.52 | 2663 | 2674 |
| adapt15 | 15 | 2097 | 33200 | 15.77 | 6.57 | 4072 | 4083 |
| adapt20 | 20 | 2846 | 45018 | 15.78 | 6.59 | 5570 | 5581 |

| design | (# mult., # add) | MOPS @ 40MHz | OPS/gate | period | frequency |
|---|---|---|---|---|---|
| adapt5 | (10,11) | 16.5 | 1445 | 51 | 0.78MHz |
| adapt10 | (20,21) | 29.3 | 1336 | 56 | 0.71MHz |
| adapt15 | (30,31) | 40.0 | 1205 | 61 | 0.66MHz |
| adapt20 | (40,41) | 49.1 | 1091 | 66 | 0.61MHz |

Table 5.11: 1D adaptive filter synthesis results. Signal word length is 8 bits.

where $\mu$ is a constant which determines the speed of adaptation. If $\mu$ is small, the adaptation is slow, but if it is too large, adaptation fails and starts oscillating into an unstable state. Note that since the filter coefficients change in time, they are treated as signals and therefore have the time index as well. Each coefficient is updated by incrementing the coefficient value by the product of the weighted error and the corresponding delayed input signal. The output is still calculated as

$$y(t) = \sum_{k=0}^{m} x(t-k) \cdot a(t,k). \tag{5.9}$$

This computation structure forms a large loop which greatly limits the sampling period. Unfortunately, the recursion unrolling techniques used in the IIR filter example cannot be used here because this greatly increases the number of computations. Synthesis results are described in Table 5.11 and Fig.5.11, 5.12. Unlike other applications we have shown earlier, the throughput decreases as the filter size grows. This leads to the decrease in efficiency as well (OPS/gate). One of the weakness of the bit-serial pipeline architecture is the inability to produce efficient designs for applications which involves large amount of recursive calculations with a very tight loop calculation period. The C++ code description can be found in the appendix (Fig.D.4).

However, the designer's job is to somehow overcome this weakness of the bit-serial architecture by being *creative* in the algorithm implementation. The C++ design capture system enables the designer to do just that and quickly verify the new ideas using our behavioral simulator. What kind of new idea can we apply on this case?

Recall that the filter coefficients are updated by the error signal of the previously processed signal (eq.(5.8)). This only gives us one sampling period to compute all the new filter coefficients and also process the next filter output. First, we look into the critical loop paths which need to be shortened somehow. Looking at Fig.5.10, we immediate see once source of this critical path, which is the cascaded adder chain. Unlike in the 1D FIR filter case where changing the direction of the adder chain resulted in reducing the latency, merely changing the direction does not shorten the critical cycle since all the coefficients have to be updated simultaneously. In order to reduce the adder chain latency, we can change the structure of the summation by constructing a adder tree (Fig.5.13). For a filter order of $m+1$, the accumulation latency of the cascaded adder chain is $m$ clock cycles, whereas the adder tree is $\lceil \log_2(m) \rceil$ cycles.

Next, we look into the number of cycle delays of the critical loop. Instead of updating the coefficients from the error signal immediately before the current sampling cycle and delay this updating cycle so that we have $d$ sampling periods for updating the coefficients, this will relax the recursive computation requirements and we can expect that the throughput will increase. In other words, modify the coefficient updating equation as

$$a(t,k) = \mu \cdot e(t-d) \cdot x(t-k-d) + a(t,k-d). \tag{5.10}$$

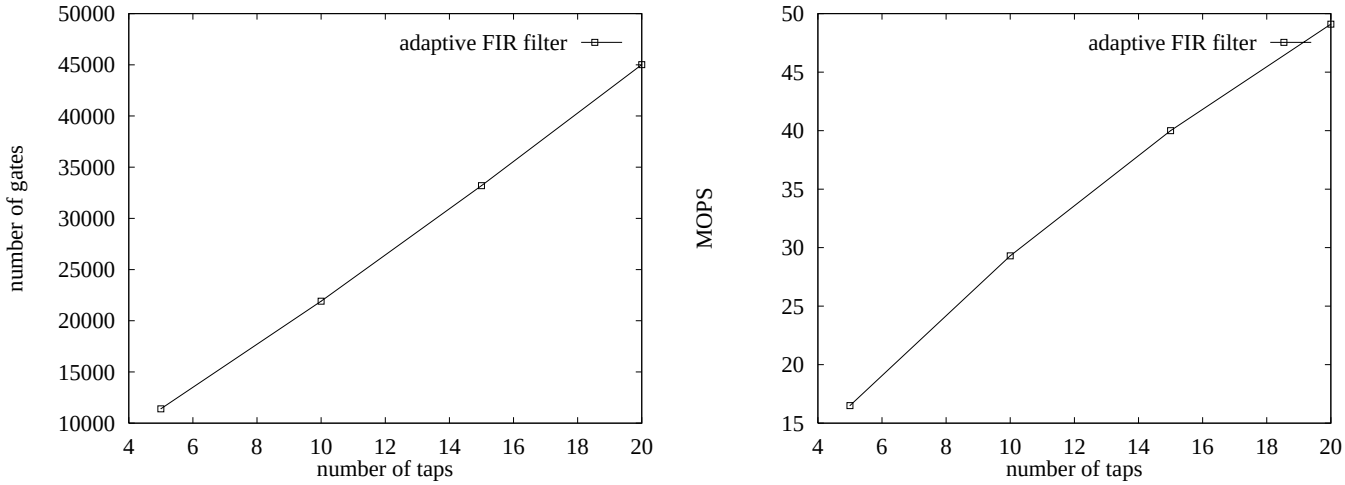



Figure 5.11: Synthesis results of adaptive filters.

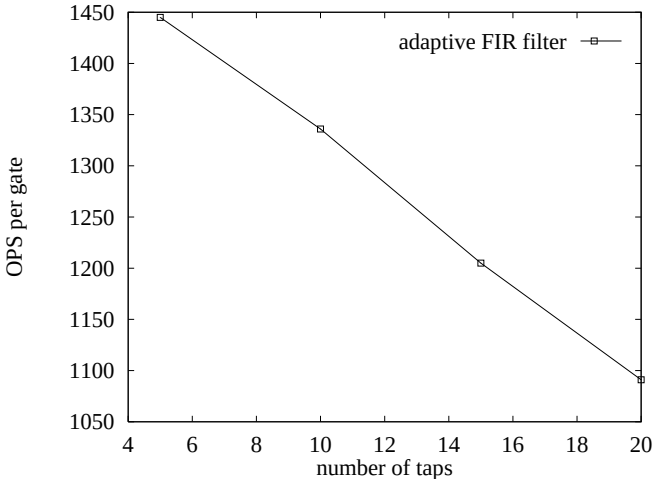
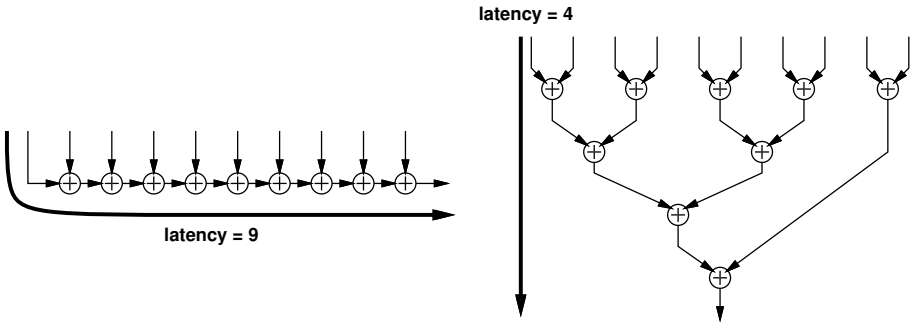



Figure 5.12: Synthesis results of adaptive filters.




(a) A cascaded adder chain. (b) An adder tree.

Figure 5.13: Different accumulation structure. For a 10-input accumulator, the latency is 9 clock cycles for a cascaded adder chain, and only 4 cycles for an adder tree.

| design | # taps | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| adapt10T-1 | 10 | 1303 | 21154 | 16.16 | 6.52 | 2536 | 2547 |
| adapt10T-2 | 10 | 1052 | 18028 | 17.14 | 6.52 | 2031 | 2042 |
| adapt10T-3 | 10 | 960 | 16918 | 17.71 | 6.52 | 1830 | 1841 |
| adapt10T-4 | 10 | 909 | 16396 | 17.93 | 6.52 | 1743 | 1754 |
| adapt10T-5 | 10 | 900 | 16318 | 18.02 | 6.52 | 1730 | 1741 |
| adapt10T-6 | 10 | 887 | 16048 | 18.09 | 6.52 | 1701 | 1712 |
| adapt10T-7 | 10 | 871 | 16036 | 18.30 | 6.52 | 1683 | 1694 |

| design | (# mult., # add) | MOPS @ 40MHz | OPS/gate | period | frequency |
|---|---|---|---|---|---|
| adapt10T-1 | (20,21) | 32.2 | 1520 | 51 | 0.78MHz |
| adapt10T-2 | (20,21) | 63.1 | 3502 | 26 | 1.54MHz |
| adapt10T-3 | (20,21) | 96.5 | 5702 | 17 | 2.35MHz |
| adapt10T-4 | (20,21) | 126.2 | 7694 | 13 | 3.08MHz |
| adapt10T-5 | (20,21) | 149.1 | 9137 | 11 | 3.64MHz |
| adapt10T-6 | (20,21) | 182.2 | 11287 | 9 | 4.44MHz |
| adapt10T-7 | (20,21) | 205.0 | 12784 | 8 | 5.00MHz |

Table 5.12: 10th order adaptive FIR filter synthesis results for different feedback delay values with adder tree structure . Signal word length is 8 bits.

The C++ description is shown below: We can see that the adder tree structure is efficiently coded using the for-loops and if-else statements. When the feedback delay value `delay` is 1, this adaptive filter behaves exactly the same as the earlier implementation. As the feedback delay value increases, we have more time to compute the new coefficients, and therefore the throughput increases. Table 5.12 and Fig.5.14, 5.15 show the synthesis results of the 10th-order FIR adaptive filter with different feedback delays. It is very interesting to see that while the throughput increases with the additional feedback delays until 7 sampling delays, the hardware size *decreases*. This is because with a shorter sampling period, the number of pipeline synchronization registers decreases in general. Therefore, the overall efficiency measured by the OPS/gate factor dramatically improves with the added feedback delay.

Although, we should not leave these synthesis results as it is. We should be reminded that even though we were able to explore an efficient bit-serial implementation of an FIR adaptive filter, we have actually modified the algorithm, and we need to verify the modified algorithm correctly adapts to the reference signal. Again, our C++ synthesis system becomes a very convenient tool for verifying the algorithm behavior since the code itself is a behavioral simulator. The simulation results are shown in Fig.5.16, 5.17. The reference system was implemented

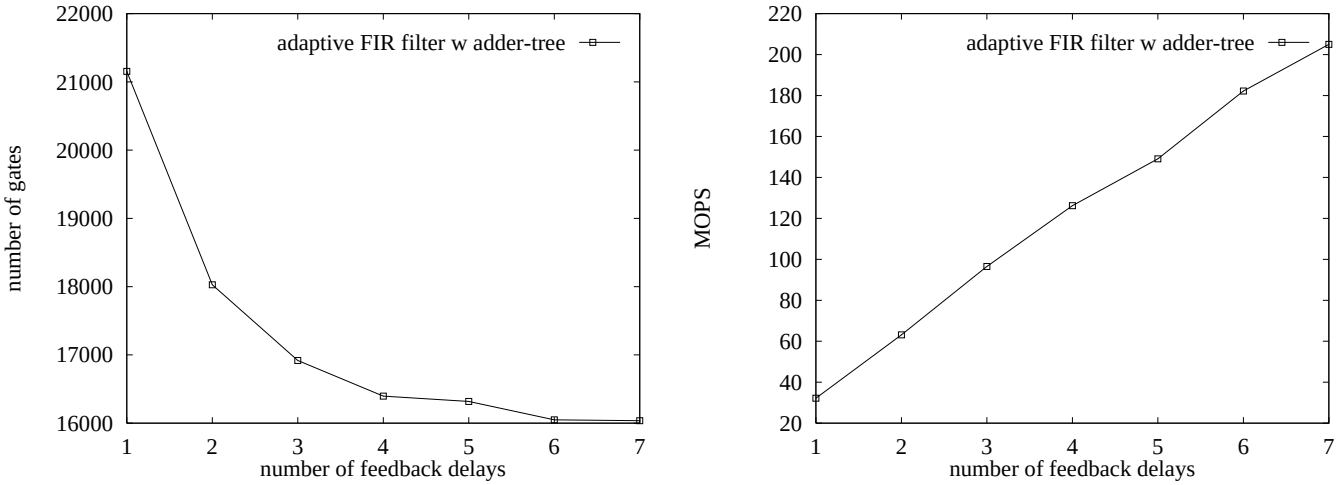



Figure 5.14: Synthesis results of adaptive filters with adder-tree and different feedback delays.

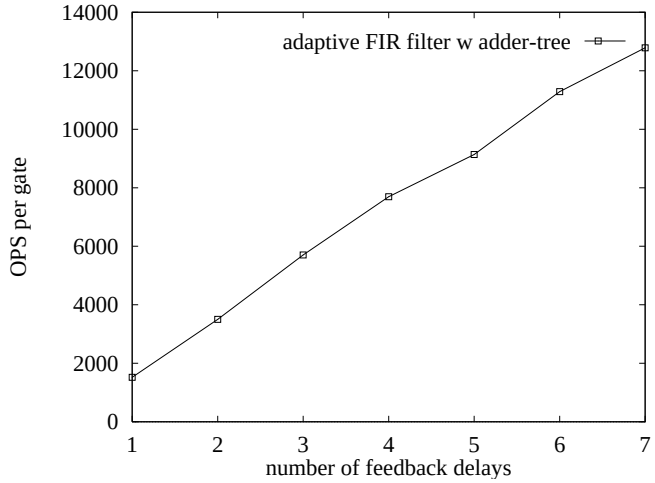



Figure 5.15: Synthesis results of adaptive filters with adder-tree and different feedback delays.

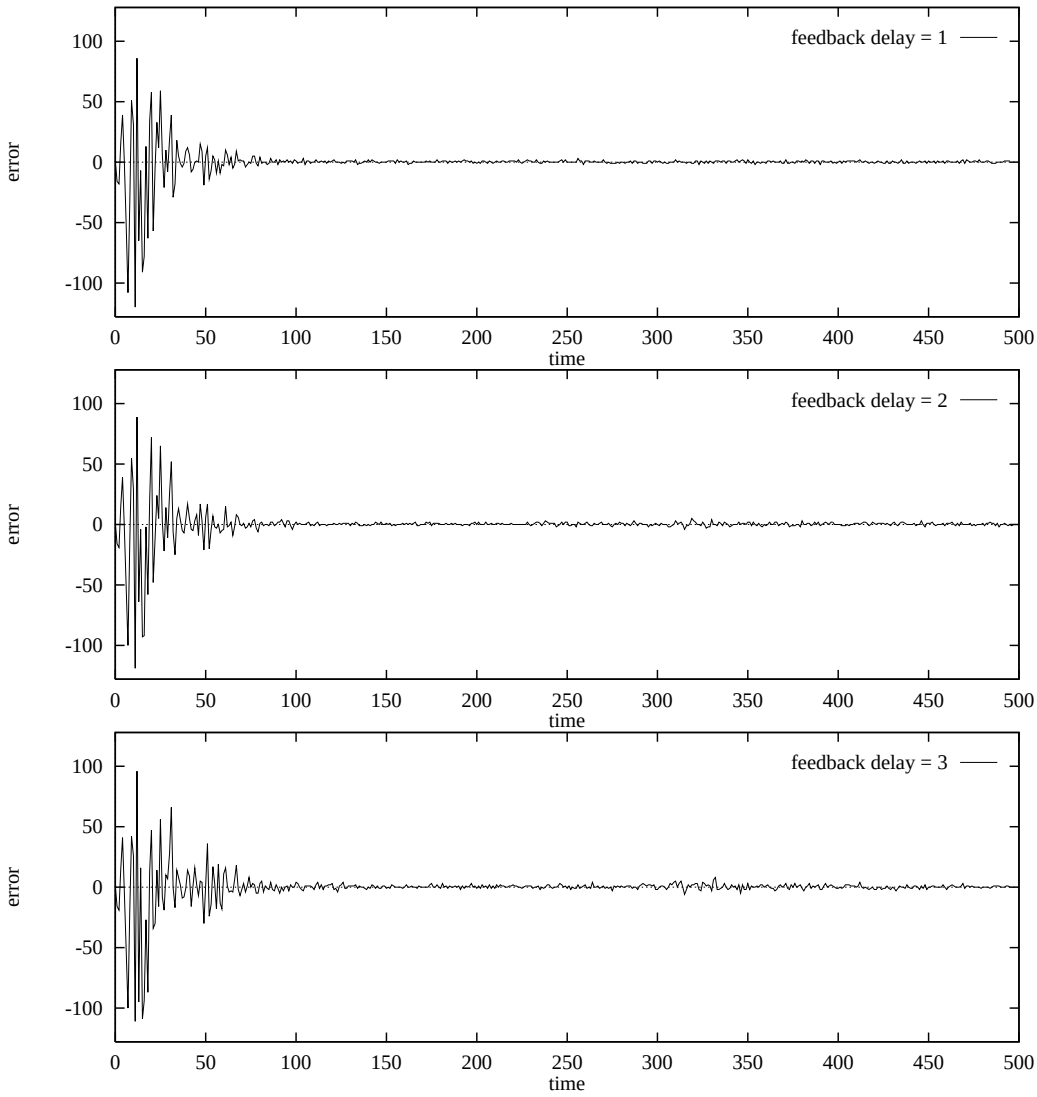



Figure 5.16: Simulation results of adaptive filters with different feedback delays. The reference system is implemented with a 10-tap FIR filter whose coefficients are $\{0.1, 0.05, -0.11, -0.05, 0.01, 0.2, 0.5, -0.2, 0.1, 0.1\}$. Error feedback scaling factor $\mu$ is set to 0.2. The signal input was generated by a random number generator.

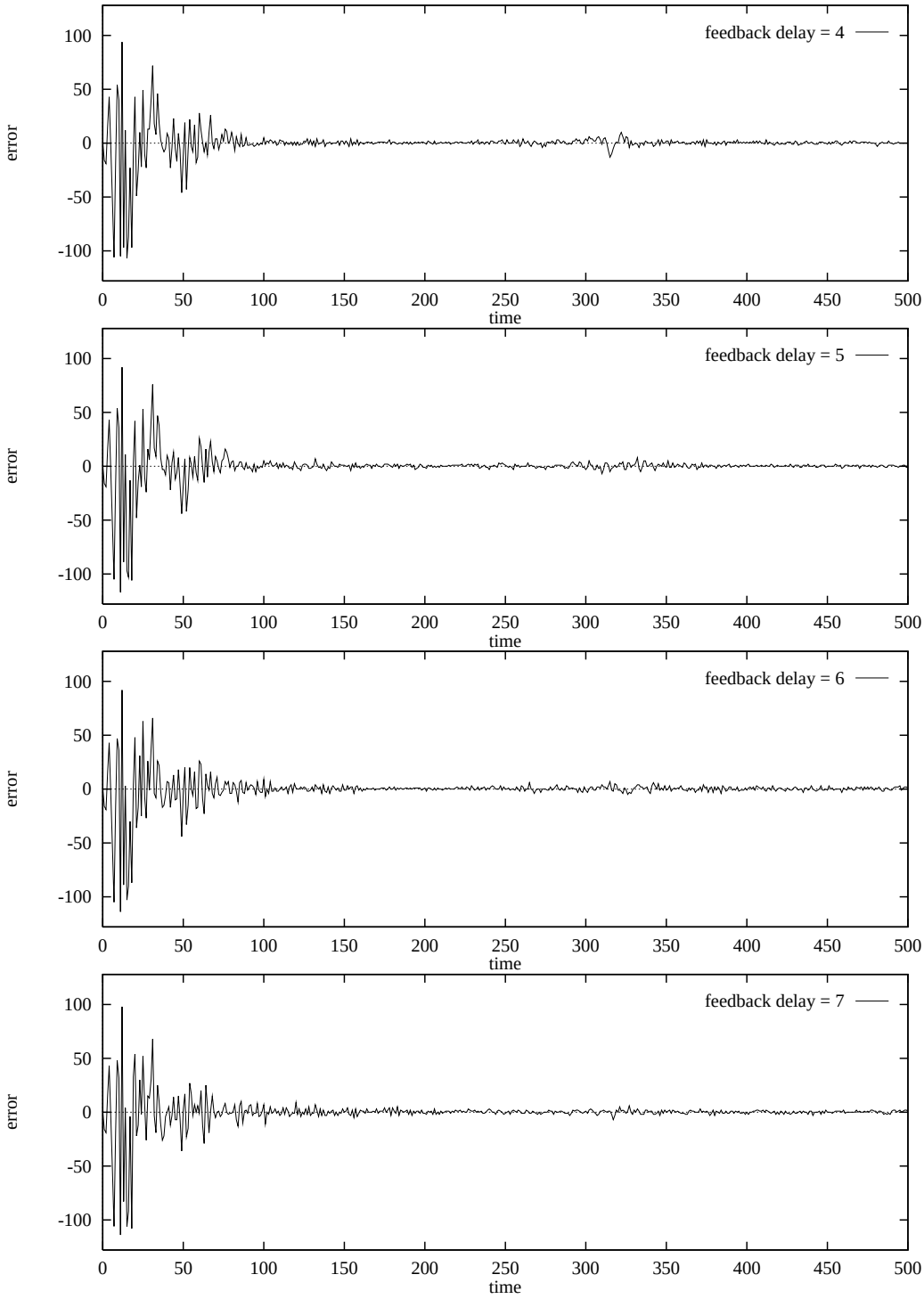



Figure 5.17: Simulation results of adaptive filters with different feedback delays

with a 10-tap FIR filter whose coefficients were set to $\{0.1, 0.05, -0.11, -0.05, 0.01, 0.2, 0.5, -0.2, 0.5, -0.2, 0.1, 0.1\}$. The error feedback scaling factor $\mu$ is set to 0.2. The signal input was generated by a random number generator. We can see that the adaptation speed becomes slightly slower as the feedback delay increases, which was totally expected. However, these behavior can vary depending on the input signal source, and also the characteristics of the reference system. Now, we can see a tradeoff between the *quality* of the algorithm implementation, hardware size and performance.

## 5.3 2-D Inverse Discrete Cosine Transform

Discrete cosine transform (DCT) and inverse discrete cosine transform (IDCT) are used in transform coding of still image (JPEG) and moving image (MPEG). The principle of transform coding is to transform the image signals in the spatial domain into signals in the frequency domain. DCT used in JPEG's and MPEG's encoder is given by the following equation [78]:

$$F(u,v) = \frac{C(u)C(v)}{4} \sum_{j=0}^{7} \sum_{k=0}^{7} f(i,j) \cos\left(\frac{(2j+1)u\pi}{16}\right) \cos\left(\frac{(2k+1)v\pi}{16}\right) \tag{5.11}$$

where

$$C(u) = \begin{cases} \frac{\sqrt{2}}{2} & (u=0) \\ 1 & (u \neq 0) \end{cases} . \tag{5.12}$$

Then, at the decoder, the inverse DCT (IDCT) is performed:

$$f(j,k) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)F(u,v) \cos\left(\frac{(2j+1)u\pi}{16}\right) \cos\left(\frac{(2k+1)v\pi}{16}\right) . \tag{5.13}$$

There are 4096 multiplications needed to process one block. Since DCT and IDCT are separable transforms, the equation of the IDCT can be rewritten as

$$f(j,k) = \frac{1}{4} \sum_{u=0}^{7} C(u) \cos\left(\frac{(2j+1)u\pi}{16}\right) \left[\sum_{v=0}^{7} C(v)F(u,v) \cos\left(\frac{(2k+1)v\pi}{16}\right)\right] . \tag{5.14}$$

Now, the number of multiplications is reduced to 1024.

The eq.(5.14) can be rewritten as

$$f(j,k) = \frac{1}{8} \sum_{u=0}^{7} C'(u) \cos\left(\frac{(2j+1)u\pi}{16}\right) \left[\sum_{v=0}^{7} C'(v)F(u,v) \cos\left(\frac{(2k+1)v\pi}{16}\right)\right] \tag{5.15}$$

where

$$C'(k) = \begin{cases} 1 & (k=0) \\ \sqrt{2} & (k \neq 0) \end{cases} .$$

The equation for the 1-D IDCT will be

$$\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 \\ C_0 & C_3 & C_6 & C_9 & C_{12} & C_{15} & C_{18} & C_{21} \\ C_0 & C_5 & C_{10} & C_{15} & C_{20} & C_{25} & C_{30} & C_{35} \\ C_0 & C_7 & C_{14} & C_{21} & C_{28} & C_{35} & C_{42} & C_{49} \\ C_0 & C_9 & C_{18} & C_{27} & C_{36} & C_{45} & C_{54} & C_{63} \\ C_0 & C_{11} & C_{22} & C_{33} & C_{44} & C_{55} & C_{66} & C_{77} \\ C_0 & C_{13} & C_{26} & C_{39} & C_{52} & C_{65} & C_{78} & C_{91} \\ C_0 & C_{15} & C_{30} & C_{45} & C_{60} & C_{75} & C_{90} & C_{105} \end{bmatrix} \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \\ F(6) \\ F(7) \end{bmatrix} \tag{5.16}$$

where

$$C_k = \begin{cases} 1 & (k=0) \\ \sqrt{2}\cos\frac{k}{16}\pi & (k \neq 0) \end{cases}, \tag{5.17}$$

$\{f(i)|i = 0,1,\cdots,7\}$ are the output row or column pixels of the 1-D IDCT and $\{F(i)|i = 0,1,\cdots,7\}$ are the input row or column DCT coefficients. Using the symmetrical property of the cosine function and the relation $C_0 = C_4 = 1$, above equation can be rewritten as

$$\begin{aligned}
\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} &= \begin{bmatrix} C_0 & C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 \\ C_0 & C_3 & C_6 & -C_7 & -C_4 & -C_1 & -C_2 & -C_5 \\ C_0 & C_5 & -C_6 & -C_1 & -C_4 & C_7 & C_2 & C_3 \\ C_0 & C_7 & -C_2 & -C_5 & C_4 & C_3 & -C_6 & -C_1 \\ C_0 & -C_7 & -C_2 & C_5 & C_4 & -C_3 & -C_6 & C_1 \\ C_0 & -C_5 & -C_6 & C_1 & -C_4 & -C_7 & C_2 & -C_3 \\ C_0 & -C_3 & C_6 & C_7 & -C_4 & C_1 & -C_2 & C_5 \\ C_0 & -C_1 & C_2 & -C_3 & C_4 & -C_5 & C_6 & -C_7 \end{bmatrix} \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \\ F(6) \\ F(7) \end{bmatrix} \\
&= \left( \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} C_1 & C_3 & C_5 & C_7 \\ C_3 & -C_7 & -C_1 & -C_5 \\ C_5 & -C_1 & C_7 & C_3 \\ C_7 & -C_5 & C_3 & -C_1 \end{bmatrix} \begin{bmatrix} F(1) \\ F(3) \\ F(5) \\ F(7) \end{bmatrix} \right. \\
&+ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} C_2 & C_6 \\ C_6 & -C_2 \end{bmatrix} \begin{bmatrix} F(2) \\ F(6) \end{bmatrix} \right. \\
&+ \left. \left. \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \left( \begin{bmatrix} 1 \\ -1 \end{bmatrix} \cdot F(4) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot F(1) \right) \right) \right) .
\end{aligned} \tag{5.18}$$

SFG description of this algorithm is shown in Fig.5.18 and the C++ description of this algorithm is shown in the appendix (Fig.D.6).

One note on the implementation is that the scaler coefficients `cc2[]` are scaled down by 2 compared from the values in eq.(5.17) since some of the coefficient values exceed 1. Thus `cc2[i]`$= \frac{\sqrt{2}}{2}\cos\frac{k\pi}{16}$. Because of this, the inputs `f[0]` and `f[4]` are shifted by one bit. As a result, the output values are scaled down by a factor of 2. Also, as indicated in Fig.5.18, there are 10 scaler pairs which share the same multiplicant input. Therefore, the input shift-register can be shared within each pair.

### 5.3.1 Chang's DCT Implementation Using 14 Multiplications

Chang et al. [78] have proposed an efficient implementation of using only 14 multiplications to implement an 8-point DCT and IDCT. Their implementation of IDCT is illustrated in Fig.5.19. C++ description is illustrated

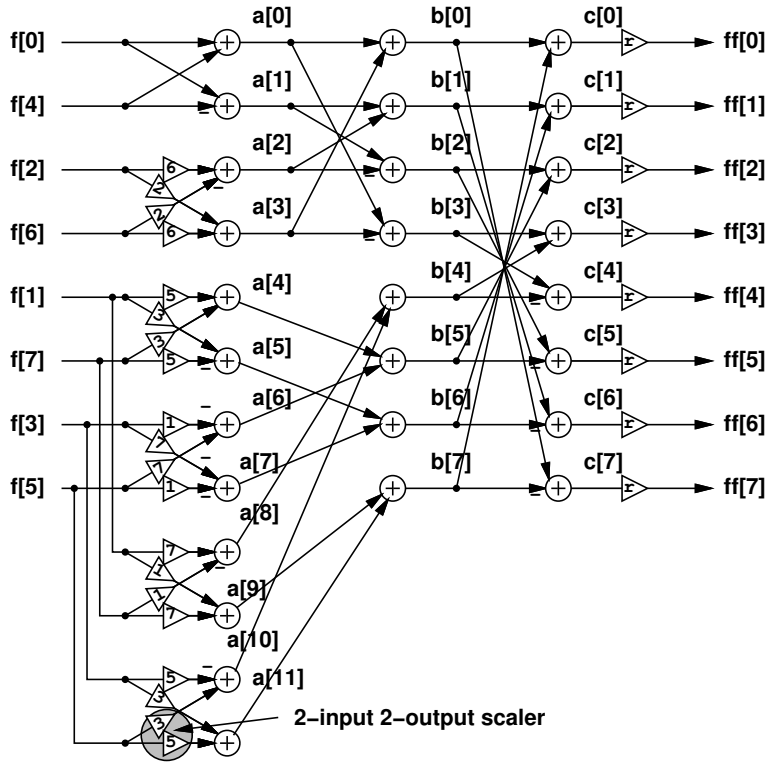



Figure 5.18: 8-point IDCT circuit using 20 multiplications.

| design | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|
| IDCT-I | 1012 | 16109 | 15.73 | 5.31 | 1788 | 1807 |
| IDCT-II (Chang's) | 1038 | 15909 | 15.142 | 5.182 | 1894 | 1913 |

| design | (# mult., # add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| IDCT-I | (20, 28) | 120.00 | 7449 | 16 | 52+4 | 2.5MHz |
| IDCT-II | (14, 28) | 105.00 | 6600 | 16 | 76+12 | 2.5MHz |

| design | # blocks/sec | # MPEG1 frames/sec |
|---|---|---|
| IDCT-I | 156250 | 65.76 |
| IDCT-II | 156250 | 65.76 |

Table 5.13: 8-point IDCT synthesis results. "IDCT-II" is derived from Chang's implementation. "# MPEG frame/sec" denote the number of image frames per second where the image size is 352x288 for the luminance signal (44x36 blocks) and 176x144 for the two chrominance signals (22x18x2 blocks).

in the appendix (Fig.D.7). This implementation is derived by the trigonometric relationships

$$\begin{aligned} \cos\tfrac{k}{16}\pi &= \cos\tfrac{k-4}{16}\pi\cdot\cos\tfrac{\pi}{4} - \sin\tfrac{k-4}{16}\pi\cdot\sin\tfrac{\pi}{4} \\ &= \tfrac{\sqrt{2}}{2}(\cos\tfrac{k-4}{16}\pi - \cos\tfrac{12-k}{16}\pi) \\ &= \tfrac{\sqrt{2}}{2}(\cos\tfrac{k+4}{16}\pi + \cos\tfrac{12-k}{16}\pi). \end{aligned} \tag{5.19}$$

To be more specific, of the 16 multiplications in the second row of eq.(5.18) can be reduced to 10 since

$$\begin{aligned} C_1 &= \tfrac{\sqrt{2}}{2}(C_3 + C_5) \\ C_7 &= \tfrac{\sqrt{2}}{2}(C_3 - C_5) \\ C_3 &= \tfrac{\sqrt{2}}{2}(C_1 + C_7) \\ C_5 &= \tfrac{\sqrt{2}}{2}(C_1 - C_7), \end{aligned} \tag{5.20}$$

and therefore,

$$\begin{aligned} \begin{bmatrix} C_1 & C_3 & C_5 & C_7 \\ C_3 & -C_7 & -C_1 & -C_5 \\ C_5 & -C_1 & C_7 & C_3 \\ C_7 & -C_5 & C_3 & -C_1 \end{bmatrix} &= \begin{bmatrix} \frac{\sqrt{2}}{2}(C_3+C_5) & \frac{\sqrt{2}}{2}(C_1+C_7) & \frac{\sqrt{2}}{2}(C_1-C_7) & \frac{\sqrt{2}}{2}(C_3-C_5) \\ C_3 & -C_7 & -C_1 & -C_5 \\ C_5 & -C_1 & C_7 & C_3 \\ \frac{\sqrt{2}}{2}(C_3-C_5) & -\frac{\sqrt{2}}{2}(C_1-C_7) & \frac{\sqrt{2}}{2}(C_1+C_7) & -\frac{\sqrt{2}}{2}(C_3+C_5) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ \sqrt{2} & 0 \\ 0 & \sqrt{2} \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2}C_3 & 0 & 0 & -\frac{\sqrt{2}}{2}C_5 \\ \frac{\sqrt{2}}{2}C_5 & 0 & 0 & \frac{\sqrt{2}}{2}C_3 \end{bmatrix} \\ &= \begin{bmatrix} -1 & -1 \\ \sqrt{2} & 0 \\ 0 & \sqrt{2} \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0 & -\frac{\sqrt{2}}{2}C_7 & -\frac{\sqrt{2}}{2}C_1 & 0 \\ 0 & -\frac{\sqrt{2}}{2}C_1 & \frac{\sqrt{2}}{2}C_7 & 0 \end{bmatrix} \end{aligned} \tag{5.21}$$

Again, since the magnitude of some of the scaler coefficients exceed 1, the scaler coefficients are either scaled down by a factor of 2 (`cc2[]`) or by a factor of 4 (`cc[]`). `sqrt2_2` denotes the value $\sqrt{2}/2$ (Fig.D.7 in appendix).

The synthesis results of the two types of implementation is shown in Table 5.13. It is surprising to see that even though Chang's implementation has four less multiplications than the first implementation, and yet the overall hardware size is larger. This is due to two reasons:

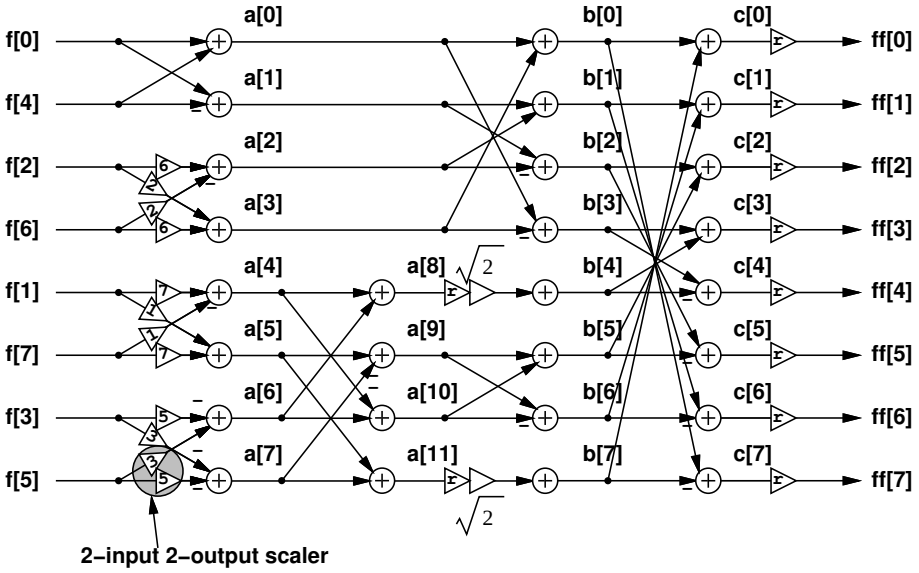



Figure 5.19: Chang's 8-point IDCT circuit.

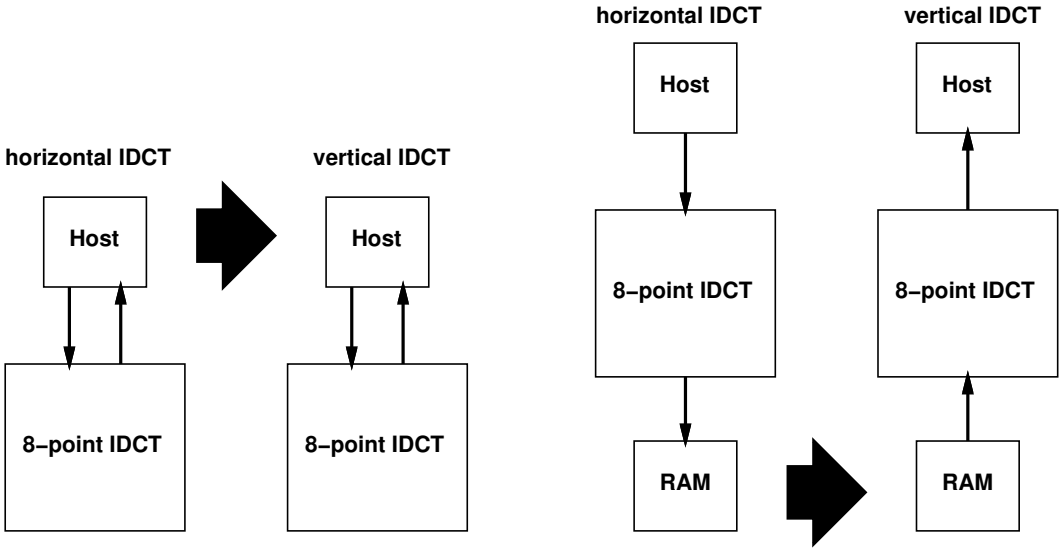
1. Comparing the structure of the two implementations, all the signal paths are fairly balanced in the first implementation (IDCT-I), whereas Chang's implementation (IDCT-II) has a very uneven signal paths. Some path include two multiplications and others have one or no multiplications. In order to synchronize the pipeline, Chang's implementation requires more pipeline registers than IDCT-I.

2. The total latency of IDCT-I is 52, and the output signals are further delayed by 4 clock cycles to avoid the bus conflict between the inputs. The eight input signals are transmitted at cycles 0 through 7, and the eight output signals are transmitted at cycles 8 though 15. Therefore, the output signals have the latency of a total of 56 where the output comes out 3 sampling cycles after the input. The total latency of Chang's IDCT-II implementation is 76, and the output signals have to be further delayed by 12 cycles to avoid bus conflict. Therefore, Chang's implementation requires 32 more CLBs just to implement these extra delays on the output.

As we have seen here, there are many other parameters besides the number of operations which determine the complexity of the hardware. The number of multiplication being less does not automatically mean that the hardware is smaller or faster. Using our C++ datapath synthesis system, the designer can immediately see the trade-offs between different implementations in terms of hardware size.

### 5.3.2 Two-Pass 2-D IDCT Implementation With Data Buffering at the Host Computer

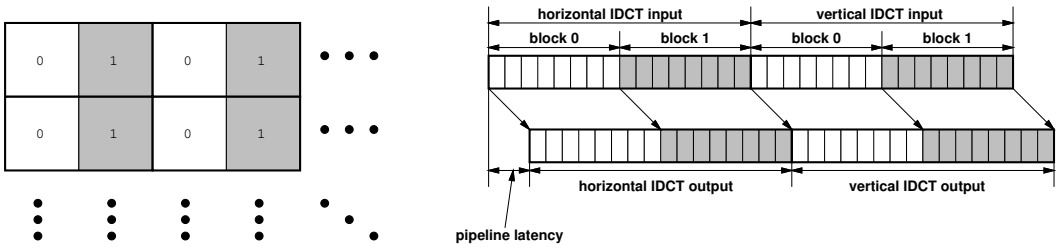
In order to implement a $8 \times 8$ 2-D IDCT, we run two passes through this 8-point IDCT circuit. If all the other necessary data buffering is done on the host computer, the C++ code for a full $8 \times 8$ 2-D IDCT is described in the appendix (Fig.D.8). The behavior of this code is illustrated in Fig.5.21. Two 8 by 8 pixel blocks are grouped together. Let us call this the *macro block*. For each macro block which is accessed by the two outer for-loops, there are two passes: horizontal IDCT (`k = 0`) and vertical IDCT (`k = 1`). During the first horizontal IDCT pass, 16 horizontal line segments (each line has 8 pixels) from the input image array `in` are sequentially fed into the 8-point




(a) Two–pass 2–D IDCT with data buffering at the host computer.

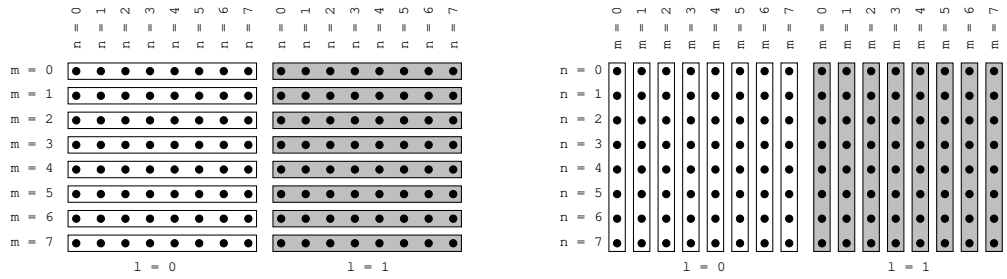
(b) Two–pass 2–D IDCT with data buffering at the local RAM.

Figure 5.20: Two different data buffering implementations for 2-D IDCT.




(a) Two 8x8 pixel blocks are grouped together.

(b) 8–point IDCT is alternated between the two blocks.




(c) Horizontal 8–point IDCT.

(d) Vertical 8–point IDCT.

Figure 5.21: $8 \times 8$ 2-D IDCT using 8-point IDCT circuit. Because of the pipeline latency, two blocks are grouped together and processed sequentially in order to avoid idle cycles. On the first run, 8-point IDCT is performed in the horizontal direction. On the second run, it is performed in the vertical direction.

| design | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|
| IDCTmem-I | 1233 | 19443 | 15.54 | 4.784 | 2172 | 2208 |
| IDCTmem-II (Chang's) | 1259 | 18955 | 15.06 | 4.63 | 2278 | 2314 |

| design | (# mult., # add) | MOPS @ 40MHz | OPS/gate | period | latency | frequency |
|---|---|---|---|---|---|---|
| IDCTmem-I | (20, 28) | 120.00 | 6172 | 16 | 54+2 | 2.5MHz |
| IDCTmem-II | (14, 28) | 105.00 | 5539 | 16 | 78+10 | 2.5MHz |

Table 5.14: 2D IDCT synthesis results. Data buffering is done on the external RAM.

IDCT circuit and the processed data is stored in the intermediate image array `inter`. During the second vertical IDCT pass, 16 vertical line segments from the intermediate image are sequentially fed into the IDCT circuit and the processed data is stored in the output image array `out`. The reason why two 8 by 8 blocks are grouped together is due to the processing latency of the IDCT circuit. In order to start the vertical IDCT, the results of all eight horizontal IDCT calculations are needed. If each block was processed one at a time, there will be some idle cycles while the host computer waits until all the horizontal IDCT calculations are done. By processing two blocks at a time, all the horizontal IDCT results will be ready by the time the host computer finishes feeding the horizontal line segments of the second block, assuming that the IDCT processing latency is less than 8 sampling periods.

### 5.3.3 Two-Pass 2-D IDCT Implementation With Data Buffering at the Coprocessor Memory

The above two-pass implementation with data buffering at the host computer needs to download and read back the whole image twice. If IO bandwidth between the host computer and the coprocessor is limited, we want to minimize the data transaction as much as possible between the two components. Next, we present another implementation with the data buffering at the coprocessor memory. The C++ description is shown in Fig.D.9 in the appendix.

The intermediate results from the horizontal IDCT calculations are stored in `ram` whose address is generated by the control variables `s1`, `s2` and `s3`. The host computer now only needs to download the image and read back the results just once. Notice the strong similarity of the C++ description compared to the previous implementation. The programmer only needs to change the integers inside the for-loops to control variables, and change the intermediate data buffering from the software data array to the hardware RAM. This is the powerful feature of our function-based C++ design capture method. By changing the variable type from integer to a custom data class which we provide, all the operators are overloaded and becomes function calls. The synthesis results of the 2D IDCT circuit including the memory control circuit are shown in Table 5.14.

## 5.4 Digital Neural Network

Cox et al. [25] implemented a very fast digital neural network called GANGLION using Xilinx FPGA chips. Because of its extremely high performance (4.48 billion interconnections per second, 20 million decision per second), their work has triggered many other researches on digital neural networks using field-programmable devices. [37][35][6]. Digital neural networks, also referred to as connectionist classifiers, are parameterized devices implemented by scalers, accumulators and sigmoid function generators which group objects into one out of a set of predefined categories or *classes*. The objects are all described and represented by a set of measurements of *features*. The class assignment is based on the experience that was gained during an adaptation procedure. During this adaptation phase, optimal parameter set for the scalers are calculated. GANGLION is a three-layer feed-forward net with 12 nodes in the input layer, 14 nodes in the hidden layer, and 4 nodes in the output layer (Fig.5.22). Each node

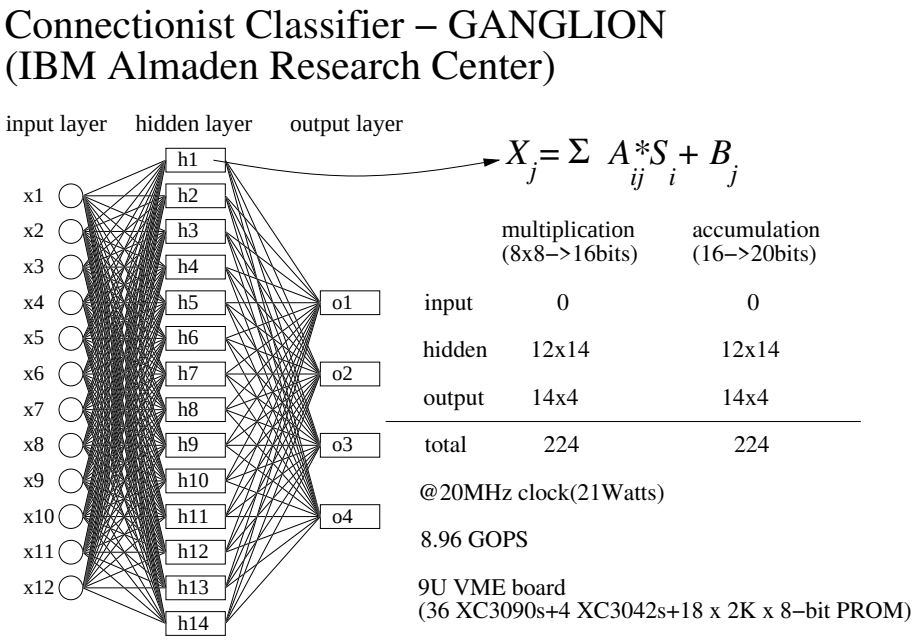



Figure 5.22: GANGLION connectionist classifier.

except the ones in the input layer performs the following computation:

$$x_j^{(l)} = \sum_i a_{ij}^{(l)} s_i^{(l-1)} + b_j^{(l)} \tag{5.22}$$

where $a_{ij}^{(l)}$ is a weight assigned to the edge between node $i$ in layer $l-1$ and node $j$ in layer $l$, $s_i^{(l-1)}$ is the current state of unit $i$ in layer $l-1$, and $b_j^{(l)}$ is the bias value for unit $j$ in layer $l$. The current state of the node $j$ in layer $l$ ($s_j^{(l-1)}$) is determined by applying an activation function to $x_j^{(l)}$ which is calculated as

$$s_j^{(l)} = \frac{1}{1 - e^{-x_j^{(l)}}} \tag{5.23}$$

This is implemented in an external PROM lookup table. To reduce the PROM size, they scale the output signal $x_j^{(l)}$ which is expressed in 22 bits down to 10 bits plus a saturation bit. Scaling factor is precalculated and the scaling operation is hardwired.

Since the weight is fixed during the classification phase, this computation can be implemented with a set of scalers. They have come up with a novel bit-parallel scaler design in which the 8-bit input scaler with 8-bit scaling constant is divided into two 4-bit by 8-bit scaling operations and each scaling is implemented using 4-input LUTs on the SRAM-based Xilinx FPGAs. Each scaler which produces 12-bit product requires only 6 CLBs. This scaler design led to an extremely efficient design. Also, their accumulators are fully pipelined and operates at a very high speed. They report that the hardware is functional at 20MHz clock which is equal to the data sampling frequency, and is able to perform 4.30 billion neural interconnections per second which is equivalent to 8.6 GOPS excluding the calculation of the activation function. They have manually designed and also did the layout manually, which led to an exceptionally efficient design. They have used 36 XC3090-70 FPGAs and 4 XC3042-70 FPGAs. XC3090

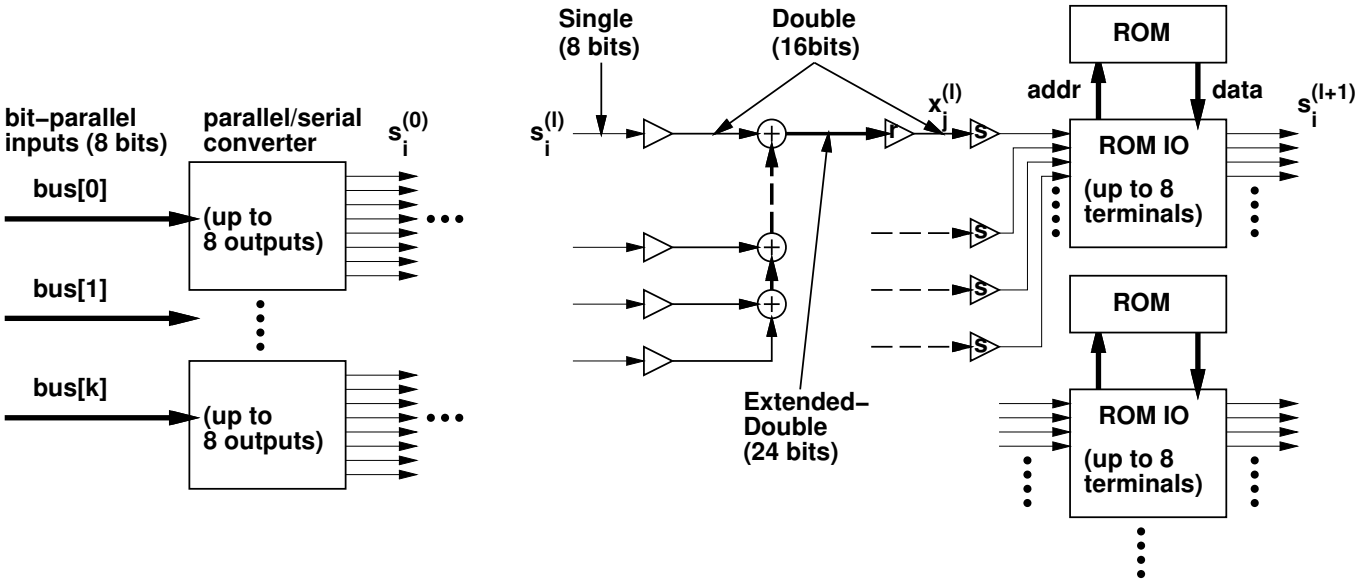



Figure 5.23: Bit-serial implementation of a digital neural network.

FPGA has 320 CLBs and 144 IOs, whereas XC3042 as 144 CLBs and 96 IOs. If their design were to converted to the faster XC3100 FPGAs, their performance would increase to 25.8 GOPS at 60 MHz clock.

We have also designed the same classifier application using our bit-serial datapath synthesis system. The C++ code is shown in Fig.D.10 in the appendix. We note here that the algorithm has been modified in order to be synthesizable in our synthesis program:

1. It does not include the bias calculation since we currently do not support adders with one constant input.
2. We do not support scaling operations to $x_j^{(l)}$. We use the extended double precision signal (24 bits) for $x_j^{(l)}$ and is simply converted into single precision signal (8 bits) by rounding and saturation.

While these modifications may inhibit out digital neural network implementation for practical applications, we identify this densely interconnected structure as an excellent benchmarking vehicle to evaluate our partitioner and the routability of our bit-serial circuits. This application is indeed most challenging for the layout synthesis problem compared to other application examples that we present in this paper because of the large fan-in (accumulation) and fan-out (broadcasting to multiple nodes) structure of the datapath. Synthesis results are given in Table 5.15 and Fig.5.24. We also see several benefits using the bit-serial circuits compared to bit-parallel implementation of GANGLION:

1. The number of IOs connecting to the digital neural network is reduced by a factor of up to 8 (Fig.5.23). While GANGLION requires 12 8-bit wide input buses to feed the data simultaneously at the frequency of up to 20 MHz and 4 output buses to capture the processed data, our bit-serial implementation requires only 2 buses at the input and 1 bus at the output.
2. Also, the number of PROMs also reduces by a factor of up to 8 since each PROM can be shared by up to 8 nodes.

The drawback of the bit-serial implementation compared to the highly pipelined bit-parallel implementation of GANGLION is the speed. When such high degree of pipelining is applied to the bit-parallel architecture, bit-serial

| design | network structure | # CLBs | # gates | gates/CLB | gates/LUT | # FFs | # nets |
|---|---|---|---|---|---|---|---|
| N-4-4-4 | (4, 4, 4) | 1126 | 20253 | 17.99 | 5.36 | 2137 | 2164 |
| N-8-8-4 | (8, 8, 4) | 3082 | 56465 | 18.26 | 5.55 | 5803 | 5803 |
| N-12-14-4 | (12, 14, 4) | 6964 | 127962 | 18.33 | 5.6 | 13068 | 13112 |

| design | (# mult., #add) | MOPS @ 40MHz | OPS/gate | period | frequency |
|---|---|---|---|---|---|
| N-4-4-4 | (32,24) | 280.0 | 13695 | 8 | 5.0MHz |
| N-8-8-4 | (96,84) | 900.0 | 15939 | 8 | 5.0MHz |
| N-12-14-4 | (224, 206) | 2150.00 | 16802 | 8 | 5.0MHz |

Table 5.15: Digital neural network synthesis results. Signal word length is 8 bits. Design "N-4-4-4" is a 3-layer network with 4 nodes at each layer. Design "N-8-8-4" is a 3-layer network with 8 nodes at the input and hidden layer, and 4 nodes at the output layer. Design "N-12-14-4" is a 3-layer network with 12 nodes at the input layer, 14 nodes at the hidden layer, and 4 nodes at the output layer.

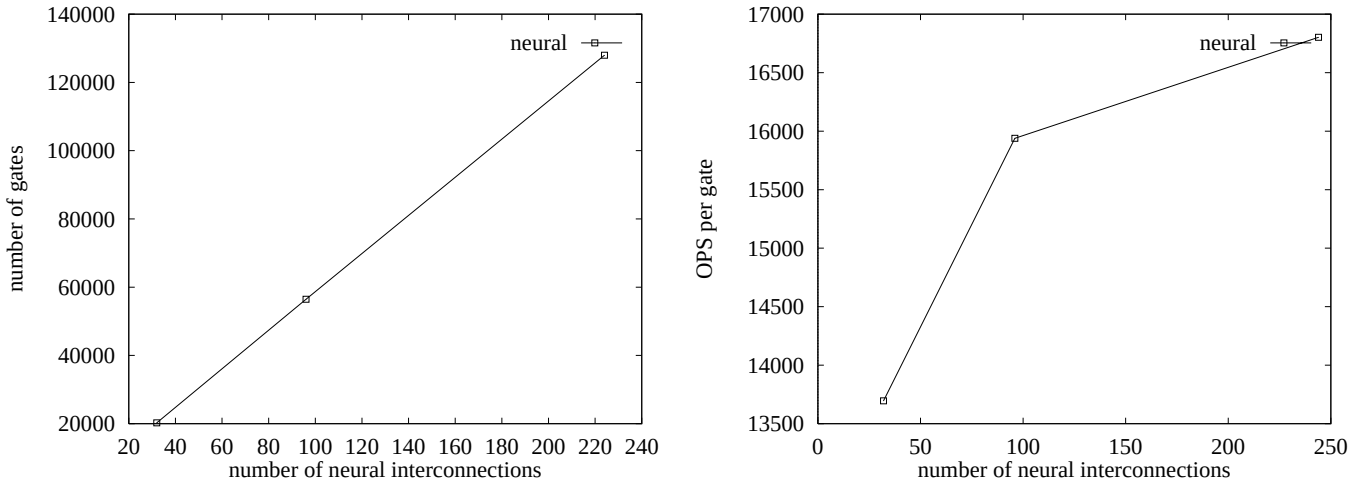



Figure 5.24: Digital neural network synthesis results. Three different designs are expressed by the number of neural interconnections in the design.

| Design | hardware size (# XC3190A) | clock speed | GOPS | efficiency (GOPS/# XC3190A) |
|---|---|---|---|---|
| bit-serial | 23 | 40MHz | 2.15 | 0.093 |
| GANGION | 38 | 60MHz | 25.80 | 0.679 |

Table 5.16: Comparison between GANGLION and our bit-serial digital neural network implementation. The performance figures of GANGLION is converted into the latest FPGA technology (XC3100A-3) which is roughly 3 times as fast as XC3090-70 which they used. Also, in the bit-serial version, biasing operation is not implemented.

architecture is still bound to be slower. Also, since we would rely on the automatic placement and routing tool, we would have to assume a slower clock speed than their manually placed design which can function at 60MHz using the XC3100 FPGAs. Assuming a 40MHz clock, our bit-serial implementation will achieve 2.15 GOPS compared to 25.8 GOPS for GANGLION (Table 5.16). As we will show in the next chapter, our bit-serial implementation fits in 23 XC3190A (identical number of CLBs, IOs and routing tracks compared to XC3090, slight increase in the connectivity between the routing channels and the logic block pins) FPGAs compared to their implementation using 36 XC3090s and 4 XC3042 which is roughly equivalent to 38 XC3090s. Therefore our implementation is 42% smaller. In terms of efficiency, their implementation is 7.3 times more efficient than our bit-serial implementation disregarding the factor that the bias operation is not implemented in the bit-serial version (if this was included, the bit-serial design will see a slight increase in hardware), mainly due to their compact bit-parallel scaler designs, heavily pipelined structure, and a complete manual design on logic-level and a manual layout.

# Chapter 6

# Bit-Serial Pipeline Partitioning Method for Maximum Logic Utilization

## 6.1 General Circuit Network Partitioning

As the first step in the FPMCM layout synthesis, we have to partition the bit-serial pipeline network into smaller groups in order to map each group into an individual FPGA. The circuit partitioning problem, one application of a graph partitioning problem, has been studied by many researchers. The main goal here is to find cuts which partition the graph into two groups (hence, a two-way partitioning or bipartitioning) such that the number of edges crossing the cut is minimized. This graph partitioning problem forms a basis for many fundamental CAD problems:

1. *Hierarchical system design* : Divide the given circuit at all levels of the system hierarchy :
   (a) System level → board level.
   (b) Board level → MCM level.
   (c) MCM level → chip level.

   The routing cost is higher on the higher level, therefore minimizing the number of edges crossing the partition cuts is essential in reducing the overall size and cost. Note that our case fits into the last category. There are more routing resources on the FPGA than on the MCM substrate and FPIC.

2. *Logic simulation* : Divide the given circuit into subcircuits to perform a parallel circuit simulation. The number of edges crossing the cut denotes the amount of communications needed between processors.

3. *VLSI circuit layout* : Perform a circuit placement by partitioning the circuit into smaller modules and perform a placement and routing on these reduced number of modules.

If minimizing the edges crossing the cuts is the only goal, there will always exist a optimal partition, in which all but one group have no member, and therefore no edge crosses the cut. In order to avoid this uninteresting obvious solution, the problem incorporates the size of each group as one of the partition criteria either as a constraint or a part of the cost function :

**Min-cut partitioning problem** For a given weighted hyper-graph $HG(\overline{E}, V, C)$ (A hyper-edge is expressed as a set of vertices $\overline{e} = \{v_0, v_1, \cdots, v_{k-1}\} \in \overline{E}$, and $C$ is the cost given to each hyper-edge), derive a bipartition

$$\begin{array}{rcl} S & = & (A_0, A_1) \\ A_0 \cap A_1 & = & \phi \\ A_0 \cup A_1 & = & V \end{array} \tag{6.1}$$

such that

$$C_{partition} = \sum_{\overline{e} \in \overline{E_{cut}}} cost(\overline{e}) \tag{6.2}$$

is minimized, where

$$\overline{E_{cut}} = \{\overline{e} | \overline{e} \in \overline{E}, \overline{e} \not\subset A_0, \overline{e} \not\subset A_1\} \tag{6.3}$$

subject to

$$F_i^- \leq |A_i| \leq F_i^+ \quad (i = 0, 1). \tag{6.4}$$

When

$$\begin{array}{rcl} F_i^- & = & (1 - \alpha) \cdot |V|/2 \\ F_i^+ & = & (1 + \alpha) \cdot |V|/2, \end{array} \tag{6.5}$$

this is called *balanced bipartitioning solution* with respect to capacity slack variable $\alpha$.

**Ratio-cut partition problem** For a given weighted hyper-graph $HG(\overline{E}, V, C)$, derive a bipartition

$$\begin{array}{rcl} S & = & (A_0, A_1) \\ A_0 \cap A_1 & = & \phi \\ A_0 \cup A_1 & = & V \end{array} \tag{6.6}$$

such that

$$C_{partition} = \sum_{e \in E_{cut}} \frac{cost(\overline{e})}{|A_0||A_1|} \tag{6.7}$$

Ratio-cut partitioning is often used to identify clusters in the circuit network which becomes the basis for constructive circuit module placement for VLSI layout [45], or to simply reduce the number of elements in the circuit network for fast partitioning [22]. For our case where each partition size is limited, min-cut partitioning is needed to satisfy the size constraints. Many of the min-cut bipartitioning algorithms [60][83][38][62][22] take the approach of sequentially moving the nodes from one partition to the other while satisfying the capacity constraint and finding the sequence of moves which minimizes the cut cost. This pass is repeated until no further improvement is observed. The first such method was proposed by Kernighan and Lin [60] which handled graphs with two-point edges instead of hyper-graphs with hyper-edges, and later expanded by Schweikert and Kernighan to handle hyper-graphs [83]. Fiduccia and Mattheyses improved these methods by reducing the running time to $O(P)$ where $P$ is the number of pins in the hyper-graph. Krishnamurthy further added lookahead capability to improve the quality of the partition [62]. All of these methods start with a random feasible partitioning as the initial solution and the quality of the final partition solution tend to vary. Therefore, in practice, several runs of these methods using different initial solutions are applied to obtain the best solution. The technique of circuit clustering is applied in Cheng and Wei's method [22] in order to speed up this process by reducing the number of nodes in the circuit network. The best partition solution obtained by the several runs of the clustered circuit network partitioning is used as the initial feasible partitioning input for the final partition run on the original circuit network.

Another popular approach is the graph spectral method [39][12][45]. With this approach, one first needs to convert the hyper-graph representation of the circuit network into a graph representation $G(E, V)$, and then

derive the eigenvectors of the matrix $Q = D - A$ where $D$ is the degree matrix and $A$ is the adjacency matrix, each derived from graph G. The eigenvectors are calculated by solving $(Q - \lambda I)x = 0$ where $\lambda$ is the eigenvalue which are the solutions to the one-dimensional quadratic placement. The eigenvector corresponding to the second smallest eigenvalue is often used as the partitioning guideline. Hagen and Kahng [45] showed that the second smallest eigenvalue gives a provably good approximation of the optimal ratio cut partition cost.

$K$-way partitioning methods can be formulated based on the bipartitioning methods. Recursive bipartitioning method is the most straightforward adaption [60][83] where bipartitioning process is repeatedly applied to the circuit network. When $K$ is not a power of 2, then partition size constraints have to be specially constructed where at some point, an intensional unbalanced partition may have to be forced to partition the circuit into, say, 2:1. Sanchis generalized the Fiduccia and Mattheyses' method with Krishnamurthy's lookahead scheme directly to $K$-way partitioning methods by handling the local moves within the $K$ partitions simultaneously. Cong, Li and Bagrodia adopted Sanchis' method to K-Way acyclic partitioning [23]. A generalization of $K$-way spectral partitioning is proposed by Chan, Schlag and Zien [20].

## 6.2 Problem Formulation for Bit-Serial Pipeline Network Partitioning for Multi-FPGA Systems

As we have discussed in Section 2.2, partitioning problem for multi-FPGA synthesis is more complex than the general partitioning problem since the ultimate goal of the multi-FPGA synthesis is not merely to partition the circuit into clusters which satisfy the size and IO capacity constraints, but to derive *routable* circuit clusters which can be physically mapped onto the FPGAs. Since FPGAs are used for wide variety of applications, the focus on this circuit partitioning problem is often set to handle general circuits using circuit benchmarks. Although this partitioning problem and the subsequent place and route problem for generalized complex circuits (sometimes characterized as *random circuits*) are very important for the whole FPGA community, we can drastically simplify the situations if we limit the class of circuitry to be handled. This is exactly what we have intended during the whole course of this thesis. We have recognized the enormous advantages of bit-serial circuits not only in the computation efficiency compared to bit-parallel circuits, but perhaps more importantly, in the physical layout synthesis problem. In this chapter, we present a very simplistic greedy multi-way partitioning algorithm which is capable of maximizing the logic utilization of each FPGA while minimizing the IO consumption. In general, it is often necessary to limit the logic utilization in order to guarantee a complete physical routing of the circuits on FPGAs since the routing resources are scarce. We simply ignore the concern about the circuit routability since bit-serial circuits are highly routable. And our confidence for the high routability is firmly supported by our experimental results, which we did not encounter an unroutable bit-serial circuit over a number of designs as we shall see later. Also, we are able to be very greedy on the multi-way partitioning problem since bit-serial circuits are extremely easy to partition. We note the readers that this type of maximum-density partitioning algorithm has not been addressed by other researchers, mainly because of the concerns for the circuit routability. Although our solution is very straightforward, our partitioning algorithm is shown to be a very powerful tool for bit-serial pipeline networks.

## 6.3 Bit-Serial Pipeline Network Multi-Way Partitioning Methodology

In this section, we will describe our simplistic partitioning scheme for bit-serial pipeline networks. As discussed in Chapter 4, we do the partitioning on the cell-level netlist $HG(\overline{E_c}, V_c)$. The reason why we do the partitioning on the cell-level netlist instead of the module-level netlist is simply because the finer granularity of the circuit creates

| designs | module-level | | | cell-level | | | logic-level | |
|---|---|---|---|---|---|---|---|---|
| | pins/net | pins/mod | CLBs/mod | pins/net | pins/cell | CLBs/cell | pins/net | pins/CLB |
| 1D-FIR30-II | 2.38 | 5.57 | 30.83 | 2.03 | 7.74 | 3.91 | 3.24 | 6.41 |
| 2D-FIR8x8 | 2.05 | 5.82 | 16.83 | 2.13 | 7.06 | 3.35 | 3.18 | 6.31 |
| 1D-IIR20-III | 2.20 | 6.67 | 31.84 | 2.02 | 10.44 | 5.32 | 3.55 | 7.00 |
| adapt10T-7 | 2.49 | 6.47 | 16.49 | 2.08 | 8.16 | 4.10 | 3.26 | 6.34 |
| IDCTmem-I | 2.29 | 6.48 | 19.57 | 2.14 | 6.77 | 4.89 | 3.18 | 5.68 |
| N-12-14-4 | 2.12 | 8.31 | 19.19 | 2.03 | 10.73 | 5.98 | 3.57 | 6.73 |

Table 6.1: Pins per net and pins per block at the module-level, cell-level and logic-level. At the module-level, multipliers, scalers and parallel-serial IO interface circuits are generally large circuits. For example, the size of an 8-bit bit-serial multiplier is 39 CLBs. On the cell-level, multipliers and scalers are partitioned into bit-serial cells. Yet, parallel-serial IO interface circuits are still treated as one cell in the bit-serial circuit library.

more chances of filling the partition bins closer to the maximum capacity. On the other hand, the reasons for not performing the circuit partitioning on the logic block-level are:

1. Bit-serial circuits have a very clear hierarchical structure in terms of connectivity. Table 6.1 shows the average pins per net and pins per block at the three hierarchies described in Section 4.7 which are the module-level, cell-level and logic-level. Global clock net is not counted in these cases. As we can see, the pins/net ratio on the cell-level netlist is extremely low, close to the minimum of 2, whereas the pins/net ratio on the logic-level is more than 3. This shows that the interconnections between cells are very sparse and we can expect the partitioning to be very easy at this level. Also, notice the high pins/CLB ratio which is close to 7. This high ratio indicates that the CLB elements are very effectively utilized. In fact, quite a few CLBs use the *direct input* which directly connects to the flip-flop inside the CLB (please refer to Fig.3.4 in Section 3.2) which would have been unused when a 5-input LUT is implemented inside the CLB. In such cases, the number of pins used at the CLB is 8, not counting the clock input.

2. The average number of CLBs per cell is around 3 to 6. The number of nodes in the graph is reduced by exactly this amount, and also the number of pins in the netlist and also the number of nets are reduced because several nets are grouped together to form a *bus* (refer to Section 4.7.4). This node clustering and net grouping contributes in reducing the problem complexity of the partitioning problem, which enables us to handle large designs over a hundred thousand gates with about 1000 cells and 4000 buses.

### 6.3.1 Basic Concept

Here, we illustrate the basic algorithm which recursively partitions the bit-serial circuits while maximizing the logic utilization of each FPGA chip.

**Pipeline network Multi-way partitioning methodology** 1. For a bit-serial pipeline network $HG(\overline{E}, V)$, let $Y_0 = V$. For $i = 1, 2, \cdots$, recursively partition the pipeline network $Y_{i-1}$ into $X_i$ and $Y_i$:

$$\begin{array}{rcll} X_i \cup Y_i & = & Y_{i-1} & (i = 0, 1, \cdots) \\ X_i \cap Y_i & = & \phi. & \end{array} \tag{6.8}$$

We call $\{X_i | i = 1, 2, \cdots\}$ the *leaf partitions* and $\{Y_i | i = 0, 1, \cdots\}$ the *residue partitions* since $Y_i$ is the residue vertices after producing $i$ leaf partitions $V = (X_1 \cup X_2 \cup \cdots \cup X_i) \cup Y_i$ (Fig.6.1).

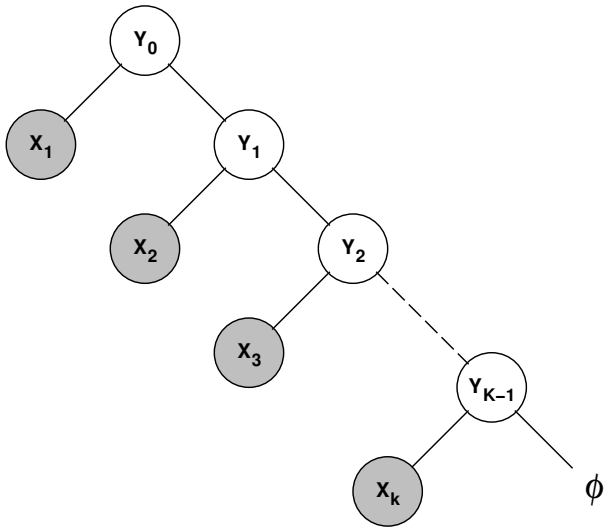



Figure 6.1: Recursive bipartitioning method to obtain a $K$-way partition. Capacity constraints are imposed only to the shaded leaf partitions.

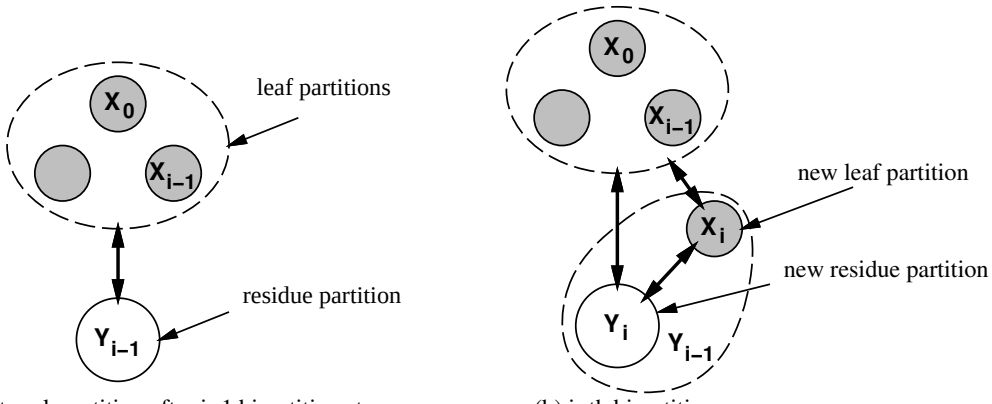



Figure 6.2: Recursive bipartition steps. (a) shows the partitioned network after $i-1$ bipartitioning steps. (b) shows the $i$-th bipartitioning which creates $X_i$ and $Y_i$ by partitioning $Y_{i-1}$.

2. Impose the size capacity and IO capacity constraints only to leaf partitions ($X_i$) which reflect the actual physical capacities of the FPGA chip.
3. Continue the procedure until $Y_{K-1}$ becomes empty. The $K$ leaf partitions essentially becomes the $K$-way partitioning solution $(X_1, X_2, \cdots X_{K-1}, X_K)$ in which all the partitions satisfy the size capacity and IO capacity constraints.

### 6.3.2 Fiduccia-Mattheyses Bipartitioning Method

For the basic bipartitioning procedure, we adopt the efficient partitioning algorithm proposed by Fiduccia and Mattheyses [38] with some modification on the constraints and initial partition. First, let us go over briefly about their approach.

1. Basic procedure : Their method is based upon the approach of the iterative bipartitioning heuristic proposed by Kernighen and Lin [60].
   (a) Compute an initial feasible partitioning of the hyper-graph $HG(\overline{E}, V)$ into $X$ and $Y$ ($X \cap Y = \phi, X \cup Y = V$) by some random process such that $X$ and $Y$ satisfies the size constraints described in eq.(6.4)(6.5).
   (b) Move a free (unlocked) vertex one at a time from one partition to the other which maximizes the *gain* (minimizes the cut cost) while satisfying the size capacity constraints and lock the vertex. Locking the vertex prevents from entering into an infinite loop. The gain may be negative, but is still allowed to move with the expectation that this move will allow the algorithm to climb out of the local minima and the subsequent moves leads to a better partition in terms of cut cost. Continue this process until no further legal move exists within the free vertices, meaning that either all the vertices are locked or the size capacity constraints prevent further moves.
   (c) Find the partition with the largest gain encountered during the pass. If the largest gain is positive, use that partition as the new initial partition, and do another pass of (b). Otherwise if the largest gain is non-positive, no improvement in the cut cost is observed and therefore the bipartitioning procedure is terminated.
2. Implementation : There are a couple of key points in which they managed to speed up the bipartitioning procedure.
   (a) *Vertex gain calculation* : In evaluating the gain of moving a vertex $v$, they have categorized each undirected hyper-edge $\overline{e} = (v_0, v_1, \cdots, v_{k-1}) \in \overline{E}$ in four groups:
   - $external(X)$ : A set of hyper-edges $\overline{e} \in \overline{E}$ which have only one terminal vertex $v$ in $X$. By moving $v$ to $Y$, this hyper-edge no longer crosses the cut, and therefore the partition cut cost decrease by $cost(\overline{e})$ (a gain of $cost(\overline{e})$).
   - $internal(X)$ : A set of hyper-edges $\overline{e} \in \overline{E}$ which have all the terminal vertices in $X$, therefore currently does not cross the cut. By moving any of these vertices to $Y$, this hyper-edge now crosses the cut, and therefore the partition cut cost increases by $cost(\overline{e})$ (a gain of $-cost(\overline{e})$). One note here is that if there are only one terminal vertex in $\overline{e}$ which happens to be included in $X$, $\overline{e}$ is included in both $external(X)$ and $internal(X)$ in which case the moving cost of the terminal vertex cancels out to be 0.
   - $external(Y)$ : A set of hyper-edges which have only one terminal vertex in $Y$. Moving this vertex to $X$ decreases the cut post by $cost(\overline{e})$.
   - $internal(Y)$ : A set of hyper-edges which have all the terminal vertices in $Y$. Moving any of these vertices increases the cut cost by $cost(\overline{e})$.

The moving gain (opposite of cost) of a vertex $v \in V$ can now be calculated as

$$gain(v) = \begin{cases} \sum_{\overline{e} \in external(X), v \in \overline{e}} cost(\overline{e}) - \sum_{\overline{e} \in internal(X), v \in \overline{e}} cost(\overline{e}) & (v \in X) \\ \sum_{\overline{e} \in external(Y), v \in \overline{e}} cost(\overline{e}) - \sum_{\overline{e} \in internal(Y), v \in \overline{e}} cost(\overline{e}) & (v \in Y) \end{cases} . \quad (6.9)$$

The gain of each vertex is first calculated at the beginning of each pass which is shown to take $O(P)$ time where $P$ is the number of pins in the hyper-graph $HG(\overline{E}, V)$. During a pass, each time a vertex is moved from one partition to the other, the gains of the vertices which are connected to it changes and therefore are recomputed. They have shown that the total amount of work needed to update all the vertex gains during a pass is $O(P)$ as well.

(b) Bucket list structure for fast maximum gain vertex search : In order to speedup the process of searching the vertex with the maximum gain, they have implemented a bucket structure in which each bucket points to a doubly-linked list of free (unlocked) vertices having the same gain. Note that as long as $cost(\overline{e})$ can be expressed in an integer form, only a finite number of buckets are needed. In particular, we can easily show that we only need $2p_{max} + 1$ buckets where

$$p_{max} = max(\{ \sum_{\overline{e} \in \overline{E}, v \in \overline{e}} cost(\overline{e}) | v \in V \}). \quad (6.10)$$

There are two set of buckets, one for $X$ and one for $Y$. The bucket structure is maintained by moving a vertex to the appropriate bucket whenever its gain changes due to the movement of one of its neighbor vertices. There is also a *maxgain* index keeping track of the bucket having a vertex of the highest gain. This index is updated by decrementing it whenever its bucket is found to be empty and resetting it to a higher bucket whenever a vertex moves to a bucket above *maxgain*. The total amount of work needed to construct and maintain the bucket structure is shown to be $O(P)$ as well.

Their effort in the implementation of the algorithm made the running time of the algorithm to $O(P)$ per pass where $P$ is the total number of pins.

### 6.3.3 Details of the Recursive Bipartitioning Method

Here, we describe the modifications to the Fiduccia-Mattheyses' bipartitioning method in order to maximize the logic utilization.

1. Input : Pipeline network $HG(\overline{E_c}, V_c)$, and the previous residue partition $Y'$.
2. Basic procedure :
   (a) Let $X = \phi$, $Y = Y'$ be the initial bipartitioning solution.
   (b) Move a free vertex one at a time from one partition to the other which minimizes the IO count of $Y$ ($|Y|_{IO}$) and satisfying the size capacity and IO capacity constraints on $X$:

$$\begin{array}{rcl} |X| & \leq & F_s \\ |X|_{IO} & \leq & F_{IO} \end{array} \quad (6.11)$$

Where $F_s$ and $|F|_{IO}$ represent the size capacity and IO capacity fo the FPGA chip, respectively. Lock the vertex after moving. Continue this process until no further legal move exists within the free vertices.

(c) Find the partition which minimizes the following function:

$$C = |Y| + \alpha \cdot |Y|_{IO} + \beta \cdot |X|_{IO} \tag{6.12}$$

where $|Y|_{IO}$ is the IO count of $Y$ and $|X|_{IO}$ is the IO count of $X$. $\alpha$ and $\beta$ are predefined weights.

3. Modifications from Fiduccia and Mattheyses' method :

(a) Fiduccia-Mattheyses' method does not take in account the IO pins of the circuit netlist $HG(\overline{E}, V)$ connecting to the external world, and only evaluate the partition cut cost. Since FPGAs have strict IO capacity constraint, we have to accurately evaluate the IO count for each partition. Therefore, we include these *external* connections in our partition problem formulation.

(b) As we have described earlier, in partitioning the network into a leaf partition and a residue partition, size capacity and IO capacity constraints are only applied to the leaf partition. Therefore, there are no balancing criterion of the partition sizes, which implies that a bipartitioning solution where the leaf partition has no vertex is still a feasible one. We use this as our initial bipartitioning solution, and do not rely on random process to determine the initial partition.

(c) There are three major objectives in our bipartitioning problem:

i. Minimize the size of the residue partition ($|Y|$).
ii. Minimize the IO count of the residue partition ($|Y_{IO}|$).
iii. Minimize the IO count of the leaf partition ($|X|_{IO}$).

Minimizing the size of the residue partition in return means maximizing the size of the leaf partition under the size capacity constraint. Note that the IO count of the residue partition includes not only the connections between the current leaf partition $X$ and residue partition $Y$ but also the connections between the residue partition and the previous leaf partitions $X_0, X_1, \cdots, X_{i-1}$. The latter becomes essentially the external connections for the problem of partitioning $Y'$ into $X$ and $Y$, which would have been ignored by the conventional bipartitioning problem formulation as we have discussed earlier. The first two criteria are equally important in solving the $K$-way partitioning problem successfully. The third criterion is not as important as the previous two since we only need to obtain a feasible leaf partition which satisfies the IO capacity constraint, and minimizing the IO count of the leaf partition is a secondary issue which does not affect the following bipartitioning steps. Our experience shows that the values $\alpha = 0.5$, $\beta = 0.25$ works well for all cases.

(d) Gain calculation for each vertex is done so that the vertex gain only reflects the decrease in the IO count of the residue partition $Y$. Since it is not the partition cut cost as in Fiduccia and Mattheyses' case, we have to categorize the directed hyper-edges representing the external connections. in the same manner and including them in calculating the accurate vertex gain. Let $\overline{E_{ext}}$ be a set of hyper-edges which are connected to a vertex not included in either the leaf partition $X$ or the residue partition $Y$. In addition to the four categories describe earlier, we also provide four similar categories for the *external* hyper-edges:

- $internal'(X)$ : A set of hyper-edges $\overline{e} \in \overline{E_{ext}}$ which have some of the terminal vertices in $X$ and none in $Y$. By moving any of these vertices to $Y$, this external hyper-edge makes a connection to $Y$, and therefore $|Y|_{IO}$ increases by $cost(\overline{e})$.
- $external'(X)$ : A set of hyper-edges $\overline{e} \in \overline{E_{ext}}$ which have only one terminal vertex in $X$. By moving this vertex to $Y$, the external hyper-edge no longer has connection to $X$, and therefore $|X|_{IO}$ decreases by $cost(\overline{e})$. One note here is that if there is one terminal vertex in $X$ and none in $X$, $\overline{e}$ is included in both $internal'(X)$ and $external'(X)$, and moving this vertex from $X$ to $Y$ increases $|Y|_{IO}$ by $cost(\overline{e})$ and decreases $|X|_{IO}$ by $cost(\overline{e})$.

| chip | CLBs | CLB util. | IOs | IO util. | chip | CLBs | CLB util. | IOs | IO util. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 478 | 98.8% | 66 | 37.5% | 8 | 481 | 99.4% | 47 | 26.7% |
| 1 | 482 | 99.6% | 62 | 35.2% | 9 | 481 | 99.4% | 40 | 22.7% |
| 2 | 484 | 100% | 61 | 34.7% | 10 | 482 | 99.6% | 39 | 22.2% |
| 3 | 483 | 99.8% | 56 | 31.8% | 11 | 484 | 100% | 35 | 19.9% |
| 4 | 484 | 100% | 54 | 30.7% | 12 | 481 | 99.4% | 34 | 19.3% |
| 5 | 484 | 100% | 54 | 30.7% | 13 | 484 | 100% | 31 | 17.6% |
| 6 | 482 | 99.6% | 54 | 30.7% | 14 | 210 | 43.4% | 23 | 13.7% |
| 7 | 484 | 100% | 53 | 30.1% | | | | | |
| mean | 464 | 95.9% | 47 | 26.7% | max | 484 | 100% | 66 | 37.5% |

Table 6.2: Partitioning results of 12-14-4 digital neural network (N-12-14-4) on XC3195A FPGAs.

- $internal'(Y)$ : A set of hyper-edges $\overline{e} \in \overline{E_{ext}}$ which have some of the terminal vertices in $Y$ and none in $X$. By moving any of these vertices to $X$, this external hyper-edge makes a connection to $X$, and therefore $|X|_{IO}$ increases by $cost(\overline{e})$.
- $external'(Y)$ : A set of hyper-edges $\overline{e} \in \overline{E_{ext}}$ which have only one terminal vertex in $Y$. By moving this vertex to $X$, the external hyper-edge no longer has connection to $Y$, and therefore $|Y|_{IO}$ decreases by $cost(\overline{e})$.

Out of these four categories, $internal'(X)$ and $external'(Y)$ contribute to the change in the IO count of the residue partition. The moving gain of a vertex $v \in V_c$ is now calculated as

$$gain(v) = \begin{cases} \sum_{\overline{e} \in external(X), v \in \overline{e}} cost(\overline{e}) - \sum_{\overline{e} \in internal(X), v \in \overline{e}} cost(\overline{e}) - \sum_{\overline{e} \in internal'(X), v \in \overline{e}} cost(\overline{e}) & (v \in X) \\ \sum_{\overline{e} \in external(Y), v \in \overline{e}} cost(\overline{e}) - \sum_{\overline{e} \in internal(Y), v \in \overline{e}} cost(\overline{e}) + \sum_{\overline{e} \in external'(Y), v \in \overline{e}} cost(\overline{e}) & (v \in Y) \end{cases} . \quad (6.13)$$

4. Algorithm running time: As with the conventional Fiduccia-Mattheyses algorithm, our single bipartitioning step takes $O(P_i)$ time where $P_i$ is the number of pins in the residue partition $Y_i$ at step $i$. Since the number of pins effectively decreases by roughly $\frac{P}{K}$ where $P$ is the total number of pins and $K$ is the number of partitions, $P_i$ can be estimated as $P_i = \frac{P(K-1)}{K}$. Also, we observe that a typical bipartition run performs one to four passes, independent of the size constraint and the circuit size, so we can assume that one bipartitioning step also takes $O(P)$ time. Therefore, the total running time for the $K$-way partitioning is $O(\frac{P(K-1)}{2}) = O(PK)$.

## 6.4 Multi-Way Bit-Serial Partitioning Results

In this section, we show the partitioning results for the bit-serial pipeline networks described in Chapter 5. The size capacity and the IO capacity constraints are derived from the four FPGA devices from Xilinx:

- XC3195A: 484 CLBs, 176 IOs.
- XC3190A: 320 CLBs, 144 IOs.
- XC3164A: 224 CLBs, 120 IOs.
- XC3142A: 144 CLBs, 96 IOs.

| chip | CLBs | CLB util. | IOs | IO util. | chip | CLBs | CLB util. | IOs | IO util. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 318 | 99.4% | 61 | 42.4% | 11 | 319 | 99.7% | 40 | 27.8% |
| 1 | 318 | 99.4% | 60 | 41.6% | 12 | 320 | 100% | 39 | 27.1% |
| 2 | 318 | 99.4% | 60 | 41.6% | 13 | 318 | 99.4% | 39 | 27.1% |
| 3 | 318 | 99.4% | 50 | 34.7% | 14 | 318 | 99.4% | 37 | 25.7% |
| 4 | 318 | 99.4% | 48 | 33.3% | 15 | 318 | 99.4% | 37 | 25.7% |
| 5 | 317 | 99.1% | 48 | 33.3% | 16 | 320 | 100% | 35 | 24.3% |
| 6 | 320 | 100% | 43 | 29.9% | 17 | 316 | 98.8% | 34 | 23.6% |
| 7 | 317 | 99.1% | 42 | 29.2% | 18 | 318 | 99.4% | 33 | 22.9% |
| 8 | 315 | 98.4% | 41 | 28.5% | 19 | 318 | 99.4% | 29 | 20.1% |
| 9 | 319 | 99.7% | 40 | 27.8% | 20 | 319 | 99.7% | 27 | 18.8% |
| 10 | 319 | 99.7% | 40 | 27.8% | 21 | 283 | 88.4% | 22 | 15.3% |
| mean | 316 | 98.9% | 41 | 28.5% | max | 320 | 100% | 61 | 42.4% |

Table 6.3: Partitioning results of 12-14-4 digital neural network (N-12-14-4) on XC3190A FPGAs.

| chip | CLBs | CLB util. | IOs | IO util. | chip | CLBs | CLB util. | IOs | IO util. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 224 | 100% | 66 | 40.0% | 16 | 224 | 100% | 35 | 35.8% |
| 1 | 221 | 98.7% | 51 | 35.8% | 17 | 224 | 100% | 35 | 22.5% |
| 2 | 224 | 100% | 48 | 26.7% | 18 | 221 | 98.7% | 34 | 19.2% |
| 3 | 223 | 99.6% | 48 | 29.2% | 19 | 220 | 98.2% | 33 | 30.8% |
| 4 | 221 | 98.7% | 44 | 42.5% | 20 | 221 | 98.7% | 32 | 27.5% |
| 5 | 220 | 98.2% | 44 | 33.3% | 21 | 224 | 100% | 31 | 55.0% |
| 6 | 221 | 98.7% | 43 | 22.5% | 22 | 224 | 100% | 31 | 36.4% |
| 7 | 221 | 98.7% | 43 | 24.2% | 23 | 224 | 100% | 31 | 22.5% |
| 8 | 224 | 100% | 43 | 35.8% | 24 | 222 | 99.1% | 29 | 25.8% |
| 9 | 220 | 98.2% | 40 | 28.3% | 25 | 222 | 99.1% | 29 | 24.2% |
| 10 | 223 | 99.6% | 40 | 32.5% | 26 | 222 | 99.1% | 27 | 40.0% |
| 11 | 224 | 100% | 39 | 22.5% | 27 | 222 | 99.1% | 27 | 25.8% |
| 12 | 223 | 99.6% | 38 | 29.2% | 28 | 224 | 100% | 27 | 30.0% |
| 13 | 224 | 100% | 37 | 36.4% | 29 | 220 | 98.2% | 27 | 31.7% |
| 14 | 221 | 98.7% | 36 | 25.8% | 30 | 222 | 99.1% | 23 | 29.2% |
| 15 | 224 | 100% | 35 | 33.3% | 31 | 70 | 31.3% | 19 | 15.8% |
| mean | 217 | 97.2% | 36 | 30.0% | max | 224 | 100% | 66 | 55.0% |

Table 6.4: Partitioning results of 12-14-4 digital neural network (N-12-14-4) on XC3164A FPGAs.

| chip | CLBs | CLB util. | IOs | IO util. | chip | CLBs | CLB util. | IOs | IO util. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 144 | 100% | 57 | 59.4% | 25 | 141 | 97.9% | 28 | 29.2% |
| 1 | 144 | 100% | 44 | 45.8% | 26 | 144 | 100% | 27 | 28.1% |
| 2 | 140 | 97.2% | 44 | 45.8% | 27 | 144 | 100% | 27 | 28.1% |
| 3 | 144 | 100% | 43 | 44.8% | 28 | 143 | 99.3% | 26 | 27.1% |
| 4 | 143 | 99.3% | 43 | 44.8% | 29 | 143 | 99.3% | 26 | 27.1% |
| 5 | 144 | 100% | 40 | 41.7% | 30 | 143 | 99.3% | 26 | 27.1% |
| 6 | 141 | 97.9% | 36 | 37.5% | 31 | 142 | 98.6% | 25 | 26.0% |
| 7 | 144 | 100% | 35 | 36.5% | 32 | 140 | 97.2% | 25 | 26.0% |
| 8 | 141 | 97.9% | 34 | 35.4% | 33 | 142 | 98.6% | 25 | 26.0% |
| 9 | 143 | 99.3% | 34 | 35.4% | 34 | 142 | 98.6% | 25 | 26.0% |
| 10 | 141 | 97.9% | 32 | 33.3% | 35 | 142 | 98.6% | 25 | 26.0% |
| 11 | 143 | 99.3% | 32 | 33.3% | 36 | 142 | 98.6% | 25 | 26.0% |
| 12 | 141 | 97.9% | 32 | 33.3% | 37 | 142 | 98.6% | 25 | 26.0% |
| 13 | 143 | 99.3% | 32 | 33.3% | 38 | 144 | 100% | 25 | 26.0% |
| 14 | 144 | 100% | 32 | 33.3% | 39 | 144 | 100% | 24 | 25.0% |
| 15 | 142 | 98.6% | 32 | 33.3% | 40 | 144 | 100% | 23 | 24.0% |
| 16 | 144 | 100% | 31 | 32.3% | 41 | 144 | 100% | 23 | 24.0% |
| 17 | 144 | 100% | 31 | 32.3% | 42 | 144 | 100% | 23 | 24.0% |
| 18 | 144 | 100% | 31 | 32.3% | 43 | 141 | 97.9% | 23 | 24.0% |
| 19 | 140 | 97.2% | 31 | 32.3% | 44 | 144 | 100% | 23 | 24.0% |
| 20 | 144 | 100% | 31 | 32.3% | 45 | 112 | 77.8% | 23 | 24.0% |
| 21 | 144 | 100% | 31 | 32.3% | 46 | 144 | 100% | 22 | 22.9% |
| 22 | 142 | 98.6% | 29 | 30.2% | 47 | 143 | 99.3% | 22 | 22.9% |
| 23 | 144 | 100% | 28 | 29.2% | 48 | 143 | 99.3% | 20 | 20.8% |
| 24 | 141 | 97.9% | 28 | 29.2% | | | | | |
| mean | 142 | 98.7% | 29 | 30.8% | max | 144 | 100% | 57 | 59.4% |

Table 6.5: Partitioning results of 12-14-4 digital neural network (N-12-14-4) on XC3142A FPGAs.

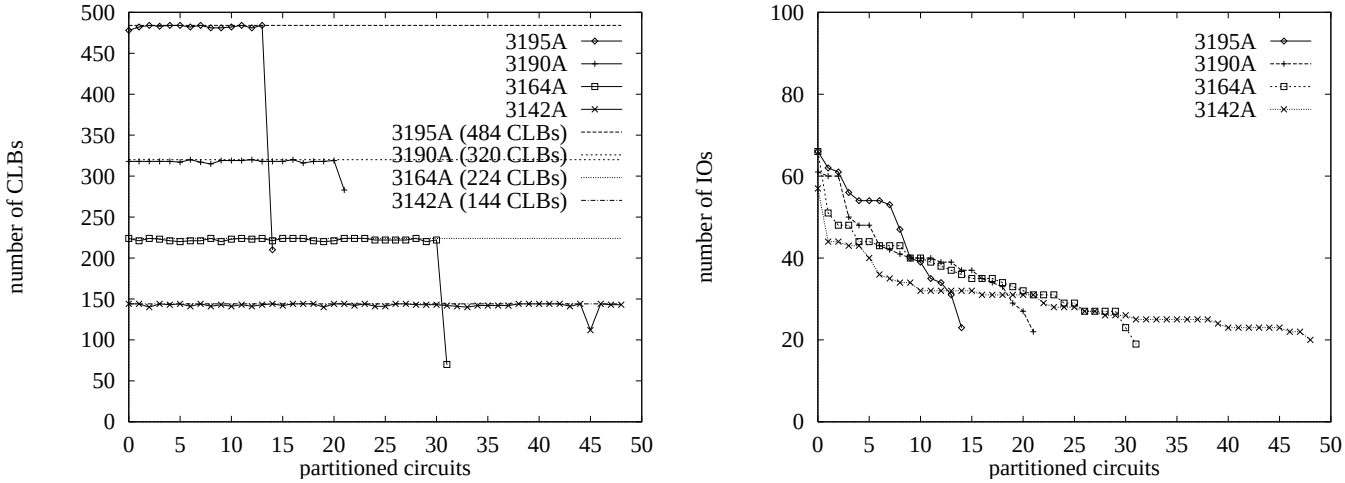



Figure 6.3: Partitioning of 12-14-4 neural network (N12-14-4) for different FPGAs. The $x$-axis corresponds to the circuit partitions, where they are sorted by the number of IOs.

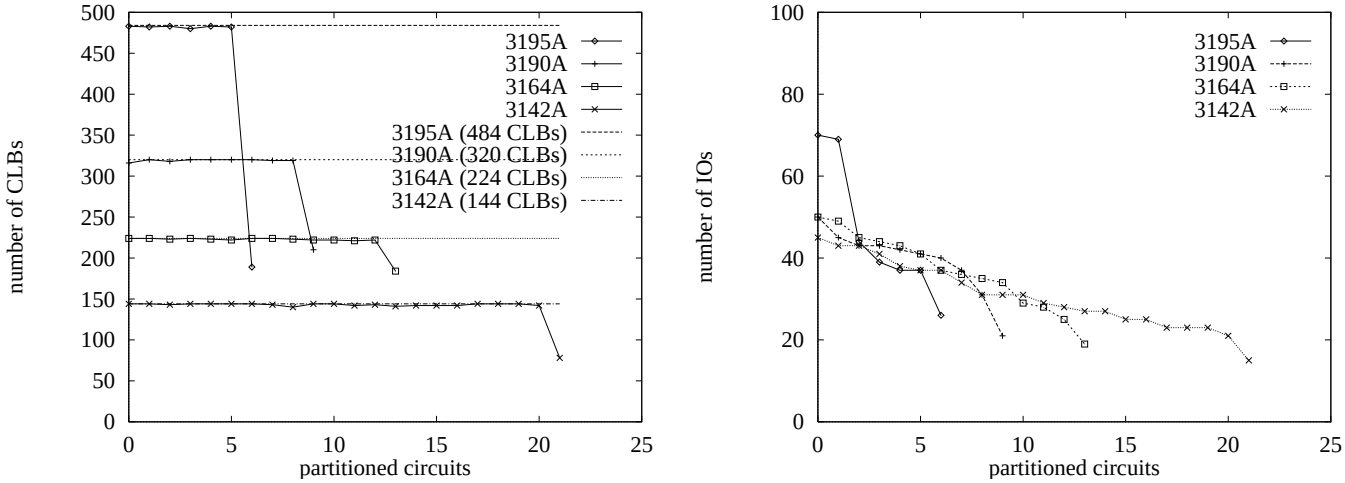



Figure 6.4: Partitioning of 8-8-4 neural network (N8-8-4) for different FPGAs.

Table 6.6: Partitioning results for bit-serial pipeline networks.

| design | 3195A (484 CLBs, 176 IOs | | | | 3190A (320 CLBs, 144 IOs | | | |
|---|---|---|---|---|---|---|---|---|
| | # of chips | CLB util. | IO util. | CPU time (sec) | # of chips | CLB util. | IO util. | CPU time (sec) |
| N-12-14-4 | 15 | 95.9% | 26.9% | 15 | 22 | 98.9% | 28.5% | 22 |
| N-8-8-4 | 7 | 91.0% | 26.1% | 6 | 10 | 96.3% | 27.3% | 8 |
| 1D-FIR30-II | 5 | 82.8% | 12.3% | 4 | 7 | 89.5% | 15.3% | 5 |
| 2D-FIR8x8 | 5 | 93.9% | 19.7% | 5 | 9 | 88.8% | 20.2% | 6 |
| 1D-IIR20-III | 5 | 90.8% | 19.5% | 5 | 7 | 98.1% | 21.3% | 5 |
| adapt10T-7 | 2 | 90.3% | 20.7% | 5 | 3 | 91.0% | 29.7% | 5 |
| IDCT-I | 3 | 69.7% | 21.6% | 5 | 4 | 79.1% | 26.9% | 5 |
| IDCTmem-I | 3 | 84.9% | 32.2% | 3 | 4 | 96.3% | 35.3% | 3 |
| design | 3164A (224 CLBs, 120 IOs) | | | | 3142A (144 CLBs, 96 IOs) | | | |
| | # of chips | CLB util. | IO util. | CPU time (sec) | # of chips | CLB util. | IO util. | CPU time (sec) |
| N-12-14-4 | 32 | 97.2% | 30.3% | 20 | 49 | 98.7% | 30.8% | 30 |
| N-8-8-4 | 14 | 98.3% | 30.7% | 10 | 22 | 97.3% | 32.1% | 9 |
| 1D-FIR30-II | 9 | 99.4% | 17.4% | 4 | 14 | 99.4% | 21.7% | 6 |
| 2D-FIR8x8 | 11 | 92.2% | 21.1% | 5 | 16 | 98.6% | 24.3% | 6 |
| 1D-IIR20-III | 10 | 98.1% | 22.9% | 6 | 16 | 95.4% | 24.8% | 7 |
| adapt10T-7 | 4 | 97.5% | 27.3% | 5 | 7 | 86.7% | 32.1% | 5 |
| IDCT-I | 5 | 90.4% | 23.7% | 5 | 8 | 87.8% | 27.2% | 6 |
| IDCTmem-I | 6 | 91.7% | 36.0% | 3 | 9 | 95.1% | 40.0% | 3 |

Tables 6.2, 6.3, 6.4, 6.5 and Fig.6.3 show the partitioning results for the 12-14-4 digital neural network using the four FPGA devices. First of all, we see the high capability of our multi-way partitioner in producing a very high logic utilization partitioning with a low IO utilization for a variety of size constraints. Average logic utilization for the four different partitioning results are all above 95%. We have no control over the logic utilization of the last leaf partition (the largest chip # in the tables) since it is the residue of the previous partition, and therefore this does not reflect the true ability of our partitioner. If we exclude the last leaf partition, the logic utilization, the effective logic utilization becomes more than 99%. Fig.6.4 shows the partitioning results for the 8-8-4 digital neural network. Other partitioning results are given in the subsequent figures for "1D-FIR30-II" (30-tap type-II 1D FIR filter), "2D-FIR8x8" (8x8-tap 2D FIR filter), "1D-IIR10-III" (10-tap type-III 1D IIR filter), and "adapt10T-7" (10-tap adaptive filter with adder-tree and 7 feedback delays). All the results are summarized in Table 6.6.

## 6.5 Routability Analysis for Bit-Serial Circuits

### 6.5.1 Rent's Exponent for Bit-Serial Circuits

In analyzing the circuit structure to evaluate the area required for VLSI layout, Rent's rule is commonly used [44][32][64]. This rule defines the relationship between the average number of pins and the average number of logic circuits in a subcircuit. It is expressed as

$$P = k \cdot G^{\gamma} \tag{6.14}$$

where $P$ is the average number of external pins in a subcircuit, and $G$ is the average number of modules in a subcircuit. $k$ is the *Rent's constant* which has empirically been found to correspond to the average number of pins

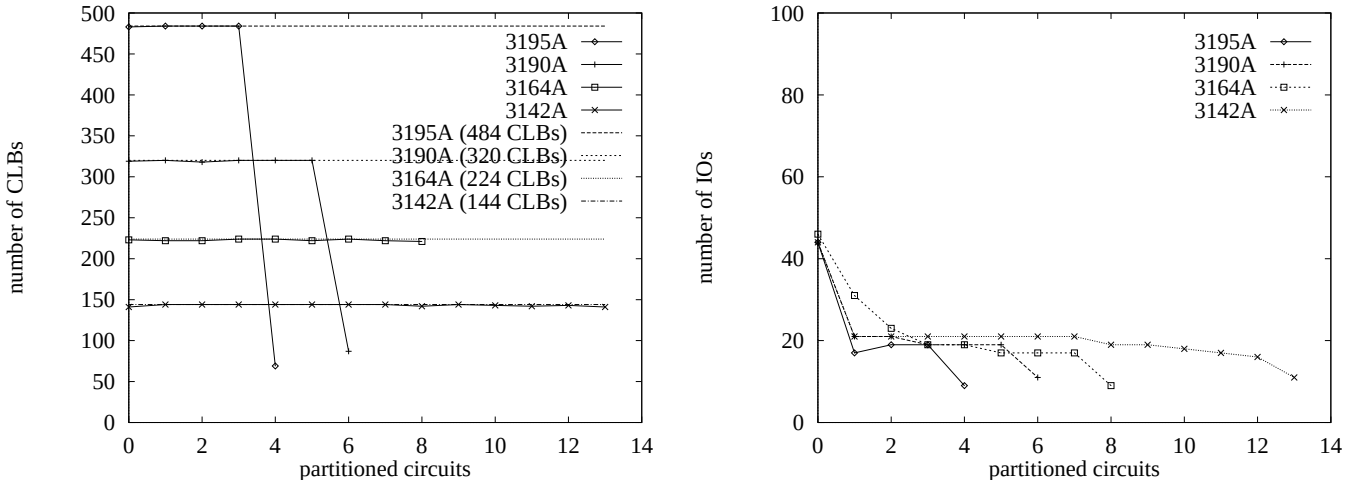



Figure 6.5: Partitioning of 30-tap 1D FIR filter (1D-FIR30-II) for different FPGAs.

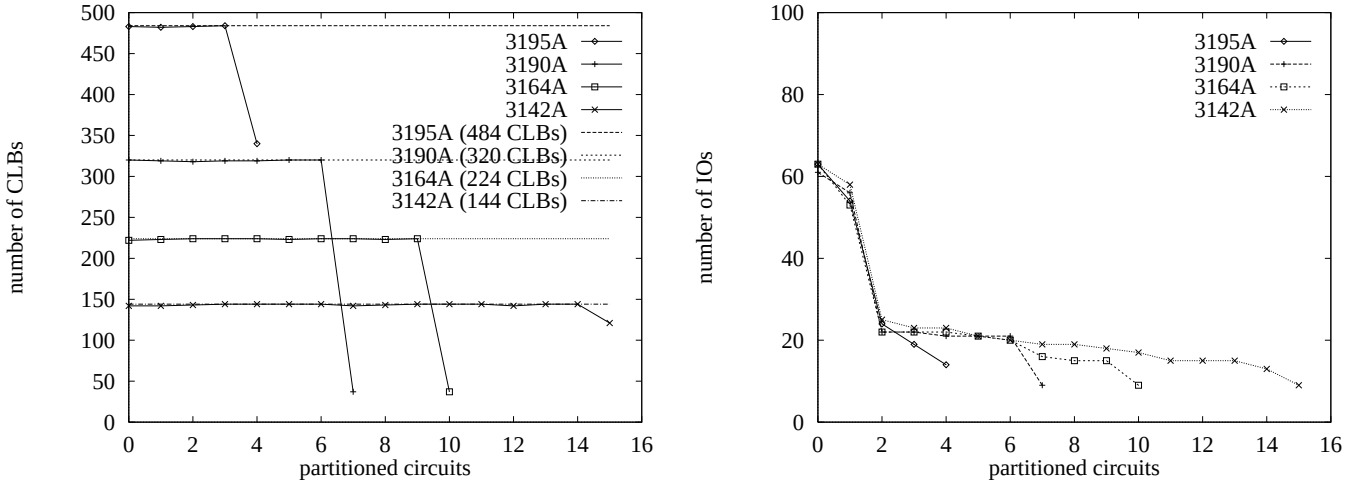



Figure 6.6: Partitioning of 8x8-tap 2D FIR filter (2D-FIR8x8) for different FPGAs.

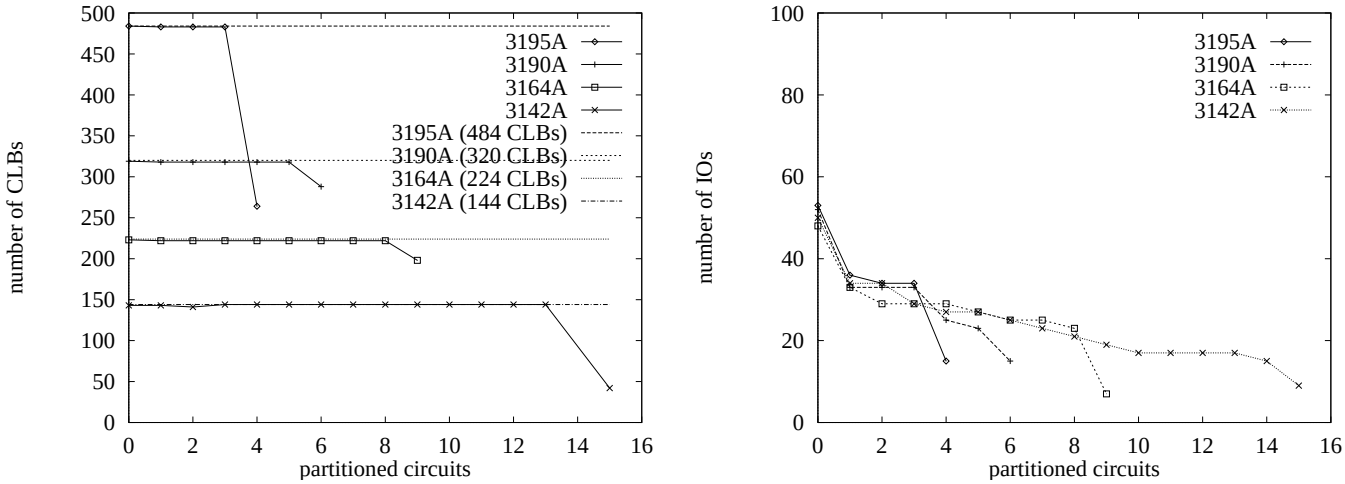



Figure 6.7: Partitioning of 20-tap 1D IIR filter (1D-IIR20-III) for different FPGAs.

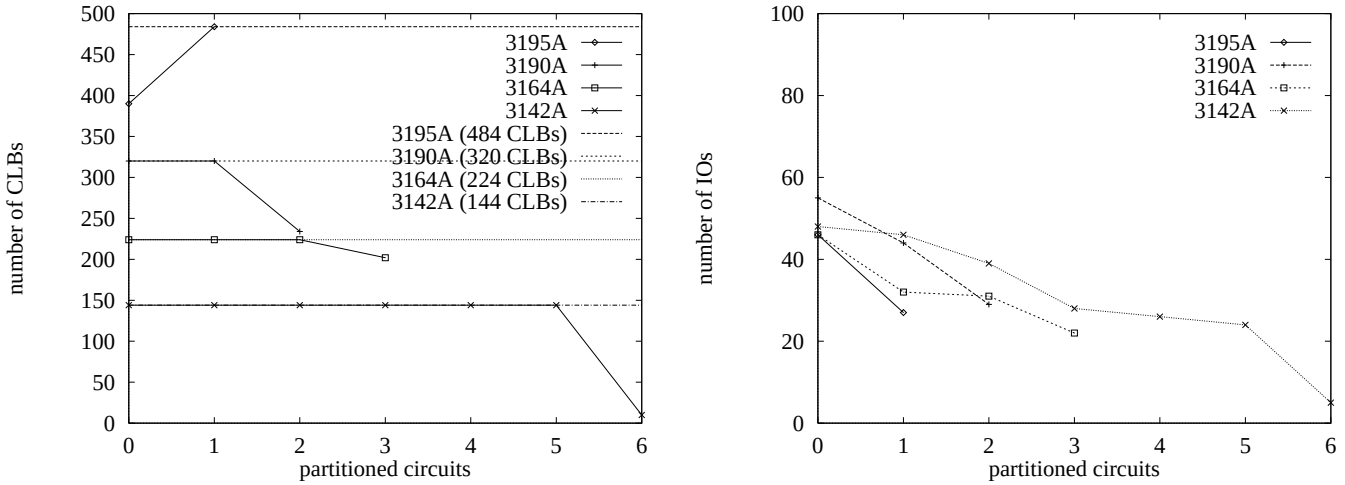



Figure 6.8: Partitioning of 10-tap 1D adaptive FIR filter (adapt10T-7) for different FPGAs.

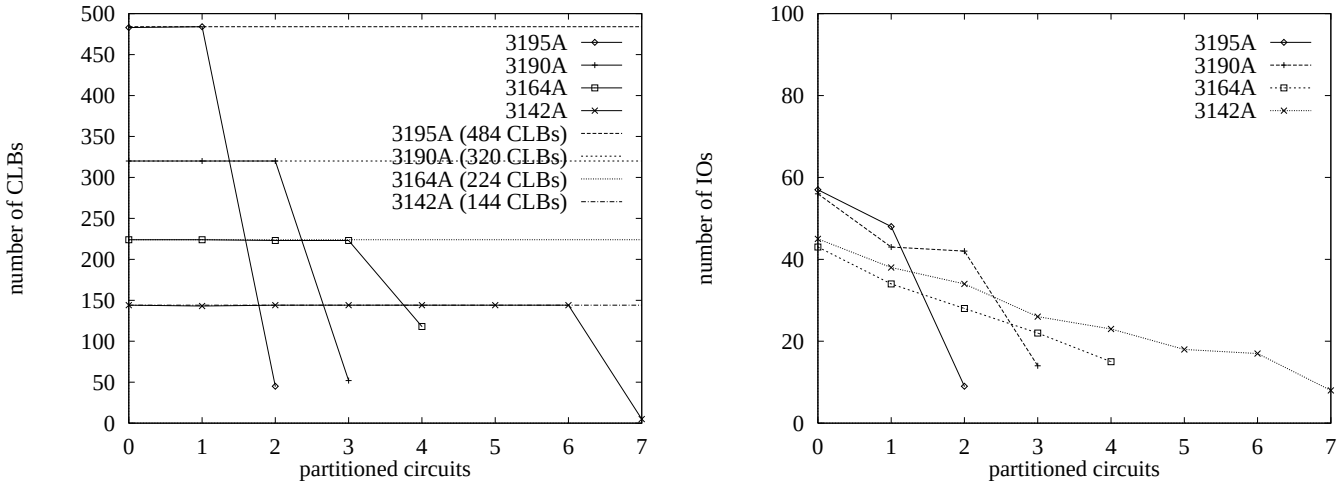



Figure 6.9: Partitioning of IDCT-I for different FPGAs.

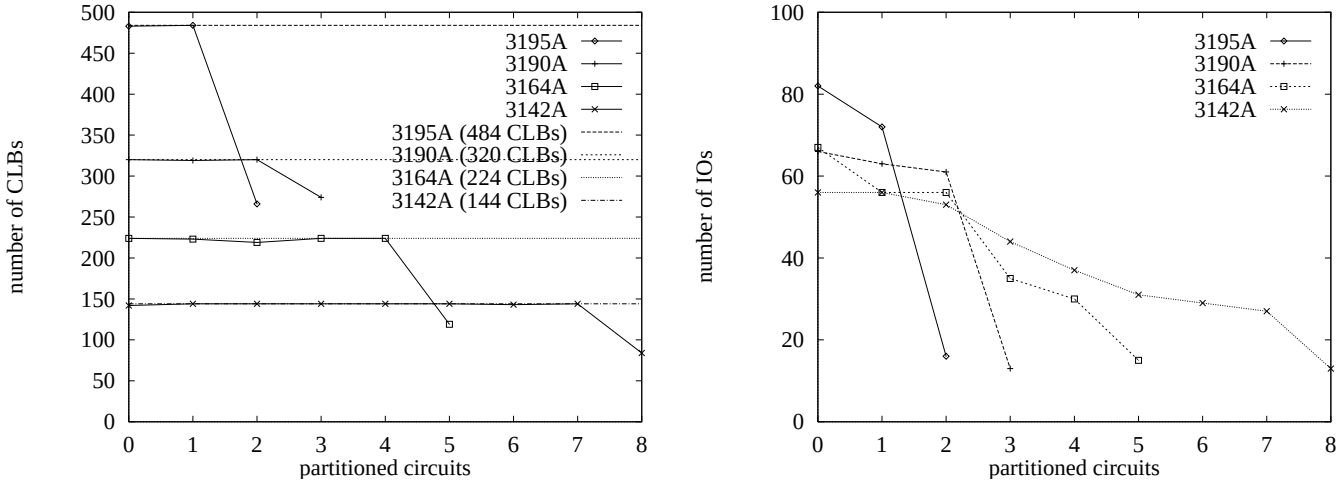



Figure 6.10: Partitioning of IDCTmem-I for different FPGAs.

Table 6.7: Rent's parameters for bit-serial circuits.

| design | $\gamma$ | $k$ |
|---|---|---|
| N-12-14-4 | 0.3724 | 4.916 |
| N-8-8-4 | 0.3264 | 5.878 |
| 1D-FIR30-II | 0.2218 | 6.018 |
| 2D-FIR8x8 | 0.3245 | 4.665 |
| 1D-IIR20-III | 0.3264 | 5.878 |
| adapt10T-7 | 0.3735 | 5.150 |
| IDCT-I | 0.3394 | 5.493 |

per module. $\gamma$ is the *Rent's exponent* which ranges between 0 and 1. First discovered by E. F. Rent of IBM in the late 1960's, Donath [32] derived the same relationship from a stochastic model of hierarchical design process. From his model which assumes a two-dimensional layout, he has derived the average wire length $r$ of the circuit using the Rent's parameters as

$$r \sim \begin{cases} f(\gamma) & (\gamma < 0.5) \\ \log G & (\gamma = 0.5) \\ G^{\gamma - 0.5} & (\gamma > 0.5) \end{cases} \tag{6.15}$$

where $f(\gamma)$ is a function independent of $G$. The intuitive explanation for this is that the case $\gamma = 0.5$ is the transition between planar and non-planar circuits, and the circuits whose Rent's exponent is lower than 0.5 can be placed such that all connections essentially lie between nearest neighbors with an average wire length being independent of $G$. For circuits where $\gamma > 0.5$, the wire length grows exponentially with $G$. Therefore, the Rent's exponent is a good indicator for estimating the amount of routing resources needed for the physical layout. Landman and Russo [64] performed an extensive study on large "real-life" circuits, and observed the Rent's exponent to be between 0.47 and 0.75. For FPGA layout synthesis, the Rent's exponent is very critical both in IO pin consumption and routability.

We have obtained the Rent's parameters for our bit-serial designs by applying multi-way partitioning with different size constraints. The method for deriving Rent's parameters is to plot the average CLB counts and the average IO counts per partition on the log-log scale and use linear regression to estimate the slope and the intercept. Note that Rent's rule on log-log scale forms a linear equation $\log P = \gamma \log G + \log k$. Figures 6.11 through 6.14 shows the results. The Rent's parameters are summerized in Table 6.7. Compared to Landman and Russo's work where they reported the Rent's exponent to be between 0.47 and 0.75, our bit-serial circuits have a significantly lower Rent's exponent between 0.22 and 0.37. This implies that our bit-serial circuits are very highly routable circuits on two-dimensional plane. There have not been any studies on logic circuits which reveal such a low Rent's exponent for *real* applications. Our bit-serial circuit design and our bit-serial pipeline network synthesis strategy led to such an extremely routing-efficient circuit structure. Since FPGAs are tuned to implement "hard-to-route" real circuits having high Rent's exponent of over 0.5, we can expect that lack of routing resource on FPGAs for our bit-serial circuits is unlikely to cause any problem, which will be strongly confirmed in our physical layout results described next.

### 6.5.2 Physical Layout Results

We have used `ppr` which is an automatic placement and routing tool from Xilinx for the physical layout of the bit-serial circuits. `ppr` uses a Generalized Force-Directed Relaxation (GFDR) algorithm instead of simulated annealing which is used in `apr` which is their older place and route tool [89]. Dispite the difference in the algorithm, both

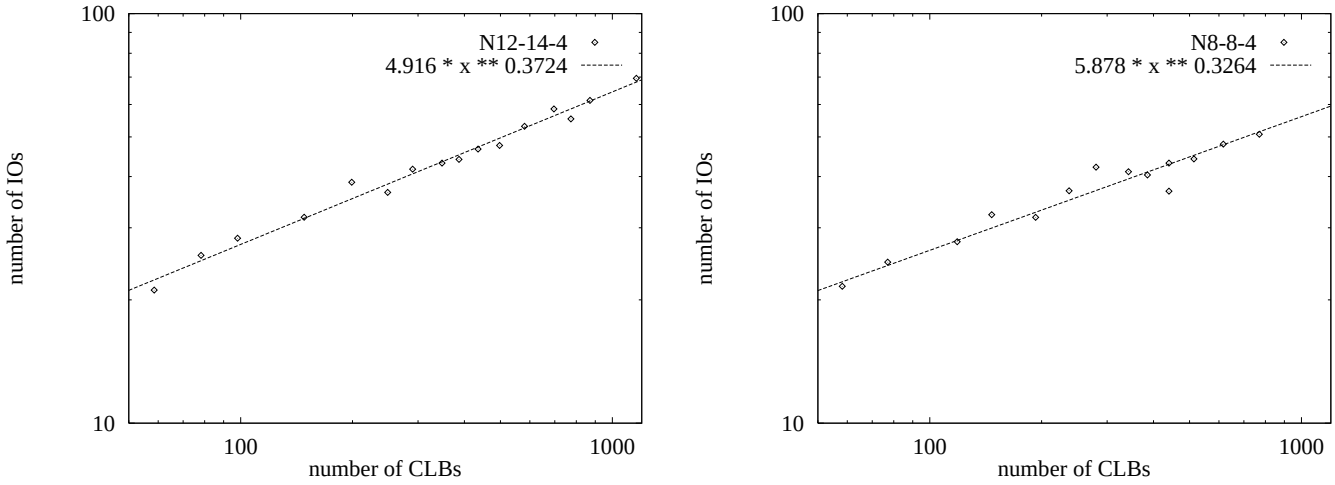



Figure 6.11: Rent's exponent approximation for N12-14-4 and N8-8-4. The two axes are log-scale.

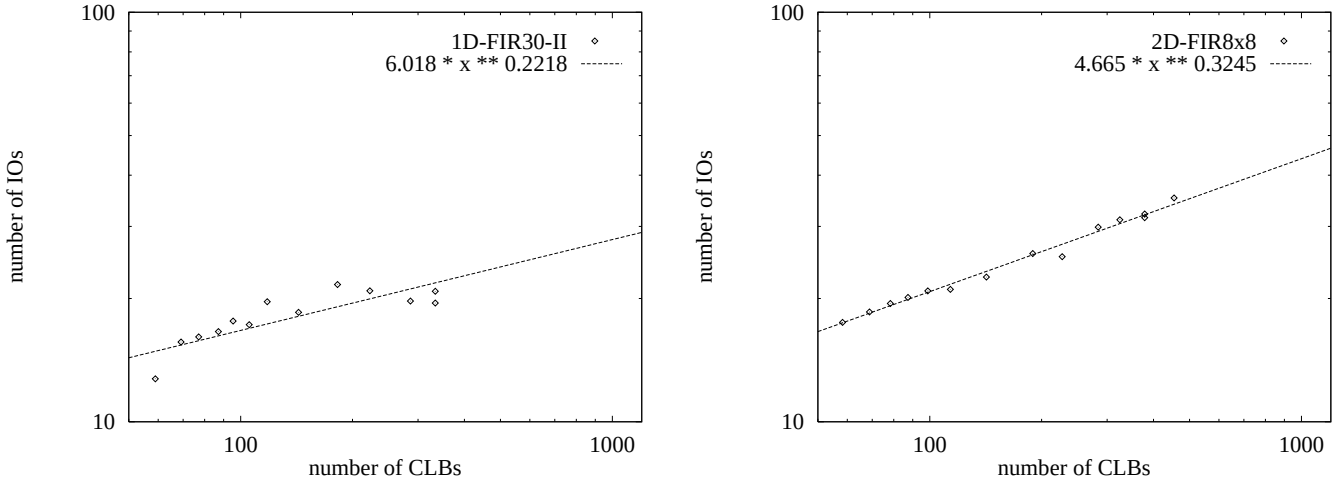



Figure 6.12: Rent's exponent approximation for 1D-FIR30-II and 2D-FIR8x8. The two axes are log-scale.

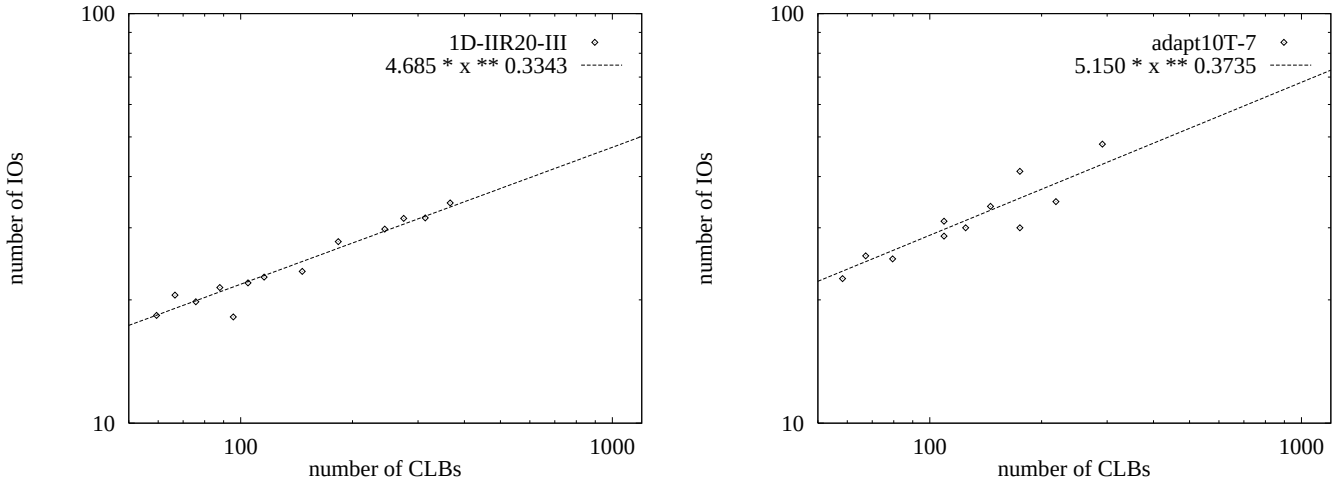



Figure 6.13: Rent's exponent approximation for 1D-IIR20-III and adapt10T-7. The two axes are log-scale.

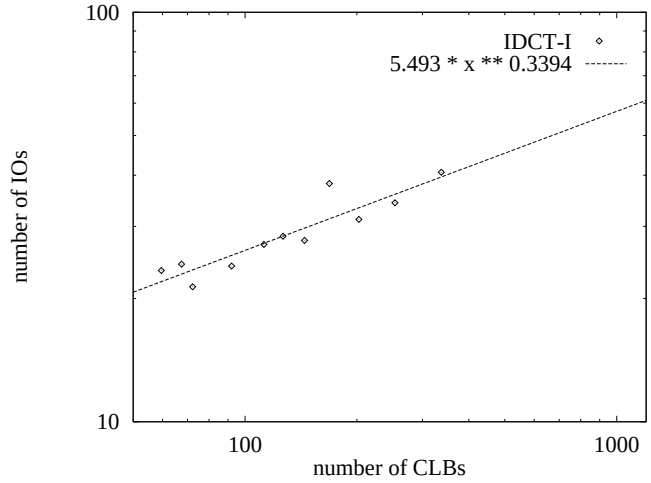



Figure 6.14: Rent's exponent approximation for IDCT-I. The two axes are log-scale.

tools are only able to derive a suboptimal placement solution because of their greedy algorithm which starts from some randomized intial placement. The layout solutions tend to vary quite significantly which severely affects the routing performance. When applying multiple place and route runs on circuits which are *marginally* routable using different intial placement for each run, the routing will succeed for some runs and will fail for others [81]. In our case, *all* circuits were routable at the first run. This indicates how easy it is to place and route our bit-serial circuits which is backed up by their low Rent's exponent values. Although, unfortunately, because of the variance in the routing performance, `ppr` is not capable of producing stable circuit performance even for our bit-serial circuits. We have set our performance goal to 40MHz using the XC3100A FPGA series which is currently the fastest device available from Xilinx. There are three types of signal propagation delay path within the FPGA:

- FF-FF: Flip-flop output to flip-flop input.
- Pad-FF: Input pad to flip-flop input.
- FF-Pad: Flip-flop output to output pad.

For multiple-FPGA systems, we also have to consider the chip-to-chip signal paths which is given by the FF-pad delay plus the pad-FF delay. Let us use the notation $D_{int}$ for the internal signal propagation delay (FF-FF) and $D_{ext}$ for the external signal propagation delay (FF-pad + pad-FF). We estimate the overall performance as the maximum of the two delay values (although to be exact, since FF-pad and pad-FF are both the maximum delay, the actual external critical path delay may be shorter than the sum of FF-pad and pad-FF. We simply take the safe side to avoid checking each external connection). However, in this analysis, parasitic capacitance and resistance on the MCM-level or board-level is not accounted for. FF-pad delay assumes a 50 pF external capacitive loads connected to the output pads, where the input capacitance of the input pads are 10 pF (large 175-pin PGA package has an input capacitance of 15 pF) [97]. If the total external capacitive load to each output pad including all the input capacitance of the other FPGA chips plus the parasitic capacitance on the inter-chip wires are less than 50 pF, the performance analysis done here would accurately reflect the actual performance. However, if the total capacitive load exceeds 50 pF, FF-pad delay will increase. Also, if there are interconnect chips along the inter-chip signal path, such as in FPMCM-I and Altera EPM8050M, this also adds more delays which will degrade the performance. Although this thesis does not address this issue, this inter-chip routing delay is a critical problem in large-scale configurable system, and therefore has to be handled on both the hardware design of the system as well as the layout methodology of minimizing the inter-chip delay. This issue should be addressed in the future work. Place and route runs are applied at most ten times to each partitioned subcircuits until $\max(D_{int}, D_{ext})$ becomes lower than 25ns. Tables 6.5.2, 6.9, 6.10, 6.11 shows the results for 12-14-4 neural network. Subsequent figures show the routing results of other bit-serial designs. Even if we ignore the MCM-level and board-level delays, the performance bottleneck comes from the inter-chip connections where signals has to reach the periphery of the chip at the one end and has to be routed to the interior of the chip from the periphery at the other end. There are a total of two subcircuits out of more than a hundred subcircuits which fails to meet the performance spec of 40MHz, both of them on the XC3195A which is the largest FPGA chip in this series. Xilinx 3000 FPGA family is known to have problem with the lack of routing resources for large chip size [89]. This is because the routing resource density is identical for all FPGA sizes ranging from 64 CLBs/chip to 484 CLBs/chip. In order to route non-planar "real-life" circuits, it is known that routing density has to increase as the chip size increase [34]. In the newer 4000 series FPGAs, larger chips are provided with higher routing resource density, which in return decreases the silicon efficiency. Another reason why the routing performance is lower for the larger FPGA chips is caused by the place and route tool. When the logic utilization is too high, it becomes harder to climb out of the local minima especially for the larger chips which decreases the placement search space. Also, FPGA placement algorithms are routability driven in general, or in other words, it tries to minimize the estimated total wire length. In the case of our bit-serial circuits where routability is high, it seems reasonable to assume that a *performance-driven* placement algorithm is possible to develop where it tries to minimize the *maximum* wire length instead of the total wire length.

| chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs | chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs |
|---|---|---|---|---|---|---|---|
| 0 | 19.3 | 20.8 | 4 | 8 | 16.2 | 22.7 | 1 |
| 1 | 14.4 | 21.1 | 1 | 9 | 15.6 | 25.8* | 10 |
| 2 | 15.8 | 24.9 | 5 | 10 | 13.6 | 19.4 | 1 |
| 3 | 14.1 | 18.6 | 1 | 11 | 13.2 | 22.6 | 4 |
| 4 | 14.1 | 23.1 | 1 | 12 | 13.6 | 24.2 | 3 |
| 5 | 13.8 | 22.3 | 1 | 13 | 16.3 | 23.7 | 1 |
| 6 | 12.9 | 24.3 | 1 | 14 | 11.7 | 20.1 | 1 |
| 7 | 12.9 | 23.2 | 1 | | | | |
| mean | 14.5 | 22.4 | 1.5 | worst | 19.3 | 25.8 | 10 |

Table 6.8: Layout result for 12-14-4 neural network (N-12-14-4) on XC3195A-3 FPGAs. $D_{int}$ is the internal signal propagation delay (FF-FF) and $D_{ext}$ is the external signal propagation delay (FF-pad + pad-FF). "# runs" is the number of place and runs applied to the subcircuit. MCM-level and board-level delays are not considered in this evaluation, although FF-pad delay assumes an external capacitive load of 50 pF where each input capacitance is 10 pF (15 pF for large 175-pin PGA). * are the designs which failed to meet the performance goal of 25ns.

| chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs | chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs |
|---|---|---|---|---|---|---|---|
| 0 | 14.0 | 21.1 | 2 | 11 | 12.7 | 18.0 | 1 |
| 1 | 15.1 | 24.0 | 4 | 12 | 12.9 | 21.9 | 1 |
| 2 | 12.2 | 17.8 | 1 | 13 | 11.7 | 22.1 | 2 |
| 3 | 18.8 | 24.9 | 2 | 14 | 17.4 | 23.3 | 2 |
| 4 | 12.9 | 19.0 | 1 | 15 | 12.4 | 22.3 | 1 |
| 5 | 13.9 | 22.4 | 1 | 16 | 13.1 | 19.1 | 1 |
| 6 | 19.0 | 17.5 | 1 | 17 | 12.5 | 19.7 | 1 |
| 7 | 13.7 | 20.3 | 1 | 18 | 12.5 | 21.4 | 1 |
| 8 | 12.6 | 21.8 | 1 | 19 | 11.6 | 18.2 | 1 |
| 9 | 12.4 | 17.0 | 1 | 20 | 13.9 | 23.1 | 1 |
| 10 | 13.7 | 18.9 | 1 | 21 | 12.1 | 18.5 | 1 |
| mean | 13.7 | 20.6 | 1.3 | worst | 19.0 | 24.9 | 4 |

Table 6.9: Layout result for 12-14-4 neural network (N-12-14-4) on XC3190A-3 FPGAs. * are the designs which failed to meet the performance goal of 25ns.

| chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs | chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs |
|---|---|---|---|---|---|---|---|
| 0 | 12.6 | 19.3 | 1 | 16 | 15.3 | 19.5 | 1 |
| 1 | 16.2 | 24.6 | 3 | 17 | 13.6 | 22.9 | 1 |
| 2 | 12.5 | 19.9 | 1 | 18 | 12.7 | 19.8 | 1 |
| 3 | 13.3 | 23.2 | 1 | 19 | 12.7 | 22.6 | 1 |
| 4 | 14.6 | 22.6 | 1 | 20 | 13.0 | 23.3 | 1 |
| 5 | 13.5 | 15.1 | 1 | 21 | 13.6 | 24.3 | 1 |
| 6 | 12.3 | 20.0 | 1 | 22 | 12.2 | 24.0 | 2 |
| 7 | 14.7 | 16.8 | 1 | 23 | 11.4 | 18.0 | 1 |
| 8 | 13.8 | 21.8 | 1 | 24 | 10.6 | 16.8 | 1 |
| 9 | 11.0 | 20.6 | 1 | 25 | 13.1 | 21.3 | 1 |
| 10 | 10.9 | 20.6 | 1 | 26 | 14.4 | 24.2 | 2 |
| 11 | 13.8 | 21.1 | 1 | 27 | 12.9 | 17.2 | 1 |
| 12 | 13.2 | 22.4 | 1 | 28 | 12.4 | 20.5 | 1 |
| 13 | 12.1 | 19.6 | 1 | 29 | 12.4 | 18.3 | 1 |
| 14 | 16.5 | 18.1 | 1 | 30 | 19.2 | 19.9 | 2 |
| 15 | 11.8 | 22.2 | 1 | 31 | 11.1 | 19.1 | 1 |
| mean | 13.2 | 20.6 | 1.1 | worst | 19.2 | 24.6 | 2 |

Table 6.10: Layout result for 12-14-4 neural network (N-12-14-4) on XC3164A-3 FPGAs. * are the designs which failed to meet the performance goal of 25ns.

Such algorithm is not available today. Also, since the inter-chip delays are likely to dominate the performance, the placement tool has to put special effort in minimizing the delays between the peripheral pads and the internal logic.

However, even though an absolute guarantee of a solid performance specification cannot be made because of the poor placement tool, we do see a fairly consistent performance, almost all circuits present a maximum clock speed exceeding 40 MHz. With the development of a true performance-driven placement tool, it can be expected that a solid performance guarantee can be quite possible.

| chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs | chip | $D_{int}$ (ns) | $D_{ext}$ (ns) | # runs |
|---|---|---|---|---|---|---|---|
| 0 | 11.9 | 16.6 | 1 | 25 | 12.8 | 21.2 | 1 |
| 1 | 12.0 | 22.4 | 1 | 26 | 12.7 | 20.3 | 1 |
| 2 | 13.4 | 23.9 | 1 | 27 | 11.8 | 14.7 | 1 |
| 3 | 13.7 | 15.7 | 1 | 28 | 13.6 | 19.1 | 1 |
| 4 | 13.4 | 16.4 | 1 | 29 | 11.5 | 18.6 | 1 |
| 5 | 14.4 | 19.1 | 1 | 30 | 15.2 | 21.5 | 1 |
| 6 | 12.2 | 17.3 | 1 | 31 | 13.2 | 22.9 | 1 |
| 7 | 13.5 | 24.3 | 1 | 32 | 13.2 | 19.3 | 1 |
| 8 | 15.8 | 21.0 | 1 | 33 | 13.6 | 22.8 | 1 |
| 9 | 11.6 | 16.8 | 1 | 34 | 11.8 | 19.6 | 1 |
| 10 | 11.5 | 16.8 | 1 | 35 | 14.2 | 24.9 | 1 |
| 11 | 13.0 | 21.4 | 1 | 36 | 12.7 | 20.7 | 1 |
| 12 | 11.4 | 17.3 | 1 | 37 | 12.2 | 22.3 | 1 |
| 13 | 12.8 | 20.2 | 1 | 38 | 11.9 | 16.8 | 1 |
| 14 | 13.5 | 20.7 | 1 | 39 | 12.1 | 21.3 | 1 |
| 15 | 12.3 | 19.5 | 1 | 40 | 11.8 | 17.1 | 1 |
| 16 | 12.4 | 21.6 | 1 | 41 | 13.7 | 19.1 | 1 |
| 17 | 11.0 | 21.2 | 1 | 42 | 13.6 | 17.0 | 1 |
| 18 | 16.3 | 20.2 | 1 | 43 | 12.1 | 17.5 | 1 |
| 19 | 11.8 | 15.8 | 1 | 44 | 10.4 | 22.7 | 1 |
| 20 | 13.1 | 20.4 | 1 | 45 | 12.4 | 24.2 | 3 |
| 21 | 11.2 | 22.4 | 1 | 46 | 14.0 | 22.1 | 1 |
| 22 | 13.2 | 19.7 | 1 | 47 | 16.2 | 22.3 | 1 |
| 23 | 11.4 | 24.0 | 1 | 48 | 12.6 | 22.0 | 1 |
| 24 | 11.5 | 17.5 | 1 | | | | |
| mean | 12.8 | 18.4 | 1.04 | worst | 16.3 | 24.9 | 3 |

Table 6.11: Layout result for 12-14-4 neural network (N-12-14-4) on XC3142A-3 FPGAs. * are the designs which failed to meet the performance goal of 25ns.

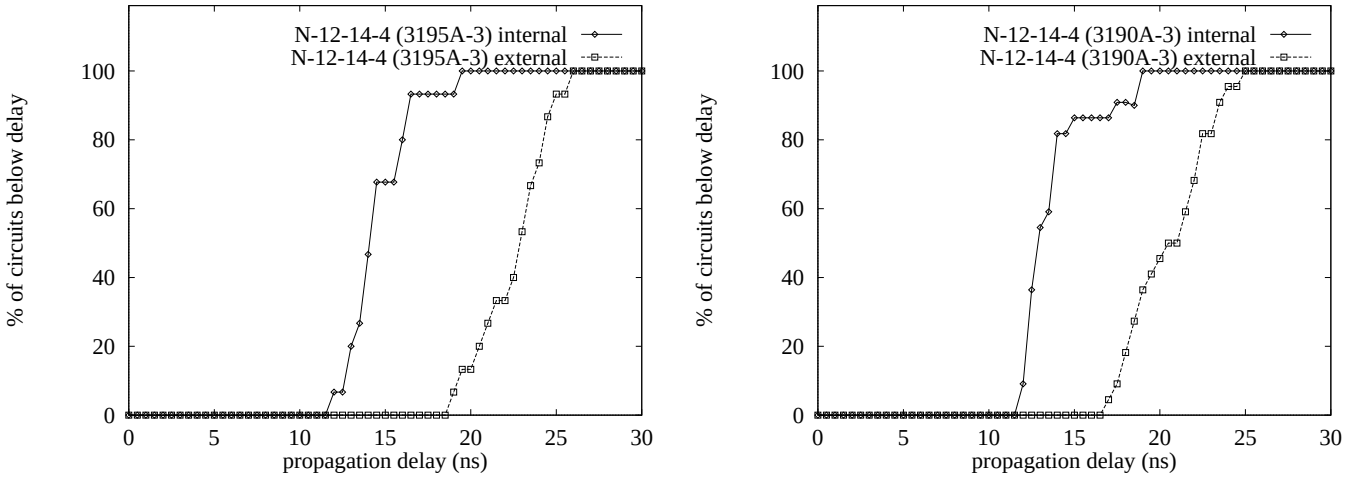



Figure 6.15: Estimated performance of N-12-14-4 on XC3195A-3 and XC3190A-3. "internal" refers to the internal signal propagation delay. "external" refers to the inter-chip signal propagation delay where the capacitive load of the output pads are assumed to be below 50 pF. The $x$-axis denotes propagation delay where the $y$-axis denotes the percentage of the total partitioned circuits whose maximum signal propagation delay is less than the corresponding value on the $x$-axis. For example, 67% of the circuits have the internal signal propagation delay of 15 ns or less, and all of the circuits have the internal signal propagation delay of 20 ns or less.

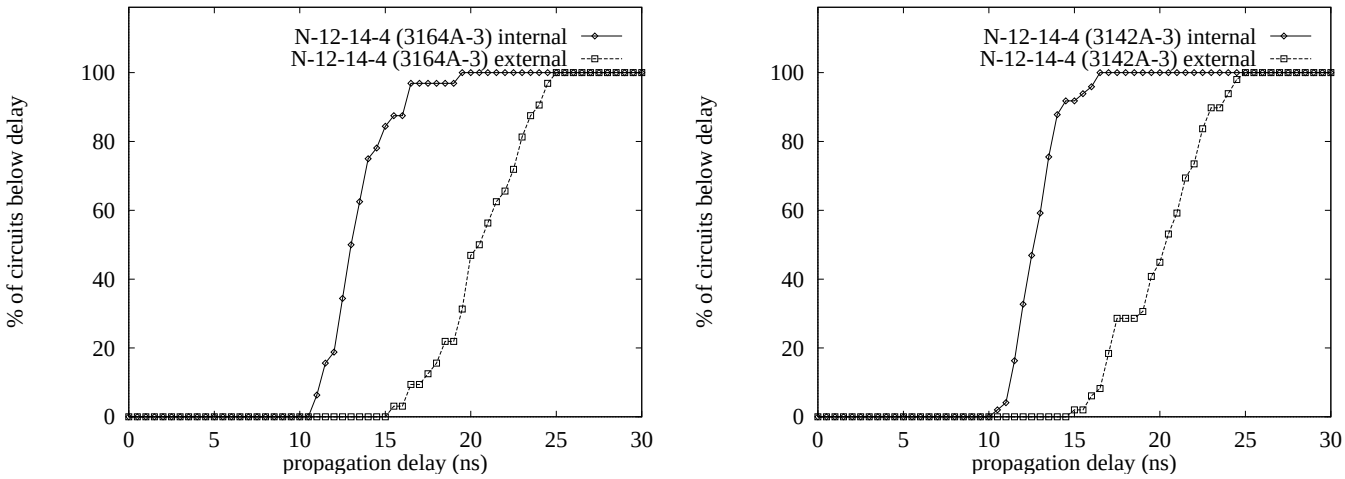



Figure 6.16: Estimated performance of N-12-14-4 for XC3164A-3 and XC3142A-3

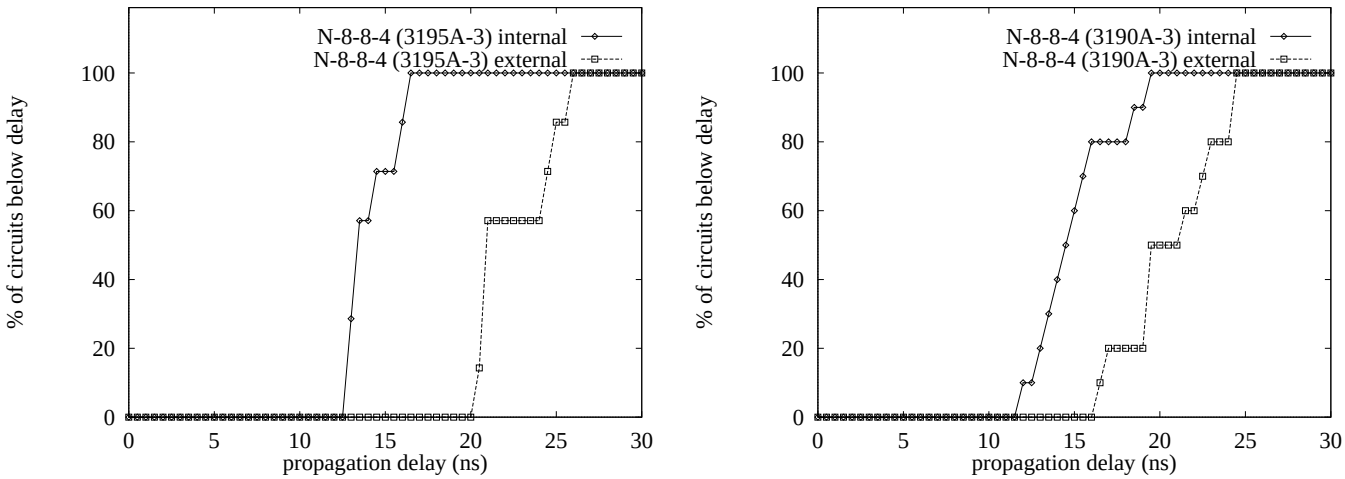



Figure 6.17: Estimated performance of N-8-8-4 for XC3195A-3 and XC3190A-3

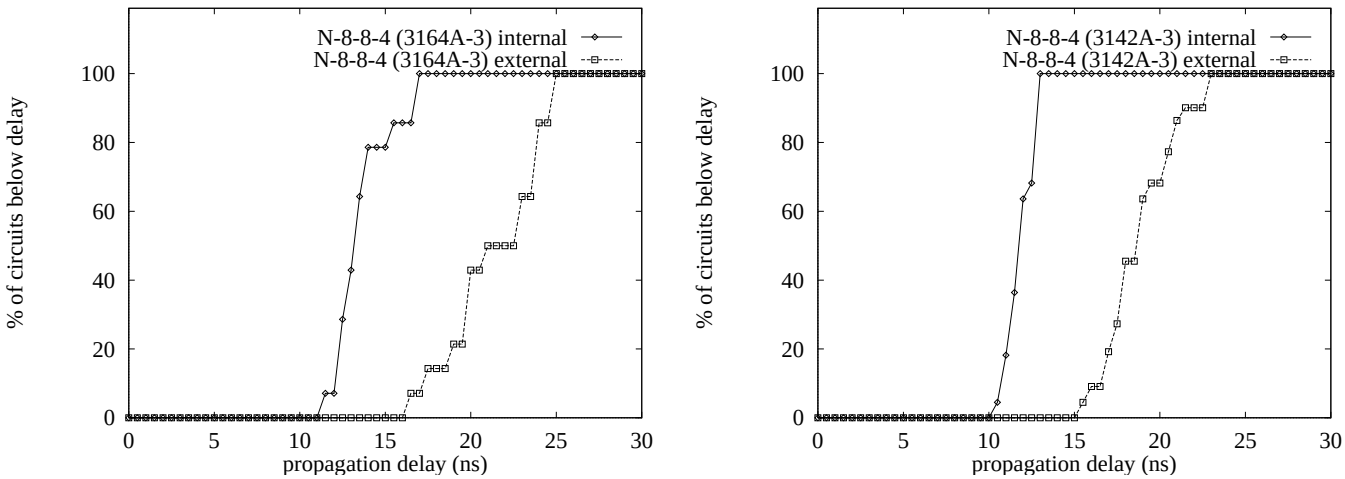



Figure 6.18: Estimated performance of N-8-8-4 for XC3164A-3 and XC3142A-3

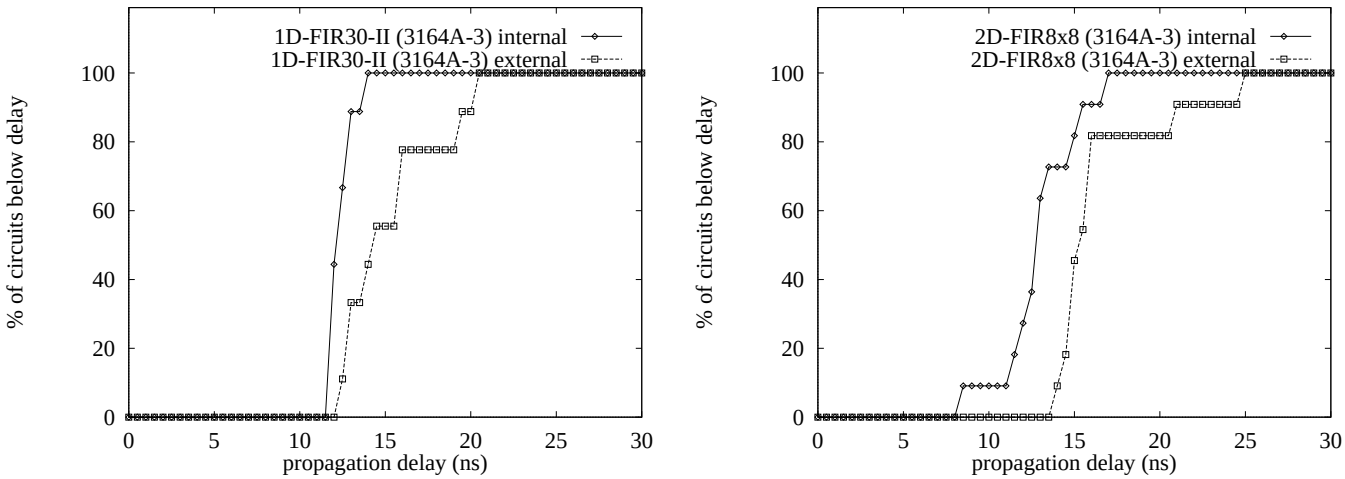



Figure 6.19: Estimated performance of 1D-FIR30-II and 2D-FIR8x8 for XC3164A-3

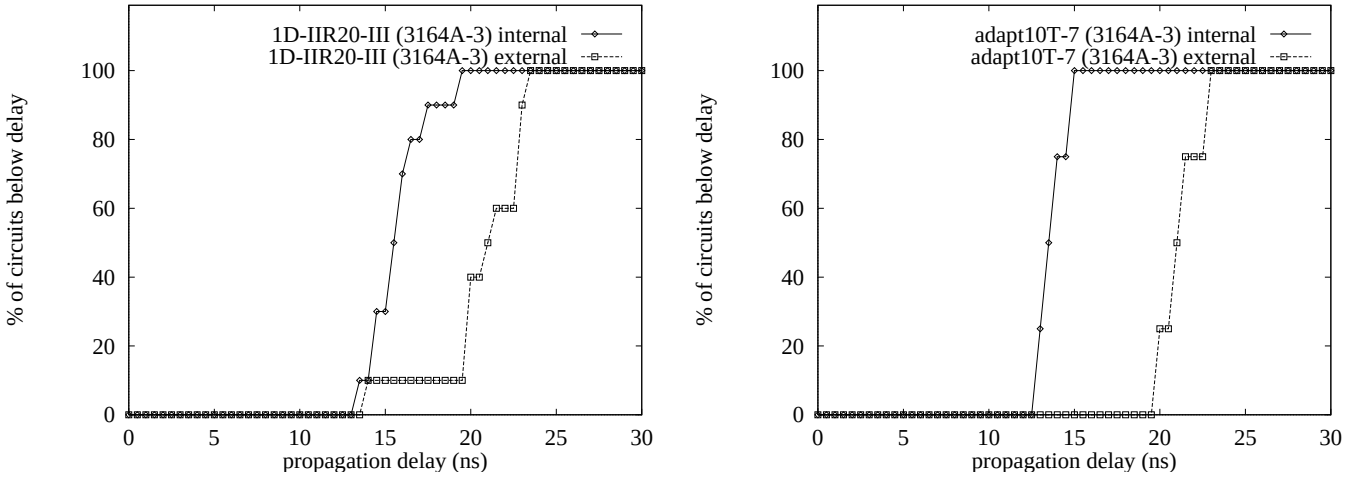



Figure 6.20: Estimated performance of 1D-IIR20-II and adapt10T-7 for XC3164A-3

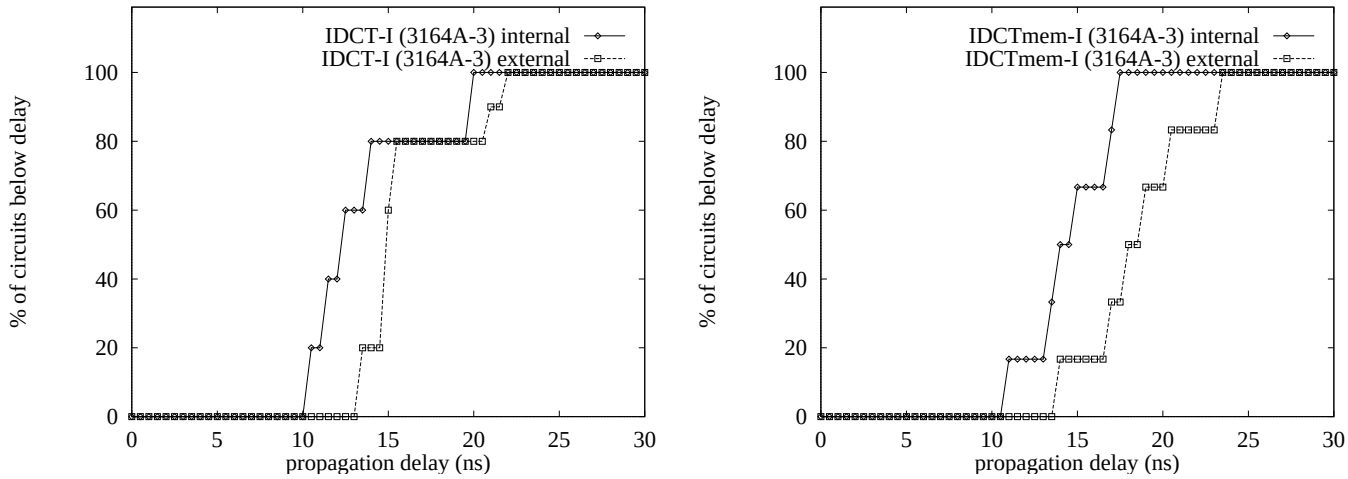



Figure 6.21: Estimated performance of IDCT-I and IDCTmem-I for XC3164A-3

# Chapter 7

# Conclusion

## 7.1 Summary

In order to deal with a large scale configurable system, the designer must be provided with a tool which enables him or her to explore a wide selection of algorithm implementation for a given problem and to efficiently implement a high performance hardware realization of the algorithm. This thesis focuses on the key issues which are essential in developing such a design environment for large-scale configurable systems. The author believes that this study has effectively paved the way for a large number of potential users of configurable systems to experiment new ideas with minimal knowledge of hardware designs and still produce high performance designs. Experienced engineers using FPGAs are still reluctant to use automated tools for their inability to produce high quality designs. We have demonstrated that the combination of C++ design capture system linked to the high performance circuit library and supported by the effective hardware synthesis system is capable of producing large-scale designs with extremely higher efficiency than other FPGA high-level synthesis systems. As for the low-level layout synthesis problem, we have demostrated that bit-serial circuits are efficiently mapped onto two-dimensional layout surface which takes away many of the problems such as routing resource limitation and IO pin limitation.

Here are the list of key contributions:

**Bit-serial datapath synthesis for FPGA design** 1. A set of bit-serial circuits have been designed for Xilinx 3000 FPGA architecture including adder, subtracter, multiplier, scaler (multiplier with one of the inputs being a constant), rounder, saturater. Their features are described below:

(a) All of these bit-serial circuits are composed of a unified logic structure with a single look-up table (LUT) followed by a flip-flop (FF). This unified logic structure enables the maximum clock frequency operation of the circuits.

(b) The bit-serial multiplier design uses a simplistic but a novel algorithm to calculate the sign-bit of the partial sum without requiring extra cycle [68], [73], [52] or large dedicated circuits [11] seen in other bit-serial multiplier designs. Our bit-serial multiplier has a throughput which is comparable to bit-parallel multipliers, where its circuit size is significantly smaller than bit-parallel multipliers.

(c) Two types of bit-serial scalers have been developed where one has a unified cell structure for providing reprogrammability of the scaling factor by changing the functionality of the look-up table (LUT), whereas the other has a customized structure for area efficiency.

(d) Four types of 2's complement bit-serial signals have been defined in order to support different signal precisions as well as different dynamic ranges.

2. A bit-serial pipeline synthesis system has been developed for the large-scale configurable system. The synthesis system developed in this thesis inherits most of the work done by the research group in IMEC on their CATHEDRAL-I bit-serial synthesis system for digital filter applications [58]. There were several new modifications needed on their work in order to apply this paradigm to the configurable system environment such as bit-serial control signal routing and IO scheduling of the system bus.
3. A simple partitioning method for the bit-serial pipeline datapaths is developed which is aimed to maximize the logic utilization of the FPGAs as well as minimizing the IO consumption.
4. Rent parameters [64] for a number of bit-serial designs have been observed including digital neural network which has a highly connective structure, 1D FIR filter, 1D IIR filter, 2D FIR filter, 1D adaptive FIR filter, and inverse discrete cosine transform (IDCT). The Rent exponents for these circuits range between 0.22 and 0.37. These results imply that the routing resources needed to perform the layout to these circuits on a two-dimensional surface is significantly less compared to that of the "real-life" circuits observed by Landman and Russo [64] and Hagen et al. [44] whose Rent exponents were observed to be between 0.47 and 0.75. These results also indicates that the partitioned circuits consume very little IO resource on the FPGAs.
5. The bit-serial circuits are placed and routed using `ppr` from Xilinx. All circuits were successfully routed on the first try even with the logic utilization as high as 100%. This confirms the high routability, and thus a low routing resource consumption, of these bit-serial circuits. Even though the placement algorithm used in `ppr` (as well as other commercially available tools) is routability-driven and does not address the performance issue effectively, performance measures of these bit-serial circuits after layout are fairly consistent among each other.

**C++ high-level design entry system**

1. This design capture method utilizes the function overloading feature of the object-oriented C++ language. By this approach, the time-consuming task of writing a dedicated compiler was eliminated. Also, this approach enables easy expansion and modification of the design capture system such as adding new feature, and therefore easy maintenance of the software.
2. The specific algorithm to be implemented on the configurable hardware are described as a difference equations on C++ using *signal variables* which are predefined data classes. This high-level design description enables the designer to explore the design space on the algorithm with minimal effort.
3. Each operation specified in the difference equations is mapped directly onto the configurable hardware using bit-serial circuits. Bit-serial circuit modules are coded in C++ to form a bit-serial circuit library.
4. A simple control-flow using for-loops and if-else statements can be described using a *control variable* which is another type of predefined data class.
5. A novel high-speed counter design has been developed. This counter design has a unique feature where the critical path length of the circuit remains constant (to be precise, one LUT and one FF to match the bit-serial circuit structure) for any counter size. This feature is not seen in any of the existing counter designs [97]. Nested for-loops are implemented as a cascade chain of this high-speed counter.
6. This design entry system also supports the use of RAMs external to the FPGAs.
   (a) Line delays utilize the external RAM. The address generator circuit for implementing line delays is automatically synthesized by combining the above high-speed counters.
   (b) Control variables can be used to specify the behavior of the address generator circuit for generalized RAM read and write access.
   (c) The external RAM can be used as a ROM where the address is generated by the signal variables.
7. A single-clock write cycle for the external SRAMs is implemented to provide high memory bandwidth. This makes use of the external controller which generates the event clock for the bit-serial datapath.

8. The C++ code can also include any regular C++ expressions to aid the design verification by means of mixed-mode behavioral simulation, such as reading the input data from a file, feeding the data into the configurable hardware, retrieving the processed data from the hardware, and displaying the results on the screen.
9. The C++ code is compiled by the GNU C++ compiler `g++` which produces the behavioral simulator as well as the circuit generator for the specific algorithm.
10. A number of signal processing applications have been coded in C++ such as 1D FIR filter, 1D IIR filter, 2D FIR filter, 1D adaptive FIR filter, IDCT, and digital neural network. In these design examples, a number of interesting trade-offs concerning the hardware size, performance and the quality of the behavior have been identified. All of these observed trade-offs are highly algorithm-dependent, and therefore the ability of this C++ design entry system which enables the designer to easily explore different algorithm implementation is well highlighted.

**Development of prototype configurable coprocessor system** A prototype configurable coprocessor system and the full supporting softwares were developed based on the first generation Field-Programmable Multi-Chip Module (FPMCM-I) which was developed by our research group [27]. This prototype system is built around the Sun workstation and SBus to prove the correctness of the bit-serial circuit designs and the bit-serial synthesis system. All of the discussed design examples except for the digital neural network was mapped onto the FPMCM-I to confirm their correct behavior.

## 7.2 Future Work

There are a number of issues which have not been addressed in this thesis which need to be handled in order to make this bit-serial approach more powerful and attractive for a wide variety of applications. Also during the course of the investigation of the bit-serial synthesis system, quite a few interesting problems have been identified which should be addressed in the future.

**Expanding the bit-serial circuit library** There are a number of other important operators which was not implemented in the current bit-serial circuit library. Those include constant adder for biasing, comparator, min-max selector and absolute operator. Other complex functions such as divider and square root operator may become necessary for some computer graphics applications.

**Expanding the bit-serial pipeline synthesis methodology** Currently, algorithms are mapped directly onto the hardware, therefore the algorithm implementation determined the outcome of the synthesized hardware. We can expand the synthesis methodology and provide several options for the synthesis strategy. Most promising approach seems to be the cooperating pipelined datapath architecture handled in CATHEDRAL-III [18] where a small degree of hardware sharing is allowed. If the available hardware is too small, the synthesis tool will effectively multiplex the operators to make the design area-efficient by sacrificing the performance. Or on the other hand, if the obtained throughput is not enough, the hardware can be duplicated to increase the throughput if such higher parallelism can further be exploited. This flexibility of the bit-serial synthesis strategy provides yet another dimension of the design space.

**Expanding the C++ design entry capability** There are a number of things which needs to be done on this C++ design entry system.

1. Support more versatile control-flow structures which is currently limited to static for-loops and if-else statements. Ideally, this C++ description should support a general FSM coding.

2. Support more versatile signal delay functions, in particular, allow a precise number of delays and line-delays in terms of sampling periods. Also, a mechanism to specify the IO interface protocal for multiple bus systems is needed which is not currently supported.
3. Expand more operators in parallel with the new bit-serial circuit design.

**Application development other than DSP applications** This thesis has focused mainly on DSP applications. By applying this system to other domains such as computer graphics and physics simulations, we should be able to observe the weakness of the current system much more clearly.

**FPGA architecture for bit-serial circuits** This thesis have successfully demonstrated the unique characteristic of bit-serial circuits in terms of Rent's rule. By optimizing the FPGA architecture for bit-serial circuit implementation, we believe that it is possible to dramatically increase the logic density which was primarily limited by the large, but still insufficient, routing area. Also, utilizing the MCM technology and introducing array-IOs adds new dimensions to the FPGA architecture optimization.

**Performance-driven placement** Since bit-serial circuits are highly routable, it seems possible to develop a performance-driven placement algorithm which sacrifices a relatively small decrease in routability for a performance increase. The bit-serial circuits designed in this thesis are implemented strictly with single-level LUT logic, and therefore *all* nets are critical. This should make the problem formulation easilier for the performance-driven placement.

**Multi-FPGA synthesis** Except for the simple yet effective partitioner, issues on multi-FPGA synthesis have not been focused deeply. Our FPMCM-I uses a centralized switch box (FPIC) which takes care of the inter-chip routing. If such device were absent in the configurable system, assigning the partitioned subcircuits to a specific FPGA location and providing a contact for each inter-chip connection is a hard problem. Together with the FPGA place and route problem, this becomes even more complicated since providing inter-chip connection means that the pin location has to be fixed after the inter-chip routing. Locking the IO pins severely hurts the place and route performance. Therefore, the IO pins must be locked with the consideration of routability.

# Appendix A

# FPMCM-I Architecture and Layout Synthesis Strategy

## A.1 FPMCM-I Architecture review

The basic architecture of the FPMCM-I has been described in the first chapter. Here, let us consider how this architecture affects the Multi-FPGA layout synthesis problem. The main feature of the FPMCM-I architecture is that there is a 1000-pin switch box (FPIC) which has direct connections to all FPGA chips. For each of the XC3042 FPGA chip on FPMCM-I which has 96 user-IO pins, 12 pins are connected to the right adjacent FPGA, 12 pins to the left adjacent FPGA, 8 pins to global nets including 2 clock nets, and 22 pins are connected to the external pads, all of which have a direct connection with the FPIC. The remaining 48 pins are connected only to FPIC (Table A.1). Note that all nets have contacts with FPIC. This means that any FPGA pins have direct access to all the nets on FPMCM through the FPIC. This property is especially important for multi-FPGA system layout problems which enables us to drastically simplify the layout synthesis tools as discussed earlier.

| type | # pins | external | FPIC | left | right | global |
|---|---|---|---|---|---|---|
| LEFT | 12 | - | ○ | ○ | - | - |
| RIGHT | 12 | - | ○ | - | ○ | - |
| EXTERNAL | 20 | ○ | ○ | - | - | - |
| INTERNAL | 48 | - | ○ | - | - | - |
| GLOBAL | 4 | ○ | ○ | - | - | ○ |

Table A.1: IO pin connections on each XC3042 FPGA chip. The left column shows the type of nets connected to the FPGA pins. For each type of net, the number of pins is shown on its right, and also connectivity to the following components are indicated : FPMCM external IO pads, FPIC, left adjacent FPGA, right adjacent FPGA, and global net.

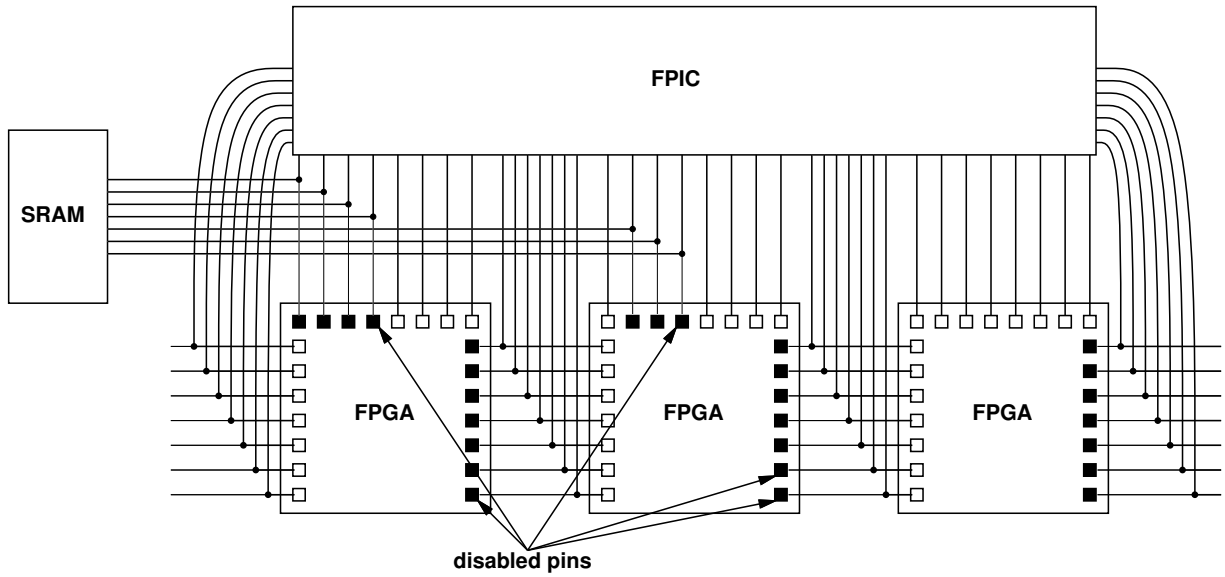



Figure A.1: FPMCM pin usages. First, the direct connections to the right adjacent FPGA are disabled. These wires are reserved exclusively for the right adjacent FPGA. Second, connections which has a direct path to any external devices is disabled. These wires are reserved exclusively for establishing connections between the external device and the FPIC.

## A.2 FPMCM Layout Synthesis Strategy

Since there are few direct connections between FPGAs, we cannot use the single-switch box model where all the inter-chip connections are done through the switch box. The synthesis strategy developed here is to sacrifice half of this direct connection pin resources and use the remaining 80 pins as user-IO pins connected directly to the switch box.

1. Disable the IO pins which are directly connected to the right adjacent FPGA. Use the IO pins which are directly connected to the left as general IO and ignore the direct connection (Fig.A.1).

2. Disable the IO pins which have direct path to any external devices through the FPMCM IO pad (Fig.A.1). In this way, the need for IO pin assignment can be totally eliminated since all types of connections (FPGA $\leftrightarrow$ FPGA, FPGA $\leftrightarrow$ system bus, FPGA $\leftrightarrow$ SRAM) can now be implemented exclusively through the FPIC (Fig.A.2), and the FPIC is flexible enough to implement connections for different pin permutations.

3. *Circuit partitioning* : Partition the bit-serial pipeline network into groups whose logic sizes are small enough to fit into one XC3042 FPGA chip.

4. *FPMCM placement* : Arbitrary assign an FPGA location to each circuit partition. Physical allocation of each circuit partition does not have a large impact in the FPMCM routability because of the FPIC.

5. *FPGA placement and routing* : With the constraint file indicating the disabled IO pins, run placement and routing for each circuit partition independently.

6. *FPIC routing* : From the IO placements given by the FPGA placement and routing, do the FPIC routing.

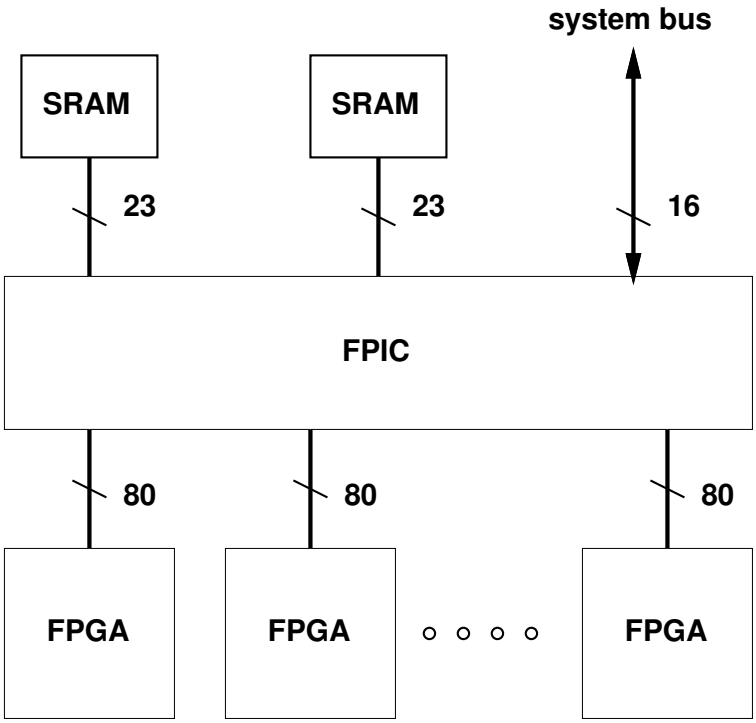



Figure A.2: FPMCM routing model. Routing between any devices are implemented exclusively through the FPIC. In this way, IO pin locations do not have to be predetermined, and therefore, the individual FPGA placement and routing can occur independently.

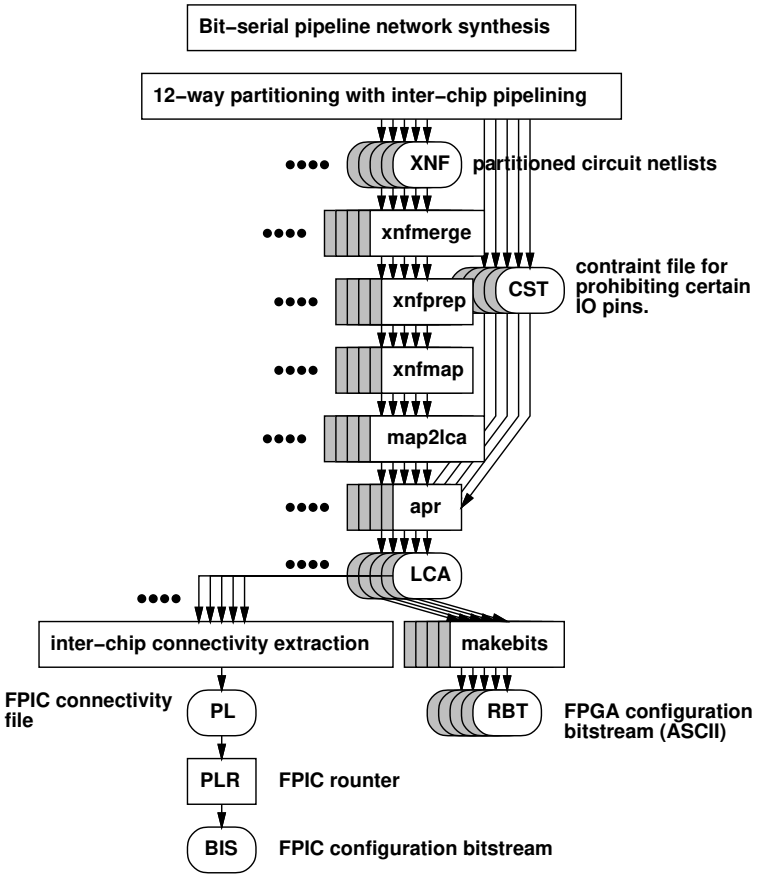



Figure A.3: FPMCM layout compilation flow.

## A.3 FPMCM Layout Flow

The FPMCM layout compilation flow is illustrated in Fig.A.3.

1. After the 12-way partitioning of the bit-serial pipeline network, up to 12 circuit netlists are generated in Xilinx Netlist Format (XNF).
2. Each circuit partition is assigned to a certain FPGA inside the FPMCM.
3. For each circuit partition, a contraint file (`.cst`) is created to indicate which IO pins are prohibited for the place & route tool.
4. Design rule checker (`xnfprep`), file format converters (`xnfmerge`, `xnfmap`, `map2lca`) and automatic placement and routing program (`apr`) are called from the synthesis program automatically.
5. From the routed circuit netlist file (`.lca`), the inter-chip connectivity is extracted and the connectivity file for the FPIC (`.pl`) is created.
6. FPIC router (`plr`) is called which generates the FPIC configuration bitstream.
7. `makebits` programs are called for each .lca file which generates the FPGA configuration bitstreams. Here, `.rbt` files are used which is in ASCII format instead of `.bit` which is in binary format. `.bit` files are only readable by the Xilinx software.

# Appendix B

# Prototype Configurable Coprocessor System

The author has built a prototype configurable coprocessor system using first generation FPMCM developed at our project team. The main purpose of the system was to first test the FPMCM itself, and then to test my bit-serial datapath synthesis system as a proof-of-concept vehicle. It should be noted that this prototype system was not intended for practical uses to compete with other existing configurable systems. There are many issues concerning the board architecture, communications between the host and the configurable coprocessor, memory configuration, device driver on the host, application software development, etc. These issues are not the main focus of this thesis and therefore are not investigated thoroughly.

## B.1 System Architecture

Fig.B.1(a) illistrates the architecture of the overall system. The host computer is the Sun workstation (SPARC-classic), and an SBus interface card called SLIC EB-1 from DAWN VME PRODUCTS is attached onto the SBus slot inside the chassis of the workstation. The prototype board which contains the FPMCM-I is connected to the SLIC EB-1 through a 60-pin ribbon cable. Please refer to [67] for information about SBus specifications.

**Sun SPARC-classic workstation** This is a standard Sun workstation which is normally operates on the network configuration. It has a small internal hard disk (200MB) and can be configured to operate on a standalone mode as well. The speed of the SBus clock on the SBus slot of this machine is 25 MHz.

**SLIC EB-1 SBus interface card** SLIC EB-1 is a kit for prototyping hardware and software which provides an easy solution for developing new SBus applications or porting existing applications from a different bus architecture. The kit centers around the Motorola MC92005 which is a complete SBus slave interface in a single chip (Fig.B.1).

- MC92005 SBus Slave Interface Controller Chip : This chip is a complete 32-bit slave interface which meets all timing, loading, and drive requirements of the SBus Specification B.0. It works properly over the entire SBus speed range, from 16.67 to 25 MHz. Burst data transfer rates over 80 Mbytes/sec are supported, with sustained data transfer rates of 33.3 Mbytes/sec. It also provides a versatile *private bus interface* called PBus that can be directly connected to peripheral or memory chips, and to the *SBus configuration PROM* which is required for all SBus slaves. The timing and protocol of PBus cycles is programmable, and can be tailored to each of the peripheral/memory chips on the PBus.

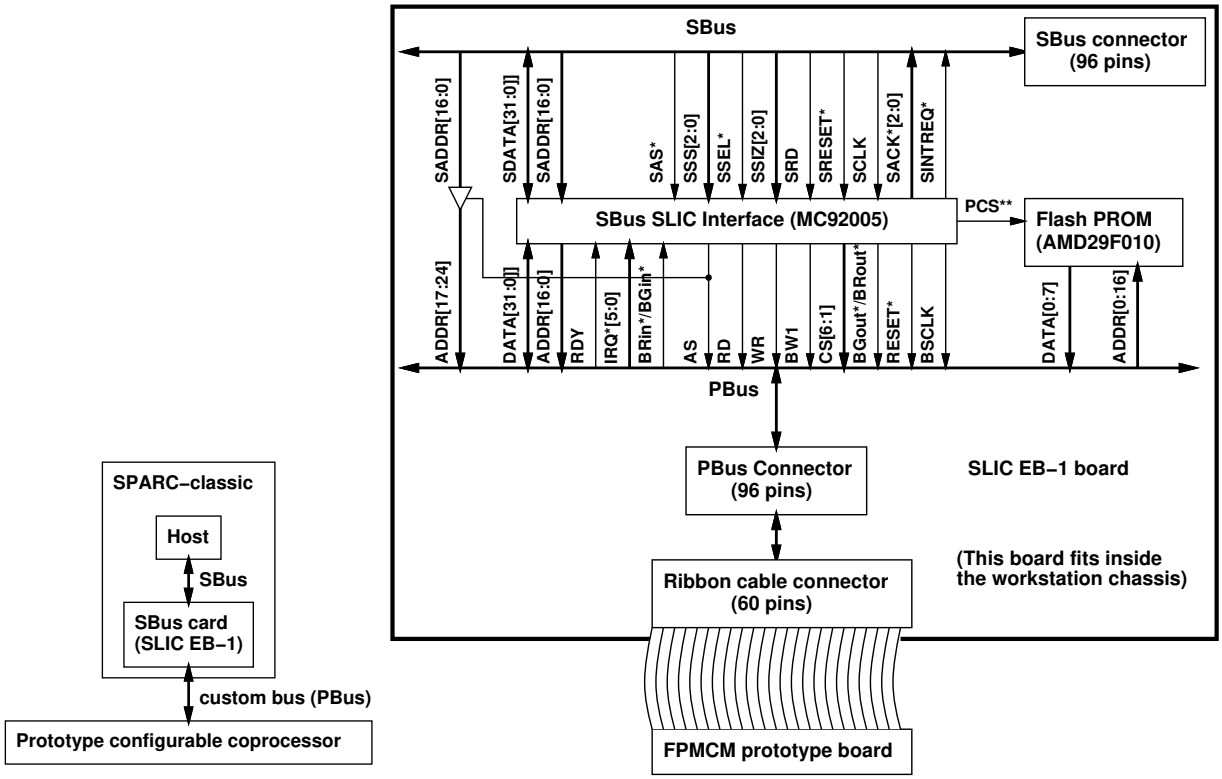



Figure B.1: Prototype configurable system architecture and the SLIC EB-1 SBus interface card.

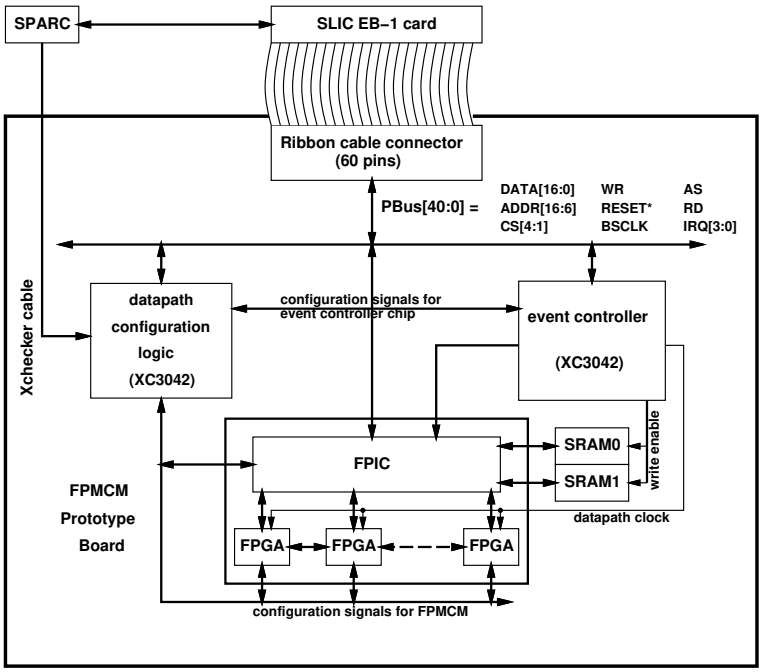



Figure B.2: Prototype configurable coprocessor board architecture

- 29F010 AMD FLASH PROM : This PROM includes the necessary information for the initialization of the SBus after powerup of the workstation. During this initialization phase, the host computer scans through all the available SBus slots on the machine to check if there are active devices attached to them. The PROM provides the host about the ID and the operating parameters.
- PBus : This is an asynchronous programmable private bus, whose signals and timing can be dynamically redefined. This programmability greatly helped us in the development of our prototype board. The PBus is connected to the prototype board through a 60-pin ribbon cable. The total number of signals in the PBus is 71, which includes 32 data signals, 17 address signals, 6 chip select signals and 6 interrupt request signals. Therefore, we were forced to eliminate many of the signals in the PBus to be connected through the ribbon cable.

**Prototype configurable coprocessor board** The basic architecture of the board is illustrated in Fig.B.2. The prototype board was developed exclusive with wirewrapping in order to save the cost and development time. We also used FPGAs for glue logic which handles FPMCM configuration and runtime support.

## B.2 Board Configuration and Operation

The configurable coprecessor board is composed of four components: datapath configuration logic, runtime datapath event controller, coprocessor datapath and SRAMs.

**Datapath configuration logic** This component is responsible for downloading the configaration bitstreams to the programmable devices which compose the datapath and the datapath event controller. This is implemented by an FPGA (Xilinx XC3042) and is programmed through the Xchecker serial cable which connects to

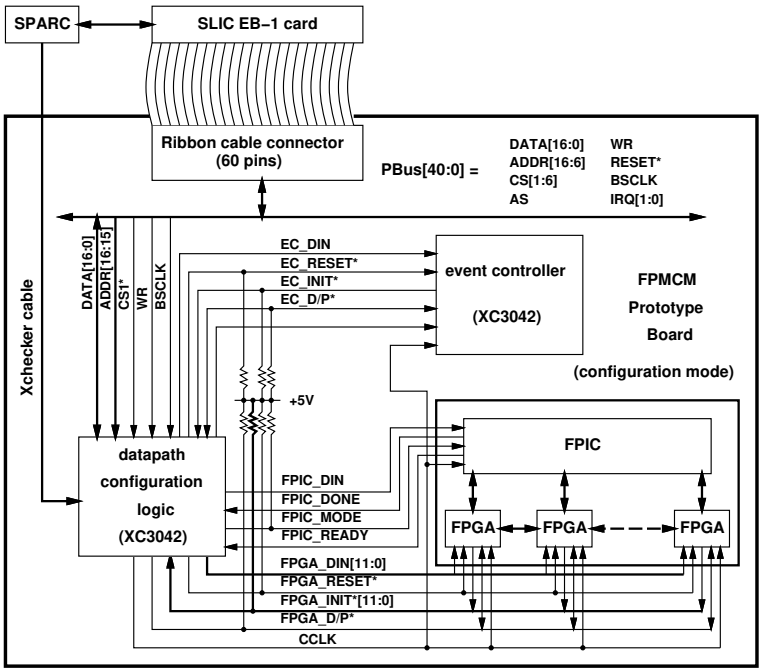



Figure B.3: Configurable coprocessor board in the configuration mode.

the workstation. This configuration logic needs to be programmed through the Xchecker cable only once after the system powerup. In the configuration mode as illustrated in Fig.B.3, the configuration bitstreams are transmitted from the host computer through the SBus and PBus and eventually captured by the configuration logic. Configuration of the programmable devices on the board is done in the following way:

1. The twelve XC3042 FPGAs and the FPIC inside the FPMCM-I is configured in parallel. The Event Controller Chip (also an XC3042 FPGA) is configured seperately from the FPMCM, since the same event controller circuit would service a number of different applications.
2. There are a number of configuration signals on each FPGA chip and the FPIC which are driven by the configuration logic ($config\_fpmcm[15:0]$=$\{FPGA\_DIN[11;0]$, $FPGA\_D/P*$, $FPGA\_RESET*$, $FPIC\_DIN$, $FPIC\_MODE\}$). These configuration signals are driven by a register inside the configuration logic. We configure the twelve FPGA chips and the FPIC in parallel by constantly rewriting the values on this register by the configuration program at the host computer. And also there are a number of status signals on each chip as well ($status\_fpmcm[14:0] = \{FPGA\_INIT*[11:0]$, $FPGA\_D/P*$, $FPIC\_DONE$, $FPIC\_READY\}$) (please refer to the Xilinx databook and Aptix databook for details). These signals are captured in parallel at the registers inside the configuration logic (Fig.B.4). The value of this register is monitored by the configuration program at the host computer. Configuration of the event controller FPGA chip is done in the same manner using seperate set of registers. A total of four registers are accessed by the two address bits $ADDR[7:6]$.
3. The average bandwidth of the PBus during configuration is observed to be around 3 million words per second. There are 300K bits of configuration data for the FPIC and 30K bits for each XC3042. Since all the devices are programmed in parallel, the configuration time of the whole FPMCM is dominated by the configuration time of the FPIC, which is about 0.1 second.

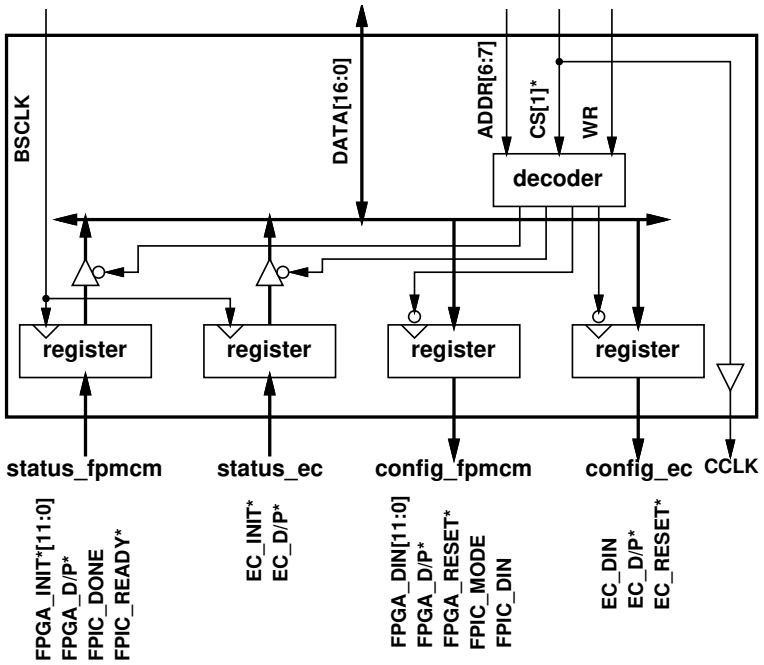



Figure B.4: Configuration logic architecture.

**Datapath event controller** During the application execution at the configurable coprocessor, the event controller handles the communication between the SBus card. During the operation mode as illustrated in Fig.B.5, the event controller generates several clock signals for the FPMCM datapath circuit:

1. All PBus transactions are initiated by the application software running on the host computer. In the current setup, SBus burst transfer is not supported. Therefore, each data has to be sent to the configurable coprocessor from the host through SBus and PBus one at a time. The effective bandwidth is thus reduced to 1 million to 3 million words per second. The difference in the effective bandwidth comes from the complexity of the SBus driver routine. The driver issue is discussed later.
2. The datapath event controller operates on the 25 MHz buffered SBus clock ($BSCLK$).
3. For PBus write transaction, it generates an input clock ($in\_clk$) for the coprocessor datapath to capture the incoming data on the PBus.
4. For PBus read transaction, it generates an output clock ($out\_clk$) for the coprocessor datapath to register the outgoing data to be written on the PBus.
5. For each PBus transaction after the input or output clock is activated, it generates event clocks for the datapath. The bit-serial circuitry operates strictly on this single clock signal except for the input and output registers connecting to the PBus. The timing of the PBus read/write operations is illustrated in Fig.B.6. The number of event clock cycles to be generated is indicated in the PBus address bits ($ADDR\_[9:6]$). In the current hardware setup, the programmer can set the number of clock cycles between 0 and 15. Note that the number of cycles can be different for every PBus transaction. Due to this event clocking scheme, the datapath becomes active only when there are data to process on the PBus.

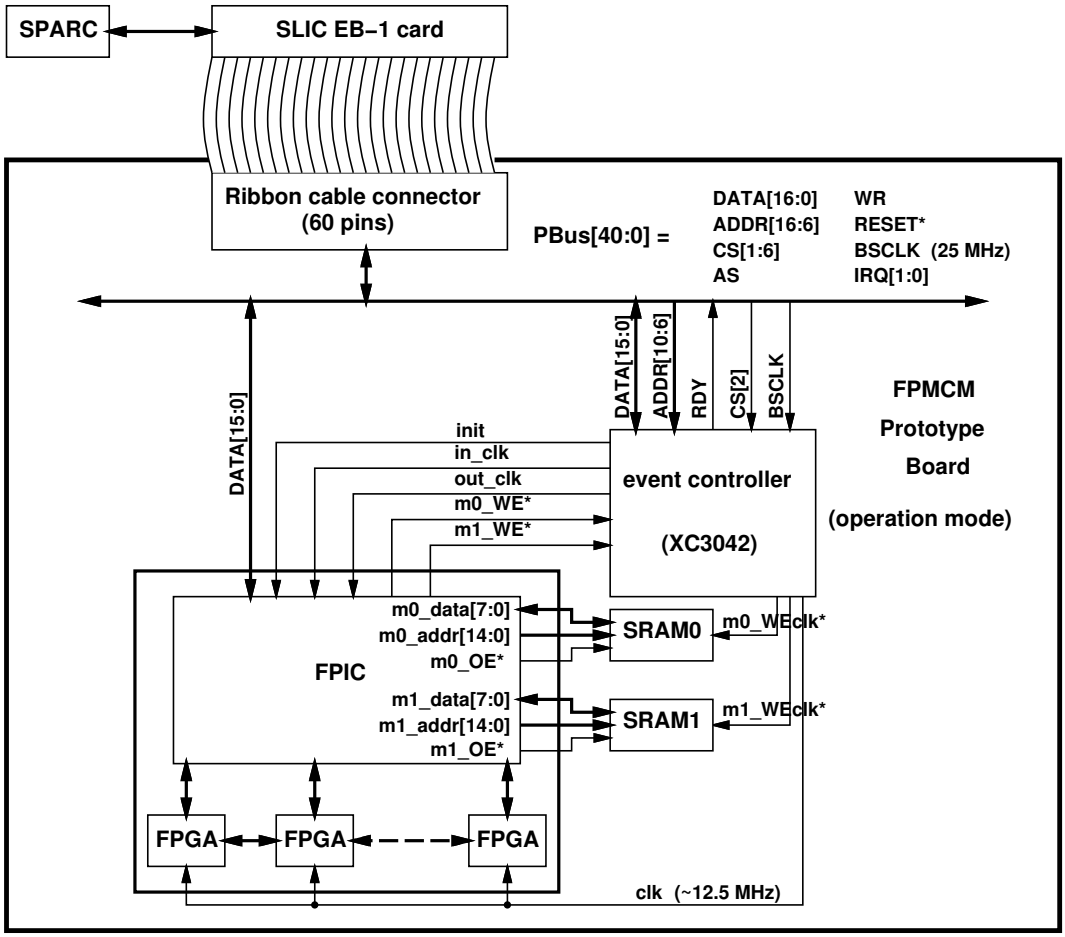
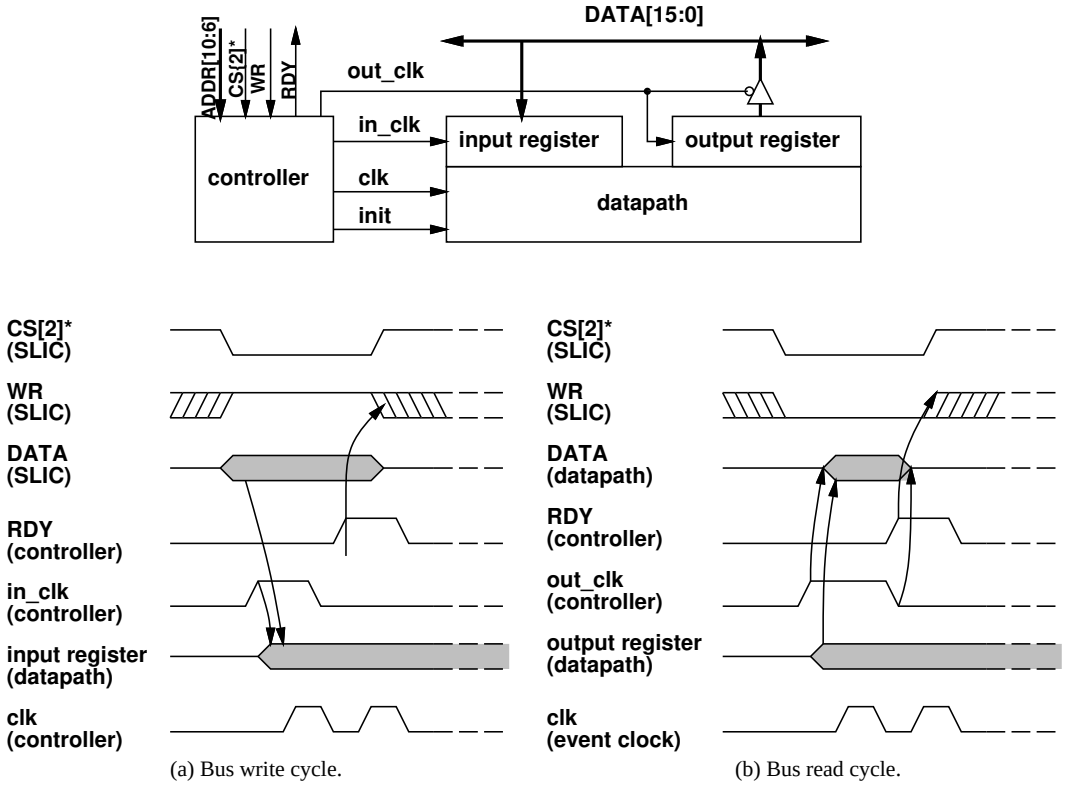


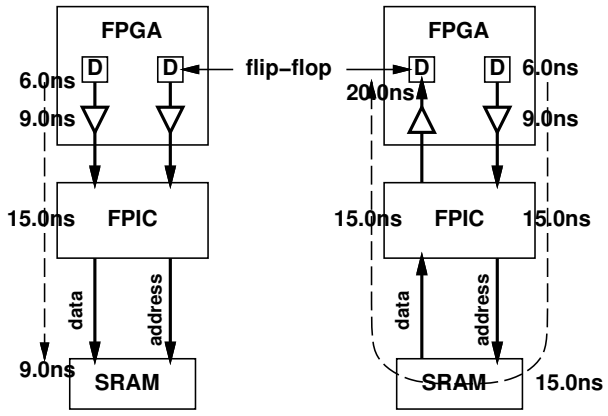

Figure B.5: Configurable coprocessor board in the operation mode.




(a) Bus write cycle.

(b) Bus read cycle.

Figure B.6: PBus timing.




(a) Memory write cycle (worst case ~40ns)

(b) Memory read cycle (worst case ~80ns)

Figure B.7: SRAM write and read operations through the FPIC. Read operations require roughly twice the time compared to write operation since the signal path is longer.

6. At the beginning of the operation after the configuration of the datapath, an initialization pulse ($init$) is generated for the datapath to create the bit-serial control signals. This information is included in $ADDR$[10].

**FPMCM datapath** All the interconnections between FPGAs and also between the external devices are implemented through the FPIC. By doing so, the partitioning problem of the datapath circuit can be completely isolated from the physical configuration of the FPMCM.

**SRAMs** Currently, two 32Kbyte SRAM chips are connected to the datapath FPMCM. The write enable signals are generated by the event controller according to the scheme we have described in Appendix A.

At the datapath event controller, event clock cycles are generated by dividing the buffered SBus clock ($BSCLK$) by two (12.5 MHz), three (8.33 MHz) or four (6.25 MHz). Most of our bit-serial datapath examples run easily on divide-by-two event clock. However, due to the long path between the FPGAs and the SRAM, circuits with memory access sometimes does not function properly on this divide-by-two clock and in this case, event controller is reconfigured to produce a slower clock. Fig.B.7 illustrates the worst case scenario. For a read operation, the signal path includes the address lines which are driven by the FPGA chip, through the FPIC, to the SRAM, and the data lines driven by the SRAM, again through the FPIC, back to the FPGA. Assuming that the maximum FPIC delay for the address lines and data lines is 15 ns for the worst case, and assuming that the FPGA chip is the slowest speed device (-70), the total signal delay is 80 ns which is too close to the divide-by-two clock period. Here, we did not consider the routing delay inside the FPGA for the address lines. Adding to the fact that the FPGA may even be slower than the slowest device available, since the FPGA bare-die was not speed tested before MCM assembly, there are enough possibility that this memory read cycle may not be fast enough. Write cycles, on the other hand, can be as twice as fast as the read cycle because of its short signal path. The slower clock speed inevitably decreases the performance, however the performance is limited by the slow bus transactions which ranges from $1 \sim 3$ MHz, this performance decrease is usually small especially when the IO usage is high. For instance, a data sampling period of 8 clock cycles with 2 IO transactions per sampling period (one input, one output), assume one bus transaction takes 1000 ns. If it is running on a 12.5 MHz event clock, it takes 640 ns to cycle one sampling period on the event clock, and a total of 2640 ns is needed to complete one sampling period. If we now change the event clock speed to 83.3 MHz, it takes 960 ns to cycle one sampling period on the event clock, and a total of 2960 ns is needed to complete one sampling period, which is only a 12 % decrease in performance.

# Appendix C

# SBus Driver and Software Issues

## C.1 SBus Addressing

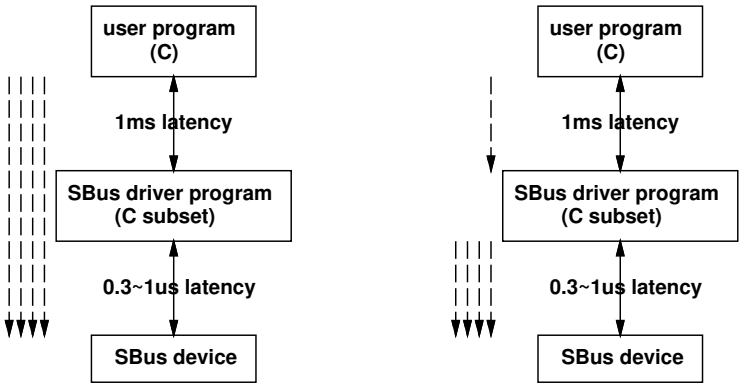
The SBus interface chip MC92005 on the SBus card handles 1-Mbyte address space on the SBus, and this address space is subdivided into eight 128-Kbyte address spaces:

1. First 128-Kbyte address space is assigned to the PROM for device ID and parameters for the SBus configuration. When this address space is accessed, chip select signal $PCS*$ becomes active.
2. Next six 128-Kbyte address spaces are assigned to the six peripheral devices. When one of these address spaces is accessed, chip select signal $CS[i]$ $(i = 1, 2, 3, 4, 5, 6)$ becomes active. In our configuration, $CS[1]$ is assigned to the FPMCM configuration logic chip, and $CS[2]$ is assigned to the FPMCM datapath and the event controller chip.
3. The last 128-Kbyte address space is assigned to the internal registers of the SBus interface chip. These internal registers are used to configure the PBus interface timing and signal polarity of each peripheral device.

## C.2 SBus Device Driver

The majority of the SBus device driver code written in a subset of C is inherited from the sample code which was provided by the SLIC EB-1 kit. In the sample driver, there were only two types of command in which the user program (written in C) can access the SBus card: `read_hw()` and `write_hw()`.

1. `u_long read_hw(u_long chip, u_long offset)` : This reads in a data on the SBus whose address is calculated by the two variables `chip` and `offest`.
   - `u_long chip` : This variable takes one of the three predefined values (`EPROM`, `PORT` and `REG`) which acts as an address index. `u_long` stands for `unsigned long`.
     - `EPROM` points to the beginning of the 128-Kbyte PROM address space.
     - `PORT` points to the beginning of the first 128-Kbyte peripheral device address space.
     - tt REG points to the beginning of the 128-Kbyte internal register address space.
   - `u_long offset` : This value is added to the `chip` variable for the final address calculation.




(a) Single data transfer from the user program.

(b) Data array transfer from the user program.

Figure C.1: Accessing from the user program. Overhead in accessing the SBus driver from the user program is about 1ms, whereas the SBus driver accessing the SBus device is around $0.3 \sim 1\mu$s. Instead of calling the functions in the SBus driver a number of times which takes an enormous amount of time, call the function once and let the SBus driver handle complex data transaction between the SBus devices.

In order to access different peripheral devices on the PBus, `chip` is set to `PORT`, and `offset` is set as `offset = i * 0x20000 + j` where `i` is the number of the peripheral device counting from zero, and `j` is the true offset of the address. In our case, there are two types of operations where providing the true offset is necessary:

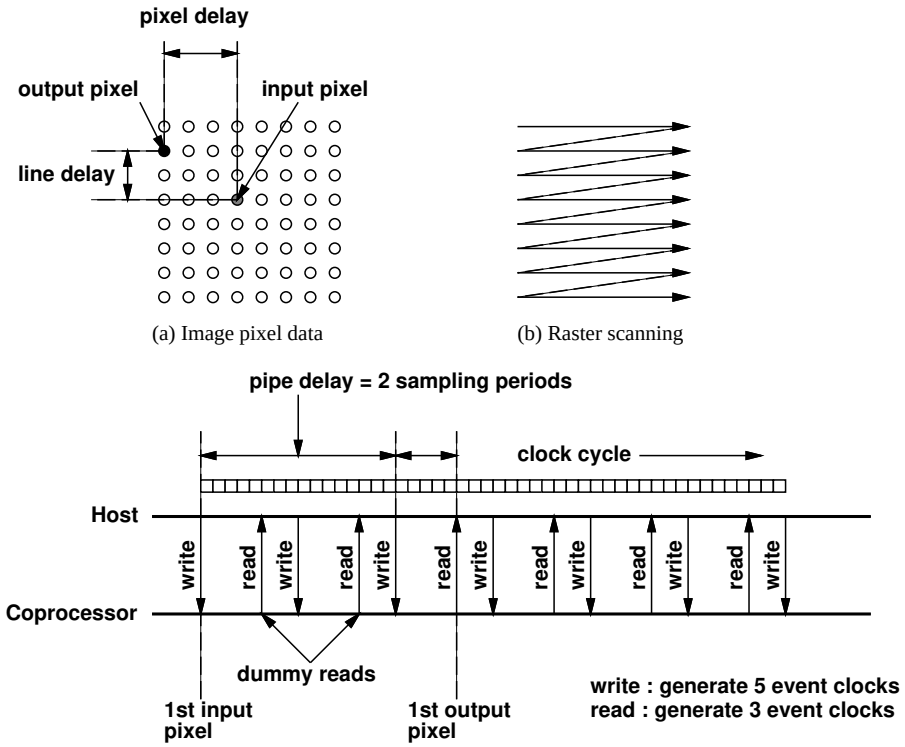
(a) During configuration, the user program has to specify which register inside the configuration logic to access by assigning the true offset as either 0, 64, 128 or 192 ($ADDR[7:6]$).

(b) At the beginning of the application execution, $ADDR[10]$ has to be high in order to generate the initialization pulse at the event controller. Therefore add 1024 to `offset`.

(c) At each SBus transaction, the user program has to provide the number of event clock cycles to be generated at the event controller. If $k$ cycles were to be generated ($k < 16$), add $k \cdot 64$ to `offset`.

2. `u_long write_hw(u_long chip, u_long offset, u_long data)` : This the value `data` to the address indicated by `chip` and `offset`.

These two functions are the only essential elements which is needed to communicate with our configurable coprocessor. Configuring the programmable devices and running applications only requires these two functions. However, there is a huge drawback with these elementary functions, and that is, *they are extremely slow*. The reason for this is that the function is called from the user program where the SBus driver is running on the kernel level. Accessing the SBus device driver from the user program introduces a large overhead, and as a result, each read or write operation takes about 1 ms which we observed on our SPARC-classic. This speed may be enough for diagnosting the hardware status. But this speed is not acceptable for any practical applications requiring large number of data transactions. With this data rate, even configuring the FPMCM would take more than 6 minutes. The solution is to this is to write customized IO functions and add them to the SBus driver code. For downloading a large amount of data to the board, instead of using `read_hw()` function for each data, the user program would send the pointer of the data array to the SBus driver and the driver will automatically download the data array to the SBus card and therefore eventually to the configurable board (Fig.C.1). In this way, the overhead of calling the SBus driver is minimized, and a fast data transaction is now possible. However, the need for adding customized IO functions




(a) Image pixel data

(b) Raster scanning

(c) Data communication of the image pixels.

Figure C.2: Data transaction of the image pixel data. The host computer writes pixel data to the SBus device while reading back the processed data.

to the SBus driver introduces inefficiencies in the application development itself. First, SBus drivers are hard to write because of many restrictions compared to the normal C-code. Array of data has to be transmitted from the user program level to the SBus driver kernel level through a specialized buffer structure. Second, errors caused by a device driver code in general often lead to a system crash which would take minutes to reboot. There is no sufficient debugging environment as in C programming. Troubleshooting the device driver problems is a very painstaking process. Third, installing a new device driver requires the workstation to reboot. A loadable device driver is possible if the user have the superuser privilege which may not be possible as in our case. However, a loadable device driver is not much of an advantage considering that while designing or modifying the driver, you will mostly likely crash the system at least a couple of times anyway.

We have written a few complex IO functions which can be used widely which is described below. The speed of the data transaction varies between 1 MHz and 3 MHz, mainly due to the complexity of the SBus device driver routines.

- `read_array_hw(u_long chip, u_long offset, u_long data[], u_long size)` : This function reads in an array of data from the SBus device whose address is calculated by `chip` and `offset` and stores the data in array `data[]`.
- `write_array_hw(u_long chip, u_long offset, u_long data[], u_long size)` : This function writes an array of data at `data[]` to the SBus device.

- `io_image_hw(u_long chip, u_long offset, u_long din[], u_long dout[], u_long sizeX, u_long sizeY, struct delays timing)` : This function writes a two-dimensional array of image pixel data at `din[]` to the Sbus device and at the same time reads back the results and stores in `din[]`. The data structure `delays` contains the information about the timing of the image data communication between the SBus device (Fig.C.2):

  - `u_long pixel_delay, line_delay` : For filtering applications, the output pixel location must be shifted by a constant amount from the corresponsing input pixel location to implement a zero-phase filtering (FIg.C.2). For instance, let us consider a convolution of three consecutive pixels in the horizontal direction:

    $$y(i,j) = (x(i,j-1) + x(i,j) + x(i,j+1))/3. \tag{C.1}$$

    If we feed the input pixels in the raster scan format (Fig.C.2), the input pixel at $(i, j+1)$ has to be provided in order to calculate the output pixel at $(i, j)$ (input pixel at $(i, j-1)$ has already been provided two sampling periods earlier). Therefore, when feeding the input pixel at the $(i, j)$, the output pixel has to be stored at $(i, j-1)$. `pixel_delay` indicates the number of pixels to be shifted to the left direction. Likewise, `line_delay` indicartes the number of pixels to be shifted to the upward direction.

  - `u_long pipe_delay` : This is to indicate the latency of the bit-serial pipeline. It is measured in terms of sampling periods and not clock cycles.

  - `u_long clock[]` : This integer array indicates the number of event clock cycles to be generated for each read and write operations. In Fig.C.2, 5 event clock cycles need to be generated between the end of a write operation and the beginning of a read operation. Therefore, `clock[0] = 5`. Next, 3 event clock cycles need to be generated between the end of a read operation and the beginning of a write operation. Therefore, `clock[1] = 3`.

# Appendix D

# C++ code descriptions from Chapter 5

```
 ---
{



     --

            {
                {
               {

             }
              {

             }
               {

             }
              {

             }
         }



     }
}
```

Figure D.1: C++ code for 1D FIR filter.

```
void 2D_FIR_filter(int in[MAX1][MAX2], int out[MAX1][MAX2], double a[CMAX][CMAX],
      int m, int n)
{
      Interface bus;
      Single x[n][m], y;
      Double w[n][m];
      set_signal_precision(8);
      LINE_SIZE = MAX2;
      for(int i = 0; i < MAX1; i ++){
            for(int j = 0; j < MAX2; j ++){
                  for(int k = 0; k <= n; k ++){
                        for(int l = m; l >= 0; l -){
                              if(l == 0){
                                    if(k == 0){
                                          x[k][l] = bus.read(in[i][j]);
                                          w[k][l] = x[k][l] * a[k][l] + w[k][l + 1];
                                    }
                                    else {
                                          x[k][l] = line_delay(x[k - 1][l]);
                                          w[k][l] = x[k][l] * a[k][l] + w[k][l + 1] + w[k - 1][l];
                                    }
                              }
                              else {
                                    x[k][l] = delay(x[k][l - 1]);
                                    if(l == m){
                                          w[k][l] = x[k][l] * a[k][l];
                                    }
                                    else {
                                          w[k][l] = x[k][l] * a[k][l] + w[k][l - 1];
                                    }
                              }
                        }
                  }
                  y = round(w[n][m]);
                  bus.write(out[i][j]) = y;
                  clocker();
            }
      }
}
```

Figure D.2: C++ description of 2D FIR filter. Line delays are implemented in the off-chip SRAM. Up to 7 line delays are implemented in one SRAM.

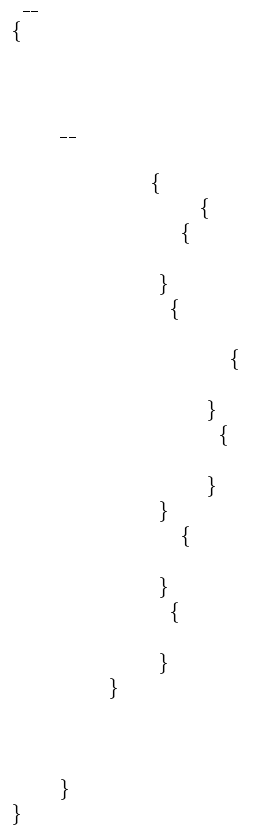


Figure D.3: C++ description of the 1D IIR filter. With this implementation, the sampling period may become longer than the signal word length due to the cyclic data dependencies of the IIR filter structure.

```
void adaptive_filter(int in[MAX], int out[MAX], int ref[MAX], int size, double mu)
{
      Single x[size + 1], y, d;
      Double w[size + 1], a[size + 1], e, dd[size + 1];
      for(int t = 0; t < MAX; t ++){
            for(int k = 0; k <= size; k ++){
                  if(k == 0){
                        x[k] = bus.read(in[t]);
                        w[k] = x[k] * delay(a[k]);
                  }
                  else {
                        x[k] = delay(x[k - 1]);
                        w[k] = x[k] * delay(a[k]) + w[k - 1];
                  }
                  y = round(w[size]);
                  bus.write(out[i]);
                  d = bus.read(ref[i]);
                  e = round(mu * (d - y));
            for(k = 0; k <= size; k ++){
                  dd[k] = e * x[k];
                  a[k] = round(delay(a[k]) + dd[k]);
            }
      }
}
```

Figure D.4: C++ code for adaptive FIR filter. This implementation severely suffers the throughput bottleneck because of the long critical path.

```
void adaptive_filter(int in[MAX], int out[MAX], int ref[MAX], int size,
double mu, int delay)
{
      int j = size;
      for(int height = 0; j; height ++, j >>= 1){} // calculate the height of the adder tree
      Single x[m + 1], y;
      Double w[height + 1][m + 1], a[m + 1], e, dd[delay][m + 1];
      for(int t = 0; t < MAX; t ++){
            for(int k = 0; k <= m; k ++){
                  if(k == 0){
                        x[k] = bus.read(in[t]);
                  }
                  else {
                        x[k] = delay(x[k - 1]);
                  }
                  w[0][k] = x[k] * delay(a[k]);
            }
            j = size;
            for(k = 0; k <= height; k ++){ // The adder tree
                  for(int l = 0; l < (j >> 1); l ++){
                        w[k][l] = w[k - 1][l * 2] + w[k - 1][l * 2 + 1];
                  }
                  if(j % 2){
                        w[k][l] = w[k - 1][l * 2];
                  }
                  j = (j >> 1) + j              }
            y = round(w[m][0]);
            bus.write(out[i]);
            d = bus.read(ref[i]);
            e = round(mu * (d - y));
            for(k = 0; k <= m; k ++){
                  for(int i = 0; i < delay; i ++){
                        if(i == 0){
                              dd[i][k] = e * x[m];
                        }
                        else{
                              dd[i][k] = delay(dd[i - 1][k]);
                        }
                  }
                  a[k] = round(delay(a[k]) + dd[delay - 1][k]);
            }
      }
}
```

Figure D.5: C++ code for adaptive filter with two modifications from the earlier implementation. First, the summation structure is changed from a cascaded adder structure to an adder tree structure which reduces the latency of the critical loop. Second, the feedback for updating coefficients has a larger cycle delay value. This give more time to complete the computation along the critical loops.

```
-
{

        {
    }
}
```

Figure D.6: C++ code for inverse discrete cosine transform using 20 multiplications.

```
--
        -
{

    -

    -

        {
    }
}
```

Figure D.7: Chang's IDCT implementation using 14 multiplcations.

```
void IDCT(int in[MAX1][MAX2], out[MAX1][MAX2], double cosine[])
{
      set_signal_precision(16);
      Single f[8], ff[8];
      Double a[12], b[8], c[8], aa[2];
      Interface bus;
      LINE_SIZE = MAX2;
      int inter[2][8][8]; // this is for storing the intermediate result.

      for(int i = 0; i < MAX1; i += 8){
            for(int j = 0; j < MAX2; j += 16){
                  for(int k = 0; k < 2; k ++){ // k=0: horizontal IDCT, k=1: vertical IDCT
                        for(int l = 0; l < 2; l ++){ // two blocks at a time
                              for(int m = 0; m < 8; m ++){
                                    for(int n = 0; n < 8; n ++){
                                    if(k == 0){
                                          f[n] = bus.read(in[i + m][j + n + l * 8]);
                                    }
                                    else {
                                          f[n] = bus.read(inter[l][m][n]);
                                    }
                                    }
                                    IDCT_core(f, ff, a, b, c, aa);
                                    for(n = 0; n < 8; n ++){
                                          if(k == 0){
                                                bus.write(inter[l][n][m]) = ff[n];
                                          }
                                          else {
                                                bus.write(out[i + m][j + n + l * 8]) = ff[n];
                                          }
                                    }
                              }
                        }
                  }
            }
      }
}
```

Figure D.8: C++ code for 2D inverse discrete cosine transform. The necessary data buffering is done at the host computer.Because of the pipeline latency, two blocks are grouped together and processed sequentially in order to avoid idle cycles. On the first run, 8-point IDCT is performed in the horizontal direction. On the second run, it is performed in the vertical direction.

```
void IDCT(int in[MAX1][MAX2], out[MAX1][MAX2], double cosine[])
{
      set_signal_precision(16);
      Single f[8], ff[8];
      Double a[12], b[8], c[8], aa[2];
      Interface bus;
      Ram ram(16); // RAM has 16 data bits. Intermediate results are stored here.
      LINE_SIZE = MAX2;

      for(int i = 0; i < MAX1; i += 8){
            for(int j = 0; j < MAX2; j += 16){
                  for(Control k = 0; k < 2; k ++){ // k=0: horizontal IDCT, k=1: vertical IDCT
                        for(Control l = 0; l < 2; l ++){ // two blocks at a time
                              for(Control m = 0; m < 8; m ++){
                                    for(Control n = 0; n < 8; n ++){
                                    if(s0 == 0){
                                          f[n + 0] = bus.read(in[i + m][j + n + l * 8]);
                                    }
                                    else {
                                          f[n + 0] = ram.read(l, m, n);
                                    }
                                    }
                                    IDCT_core(f, ff, a, b, c, aa);
                                    for(n = 0; n < 8; n ++){
                                          if(l == 0){
                                                ram.write(l, n, m) = ff[n + 0];
                                          }
                                          else {
                                                bus.write(out[i + n][j + m + l * 8]) = ff[l + 0];
                                          }
                                    }
                              }
                        }
                  }
            }
      }
}
```

Figure D.9: C++ code for 2D inverse discrete cosine transform using external SRAM for data buffering for the intermediate results after the horizontal 8-point IDCT.

```
void neuron(int size[], int layers)
{
      int max_size = 0;
      for(i = 0; i < layers; i ++){
            if(max_size < size[i]) {
                  max_size = size[i];
            }
      }
      set_signal_precision(8);

      Single x[layers][max_size], s[layers][max_size];
      Ext_Double y[layers][max_size][max_size + 1];
      Wildcard a[layers][max_size][max_size];

      Rom rom[layers][max_size / PERIOD + 1];
      Interface in[size[0] / PERIOD + 1], out[size[layers - 1] / PERIOD + 1];

      int ROM_SHARING = PERIOD;

      for(int l = 0; l < layers; l ++){
            for(int j = 0; j < size[l]; j ++){
                  if(l > 0) {
                        for(int i = 0; i < size[l - 1]; i ++) {
                              if(i == 0){
                                    y[l][i][j] = s[l - 1][i] * a[l][i][j];
                              }
                              else {
                                    y[l][i][j] = s[l - 1][i] * a[l][i][j] + y[l][i - 1][j];
                              }
                        }
                        x[l][j] = saturate(round(y[l][size[l - 1]][j]));
                        s[l][j] = rom[l][j / PERIOD].read(x[l][j]);
                  }
                  else {
                        s[l][i] = in[i / PERIOD].read();
                  }
                  if(l == layers - 1){
                  else {
                        out[j / PERIOD].write() = s[l][j];
                  }
            }
      }
}
```

Figure D.10: C++ description of a digital neural network.

# Bibliography


[1] L. Agarwal, Mike Wazlowski and S. Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[2] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes and G. Snider, "Teramac - Configurable Custom Computing," *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, 1995.

[3] J. Arechabala, E. I. Boemo, J. Meneses, F. Moreno and C. Lopez Barrio, "Full Systolic Binary Multiplier," *IEE Proceedings-G*, pp.188-190, April 1992.

[4] J. M. Arnold and D. A. Buell, "VHDL Programming on Splash 2," *Proc. Workshop on Field-Programmable Logic and Applications*, 1993.

[5] J. M. Arnold, D. A. Buell and E. G. Davis, "Splash 2," *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pp.316-322, 1992.

[6] S. L. Bade and B. L. Hutchings, "FPGA-Based Stochastic Neural Networks - Implementation," *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[7] W. Balakrishnan and N. Burgess, "Very-High-Speed VLSI 2s-Complement Multiplier Using Signed Binary Digits," *IEE Proceesings-E*, pp.29-34, 1992.

[8] P. Bertin, D. Roncin and J. Vuillemin, "Programmable Active Memories - Performance Measurements," *Proc. ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1992.

[9] P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[10] A. Bet-Shliemoun, N. -T. Shen, D. Malladi, D. Towne and A. Tam, "An MCM-D Memory Module in an Overmolded Plastic Package," *Proc. IEEE Multi-Chip Module Conference*, pp.34-39, 1995.

[11] G. Bi and E. V. Jones, "High-Performance Bit-Serial Adders and Multipliers," *IEE Proceedings-G*, pp.109-113, Feb. 1992.

[12] R. B. Boppana, "Eigenvalues and Graph Bisection : An Average-Case Analysis," *Proc. IEEE Symp. Found. Computer Sci.*, pp.280-285, 1987.

[13] B. Box, "Field Programmable Gate Array Based Reconfigurable Preprocessor," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.40-48, 1994.

[14] R. P. Brent, "The Area-Time Complexity of Binary Multiplcation," *Jo. Ass. Comp. Mach.*, pp.521-534, 1981

[15] F. Brewer and D. Gaijski, "Chippe: A System for Constraint Driven Behavioral Synthesis," *IEEE Trans. Computer-Aided Design*, pp.681-695, 1990.

[16] D. R. Bull and G. Wacey, "Bit-Serial Digital Filter Architecture Using RAM-Based Delay Operators," *IEE Proc. Circuits Devices Systs.*, pp.371-376, Oct. 1994.

[17] S. Casselman, "Virtual Computing and The Virtual Computer," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.43-48, 1993.

[18] F. Catthoor and H. J. De Man, "Application-Specific Architectural Methodologies for High-Throughput Digital Signal and Image Processing," *IEEE Trans. Acoustics, Speech, and Signal Processing*, pp.339-349, 1990.

[19] P. K. Chan, M. Schlag and M. Martin, "BORG: A reconfigurable Prototyping Board Using Field-Programmable Gate Arrays," *Proc. ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp.47-51, Feb. 1992.

[20] P. Chan, M. Schlag, and J. Zien, "Spectral K-Way Ratio-Cut Partitioning and Clustering," *Proc. 3th ACM/IEEE Design Automation Conference*, 1993.

[21] P. Chan, M. Schlag, "Spectral-Based Multi-Way FPGA Partitioning," *Proc. ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp. 133-139, Feb. 1995.

[22] C. -K. Cheng and Y. -C. A. Wei, "An Improved Two-Way Partitioning Algorithm with Stable Performance," *IEEE Trans. Computer-Aided Design*, pp.1502-1511, Dec. 1991.

[23] J. Cong, Z. Li and R. Bagrodia, "Acyclic Multi-Way Partitioning of Boolean Networks," *Proc. 31st ACM/IEEE Design Automation Conference*, pp.670-675, 1994.

[24] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," *MIT Press*, 1989.

[25] C. E. Cox and W. E. Blanz, "GANGLION - A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier," *IEEE Jo. Solid-State Circuits*, pp.288-299, 1992.

[26] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.142-151, April 1993.

[27] J. Darnauer, P. Garay, T. Isshiki, J. Ramirez and W. W. -M. Dai, "A Silicon-On-Silicon Field Programmable Multichip Module (FPMCM) - Integrating FPGA and MCM Technologies," *IEEE Trans. Components, Packaging, and Manufacturing Technology, part B: Advanced Packaging*, pp.601-609, Nov. 1995.

[28] J. Darnauer, P. Garay, T. Isshiki, J. Ramirez and W. W. -M. Dai, "A Field Programmable Multi-Chip Module (FPMCM)," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.1-10, April 1994.

[29] J. Darnauer, P. Garay, T. Isshiki, J. Ramirez and W. W. -M. Dai, "Field Programmable Multi-Chip Module (FPMCM) - An Integration of FPGA and MCM Technology," *Proc. IEEE Multi-Chip Module Conference*, pp.50-55, 1995.

[30] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note and J. Huisken, "Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms," *Proceedings of the IEEE*, pp.319-335, 1990.

[31] S. Devadas and R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, pp.768-781, 1989.

[32] Wilm E. Donath, "Placement and Average Interconnection Lengths of Computer Logic," *IEEE Trans. Circuits and Systems*, pp.272-277, April 1979.

[33] M. F. Dossis, J. M. Noras and G. J. Porter, "Custom Co-Processor Compilation," *Workshop on Field-Programmable Logic and Applications*, 1993.

[34] Abba A. El Gamal, "Two-Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits," *IEEE Trans. Circuits and Systems*, pp.127-138, Feb. 1981.

[35] J. G. Eldredge and B. L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, 1994.

[36] M. D. Ercegovac, T. Lang, "Digital Systems and Hardware/Firmware Algorithms," 1985.

[37] Aaron T. Ferrucci, "ACME: A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network," *Master's Thesis, UC Santa Cruz*, 1994.

[38] C. M. Fiduccia and R. M. Mattheyses, "A Lenear-Time Heuristic for Improving Network Partitions," *Proc. 19th ACM/IEEE Design Automation Conference*, pp.241-247, 1982.

[39] J. Frankle and R. M. Karp, "Circuit Placement and Cost Bounds by Eigenvector Decomposition," *Proc. IEEE Conference on Computer-Aided Design*, pp.414-417, 1986.

[40] Robert C. Frye, "Balancing Performance and Cost in CMOS-Based Thin Film Multichip Module," *Proc. IEEE Multichip Module Conference*, pp. 6-11, 1993.

[41] R. C. Frye, K. L. Tai, M. Y. Lau and A. W. C. Lin, "Low-cost silicon-on-silicon MCMs with integrated passive components," *Proc. 1992 International Electronics Packaging Conference*, pp. 343, 1992.

[42] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1995.

[43] R. A. Haddad and Thomas W. Parsons, "Digital Signal Processing : Theory, Applications, and Hardware," *Computer Science Press*, 1991.

[44] L. Hagen, A. B. Kahng, F. J. Kurdahi, C. Ramachandran, "On the Intrinsic Rent Parameter and Spectra-Based Partitioning Methodologies," *IEEE Trans. Computer-Aided Design*, pp.27-37, Jan. 1994.

[45] L. Hagen and A. B. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering," *IEEE Trans. Computer-Aided Design*, pp.1074-1085, Sept. 1992.

[46] B. J. Han, S. Das, R. C. Frye, K. L. Tai and M. Y. Lau, "A Silicon-on-Silicon MCM for Voice Recognition/Telephone Answering Device," *Proc. IEEE Multichip Module Conference*, pp. 45-48, 1995.

[47] H. J. Herpel, N. Wehn, M. Gasteier and M. Glesner, "A Reconfigurable Computer for Embedded Control Applications," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.111-120, 1993.

[48] Dzung T. Hoang, "Searching Genetic Databases on Splash 2," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–191, April 1993.

[49] X. Hu, S. C. Bass and R. G. Harber, "Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems," *IEEE Trans. Computer Aided Design*, pp. 1441-1449, Dec. 1994.

[50] C. T. Hwang and J. H. Lee, "A Formal Approach to the Scheduling Problem in High Level Synthesis," *IEEE Trans. Computer-Aided Design*, pp.464-475, 1991.

[51] James P. Ignizio, "Linear Programming in Single- & Multiple-Objective Systems," *Prentice-Hall*, 1982.

[52] P. Ienne and M. A. Viredaz, "Bit-Serial Multipliers and Squarers," *IEEE Trans. Computers*, pp. 1445-1450, Dec. 1994.

[53] C. Iseli and E. Sanchez, "A C++ Compiler for FPGA Custom Execution Unit Synthesis," *IEEE Symp. FPGAs for Custom Computing Machines*, 1995.

[54] J. Isoaho, J. Nousianen and O. Vainio, "FPGA-implementable Digital Filters," *Workshop on Field-Programmable Logic and Applications*, 1993.

[55] T. Isshiki and W. W. -M. Dai, "High-Peroformance Datapath Implementation on Field-Programmable Multi-Chip Module (FPMCM)," *Proc. Workshop on Field Programmable Logic and Applications*, Sept. 1994.

[56] T. Isshiki and W. W.-M. Dai, "Field-Programmable Multi-Chip Module (FPMCM) for High-Performance DSP Accelerator," *IEEE Asia-Pacific Conference on Circuits and Systems*, Dec. 1994.

[57] T. Isshiki and W. W. -M. Dai, "High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems," *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Feb. 1995.

[58] R. Jain, F. Catthoor, J. Vanhoof, B. J. S. De Loore, G. Goossens, N. F. Goncalvez, L. J. M. Claesen, J. K. J. Van Ginderdeurn, J. Vandewalle and H. J. De Man, "Custom Design of a VLSI PCM-FDM Transmultiplexer from System Specifications to Circuit Layout Using a Computer-Aided Design System," *IEEE Jo. Solid-State Circuits*, pp.73-84, 1986.

[59] A. Jantsch, P. Ellervee, J. Oberg and A. Hemani, "A Case Study on Hardware/Software Partitioning," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[60] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, pp.76-80, 1970.

[61] Isreal Koren, "Computer Arithmetic Algorithms," 1993.

[62] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks," *IEEE Trans. Computers*, pp.438-446, 1984.

[63] S. Kumar, K. Forward and M. Palaniswami, "A Fast-Multiplier Generator for FPGAs," *Proc. IEEE Int. Conference on VLSI Design*, pp.53-56, 1995.

[64] B. Landman and R. Russo, "On a Pin Versus Block Relationship for Partitioning Logic Graphs," *IEEE Trans. Computers*, pp.1469-1479, 1971.

[65] M. Leeser, R. Chapman, M. Bertone and A. Wenban, "Implementing Filters with Programmable Logic," *Workshop on Field-Programmable Logic and Applications*, 1993.

[66] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *J. VLSI and Comput. Syst.*, vol. 1, no. 1, pp. 41-67. Oct. 1983.

[67] James D. Lyle, "SBus : Information, Applications, and Experience," *Springer-Verlag*, 1992.

[68] R. E. Lyon, "Two's Complement Pipeline Multipliers," *IEEE Trans. Computers*, pp.418-425, 1976.

[69] V. Maheshwari, J. Darnauer, J. Ramirez and W. W. -M. Dai, "Design of FPGAs with Area I/O for Field Programmable MCM," *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 1995.

[70] "MC92005 User's Manual," *Motorola*, 1992.

[71] L. McMurchie and C. Ebeling, "Wirec Manual," *University of Washington*, April, 1991.

[72] J. Moran, I. Rios and J. Meneses, "Signed Digit Arithmetic on FPGAs," *Proc. Workshop on Field Programmable Logics and Applications*, 1993.

[73] A. F. Murray and P. B. Denyer, "A CMOS Design Strategy for Bit-Serial Signal Processing," *IEEE Jo. Solid-State Circuits*, pp.746-753, 1985.

[74] H. Nakada, K. Yamada, A. Tsutsui and N. Ohta, "A Design Method for Realising Real-time Circuits on Multiple-FPGA Systems," *Proc. Workshop on Field-Programmable Logic and Applications*, 1993.

[75] S. Note, F. Catthoor, G. Goossens, H. J. De Man, "Combined Hardware Selection and Pipelining in High-Performance Data-Path Design," *IEEE Trans. Computer-Aided Design*, pp.419-423, 1992.

[76] D. A. Patterson and J. L. Hennessy, "Computer Architecture : A Quantitative Approach," 1990.

[77] Greg Perry, "Moving from C to C++: Ins-and-Outs of Object-Oriented Programming," *Sams Publishing*, 1992.

[78] Kamisetty Ramamohan Rao, "Discrete Cosine Transform : Algorithms, Advantages, Applications," *Academic Press*, 1990.

[79] K. Roy-Neogi and C. Sechen, "Multiplie FPGA Partitioning with Performance Optimization," *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 1995.

[80] Laura A. Sanchis, "Multiple-Way Network Partitioning," *IEEE Trans. Computers*, pp.62-81, Jan. 1989.

[81] M. Schlag, J. Kong and P. K. Chan, "Routability-Driven Technology Mapping for Lookup Table-Based FPGA's," *IEEE Trans. Computer-Aided Design*, pp.13-26, Jan. 1994.

[82] H. Schmit, L. Arstein, D. Thomas and E. Lagnese, "Behavioral Synthesis for FPGA-based Computing," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.125-132, 1994.

[83] D. G. Schweikert and B. W. Kernighan, "A Proper Model for the Partitioning of Electronic Circuits," *Proc. 9th ACM/IEEE Design Automation Workshop*, pp.57-62, 1972.

[84] Richard G. Shoup, "Parameterized Convolution Filtering in an FPGA," *Proc. Workshop on Field-Programmable Logic and Applications*, 1993.

[85] S. Singh and P. Bellec, "Virtual Hardware for Graphics Applications Using FPGAs," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.49-58, 1994.

[86] "SLIC Evaluation Board User's Guide for DAWN VME PRODUCTS SLIC EB-1," *DAWN VME PRODUCTS*, 1993.

[87] T. Tanaka, T. Kobayashi and O. Karatsu, "HARP: Fortran to Silicon," *IEEE Trans. Computer-Aided Design*, pp.649-660, 1989.

[88] Richard Terrill, "A Novel 2-Sided Multi-Chip-Module used to Create a 50,000 Gate Programmable Logic Device," *Proc. IEEE Multi-Chip Module Conference*, pp.56-60, 1995

[89] Steve M. Trimberger, "Field-Programmable Gate Array Technology," *Kluwer Academic Publishers*, 1994.

[90] D. E. Van den Bout, "The Anyboard: Programming and Enhancements," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp.68-76, April 1993.

[91] Jan Vanhoof, et al., "High-Level Synthesis for Real-Time Digital Signal Processing," *Kluwer Academic Publishers*, 1993.

[92] J. Varghese, M. Butts and J. Batcheller, "An Efficient Logic Emulation System," *IEEE Trans. Very Large Scale Integration Systems*, Vol.1, No.2, pp.171-174, June 1993.

[93] M. Wahab and D. Puckey, "FPGA-Based DSP Systems," *Proc. Int. Workshop on Field Programmable Logic and Applications*, 1993.

[94] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, "PRISM-II Compiler and Architecture," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[95] Y. -C. Wei and C. -K. Cheng, "Ratio Cut Partitioning for Hierarchical Designs," *IEEE Trans. Computer-Aided Design*, pp.911-921, July 1991.

[96] Ulrich Weinmann, "FPGA Partitioning under Timing Constraints", *Proc. Workshop on Field-Programmable Logic and Applications*, 1993.

[97] "The Programmable Gate Array Data Book," Xilinx, 1994.