AN IMPROVED DATA STREAMING MODEL FOR SYNTHETIC SYSTEM-ON-CHIP BENCHMARK CIRCUIT GENERATION

by

Jankiben Patel
B.E, Gujarat University, 1998

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

In the
School of Engineering Science

© Jankiben Patel 2010

SIMON FRASER UNIVERSITY

Summer 2010

APPROVAL

Name:	**Jankiben Patel**
Degree:	**Master of Engineering**
Title of Project:	**An improved data streaming model for Synthetic System-on-Chip benchmark circuit generation**

Examining Committee:

Chair:

Dr. Rick Hobson, P.Eng
Professor, School of Engineering Science

Dr. Lesley Shannon, P.Eng
Senior Supervisor
Assistant Professor, School of Engineering Science

Dr. Marek Syrzycki
Supervisor
Professor, School of Engineering Science

Dr. Glenn Chapman, P.Eng
Internal Examiner
Professor, School of Engineering Science

Date Defended/Approved: July 20, 2010

ABSTRACT

Field Programmable Gate Array (FPGA) researchers aim to improve the quality of the Computer-Aided Design (CAD) tools used to map designs onto FPGAs and evaluate different possible architectures. To test new architectures and CAD tools, researchers need to use benchmark circuits representative of realistic applications, which are not freely available. Therefore, researchers have considered randomly-generated benchmark circuits that can model the complexities of real systems. We use the existing Benchmark Circuit Generator (BCGEN), which generates circuits with System-on-Chip (SoCs) architectures. While BCGEN provides a good framework for circuit generation, the existing dataflow communication patterns have some limitations. Specifically, they have large numbers of inputs-outputs (I/Os) which is not scalable, and do not support data-buffering. We aim to improve BCGEN's dataflow communication model by reducing the number of I/Os. Hence, they will be scalable and include data buffering capabilities between logic stages which is typical for data-streaming applications.

Keywords: Field Programmable Gate Array; Computer-Aided Design; System-on-Chip; Synthetic Benchmarks

ACKNOWLEDGEMENTS

I am heartily thankful to my supervisor Dr. Lesley Shannon for this project. I would have been lost without her kind support and knowledge. I would also like to thank Dr. Steve Wilton (University of British Columbia) and his student Cindy Mark for sharing their work on the Benchmark Circuit Generator and for their permission to carry on the project which made this work successful.

I am also thankful to my husband Jasbir and my son Shubham for supporting me throughout my work and dealing with my absence from home to make this work successful. I would also like to thank to my all lab members (especially David, Jason and Jian) for their kind support in different fronts whenever I needed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ASIC	Application-Specific Integrated Circuit
BCGEN	Benchmark Circuit Generator
BLIF	Berkeley Logic Interchange Format
CAD	Computer-Aided Design
DEMUX	Demultiplexer
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
I/O	Input/Output
IP	Intellectual Property
LUT	Look-Up Table
MCNC	Microelectronic Center of North Carolina
MUX	Multiplexer
NoC	Network-on-Chip
SoC	System-on-Chip

1: CHAPTER 1

Introduction

Field Programmable Gate Arrays (FPGAs) are Integrated Circuits (ICs) that can be configured by the user after manufacturing. The user describes a design using a Hardware Description Language (HDL), such as VHDL or verilog, which runs through a Computer Aided Design (CAD) flow to generate a bitstream for the FPGA configuration.

FPGAs have become a very popular implementation technology for digital circuits, an attractive alternative to Application-Specific Integrated Circuits (ASICs) due to their lower volume costs. ASICs are mainly used for implementing a high volume/high speed/low power designs, as FPGAs now have sufficient logic density to implement System-on-Chips (SoCs). FPGAs are used in many applications, including wireless applications [1], biomedical imaging [2], digital signal processing [3], automotive electronics [4] and computer vision [5].

In the past two decades, the logic capacity of FPGAs has increased dramatically. The FPGA market increased from $1.9 billion in 2005 to $2.75 billion by the year 2010 [6]. This increase is due to their increased logic density, performance capabilities and lower development costs. The performance of circuits on a FPGA is based on the quality of the FPGA architecture, the FPGA's hardware fabric, and quality of the Computer-Aided Design (CAD) tools used to map circuits onto the FPGA.

FPGA CAD tool research aims to improve the quality of the final design mapped onto a FPGA via improving the algorithms that convert the HDL design into the final bitstream. Similarly, new FPGA architecture is analyzed using these CAD tools. CAD tools are used to implement different circuits to the new FGPA architecture. To evaluate new CAD algorithms and architectures, a number of test circuits required, known as benchmark circuits. The benchmark circuits are mapped, placed and routed onto the different FPGA architectures using different CAD algorithms. The quality of the architecture is evaluated based on the area and delay measurement for the different benchmark circuits.

1.1 Motivation

There are many benchmark circuits available that can be implemented on an FPGA. Many vendors have their own large databases of such benchmark circuits. However, such vendor specific circuits are not freely available to researchers and access to the source code is restricted. To overcome this limitation, randomly generated benchmark circuits can be used for research. In such circuits, random netlists are generated from parameters specified by the user. Large numbers of test circuits can be generated using a benchmark circuit generator because they do not require any design time. Previously introduced benchmark circuit generators are GEN [7] and GNL [8]. More recently, Benchmark Circuit Generator (BCGEN), a stochastical benchmark circuit generator developed by Cindy Mark (UBC) [9], was created to generate circuits that are designed to emulate Systems-on-Chip (SoC). BCGEN creates circuits by connecting intellectual property (IP) modules, such as processors and memory, using different communication patterns, including bus, star and dataflow communication patterns.

The BCGEN tool provides a good framework for generating benchmark circuits. However, there are limitations when generating circuits with a dataflow communication pattern. In the existing dataflow communication pattern, the number of Inputs and Outputs (I/Os) in the final circuit increases dramatically, and there is no support for data buffering. Hence, our aim is to reduce the number of inputs and outputs and augment the BCGEN tool to include the data buffering capability in the dataflow communication pattern of the BCGEN tool.

1.2 Objective

The main objective of this project is to improve the representation of the dataflow system in BCGEN by including data buffering capabilities and minimizing I/O growth. Limiting the number of I/Os is achieved by inserting multiplexer and demultiplexer nodes between the original nodes (stages). Additionally, inserting a FIFO (First-In-First-out) between the original nodes provides communication media for data storage, which enables data buffering between stages.

The original BCGEN tool has the following characteristics:

- The final circuits are generated by stitching modules together using different communication patterns such as star, bus or dataflow.
- The component circuits are selected from existing benchmarks (such as circuits from the MCNC benchmark suite) or from other existing generators.

- The final circuits are generated based on either user selectable parameters or default parameters.
- The generator is able to handle combinations of multiple communication patterns in a single circuit.

Our modifications to the existing generator aim to improve the dataflow modelled circuits.

The characteristics of the modified dataflow pattern are:

- Multiplexer and Demultiplexer nodes are used in between the existing nodes to reduce the I/O count.
- FIFO modules are inserted in between the existing nodes (stages) to implement data buffering, typically found in dataflow designs.

1.3 Contribution

Contributions for this work are as follows:

1) Firstly, multiplexer and demultiplexer modules are designed such that they are able to produce the desired number of output connections from the given number of input connections.

2) Secondly, FSL FIFO module is used for data buffering. Source code is obtained from Open Core [10].

3) Thirdly, a more accurate and scalable representation of the dataflow communication pattern is implemented by stitching multiplexer, demultiplexer and FIFO modules between the original modules.

The existing BCGEN tool is improved by integrating the above mentioned modifications. After successful implementation, the circuits generated using the improved BCGEN are tested using T-V pack and VPR [11]. The test results are compared with the circuits generated using the original BCGEN. The analysis showed that the circuits generated using the improved BCGEN has fewer I/O connections than the previously generated circuits (using original BCGEN).

1.4 Organization

The background information related to this work including the FPGA architecture, CAD tools and previous work in the synthetic circuit generator as well as detailed functionality of the BCGEN is described in Chapter 2. Chapter 3 details our updates for BCGEN to improve the dataflow circuit models. Finally, Chapter 4 covers system verification and testing followed by Chapter 5, which concludes this report and suggests possible future work.

2: CHAPTER 2

Background

This chapter provides an overview of FPGA architecture as well as information about its CAD flow. The chapter also summarizes previous work done in the synthetic benchmark circuit generators including BCGEN.

2.1 FPGA Architecture

FPGAs have three main components: logic blocks, programmable routing fabric and, input/output (I/O) blocks. Logic blocks are mainly used for implementing the combinational and sequential logic functionality for the desired circuit. Typically, a logic block contains Look-Up Tables (LUTs), Flip-Flops (FFs) and 2-to-1 multiplexers. Figure 2.1 shows a simple architecture of a logic block.

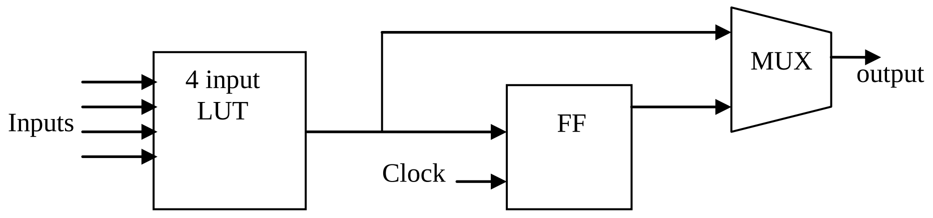

Figure 2.1: Architecture of a logic block [12]

Figure 2.1 shows a 4-input LUT, a Flip-Flop and a 2-to-1 multiplexer based logic block. An N input LUT is implemented as a N-bit addressable $2^N \times 1$ memory. Each single bit memory can store two different values. Therefore, 2^N bits can be used to store 2^{2^N} different combinations representing 2^{2^N} different logic functions. The flip-flop is used for implementing the sequential logic in a design. Finally, the multiplexer selects whether the implemented logical function is purely combinational or sequential. The programmable routing fabric is used to provide the interconnect between the logic blocks, or between the logic blocks and the I/O blocks. I/O blocks behave as an input or output pads as per the circuit requirement. FPGA architectures are mainly classified as:

- Island-Style FPGA
- Row-Based FPGA
- Hierarchical FPGA

As the Island-Style FPGA architecture (see Figure 2.2) is the most popular for researchers, we provide a detailed discussion in the following.

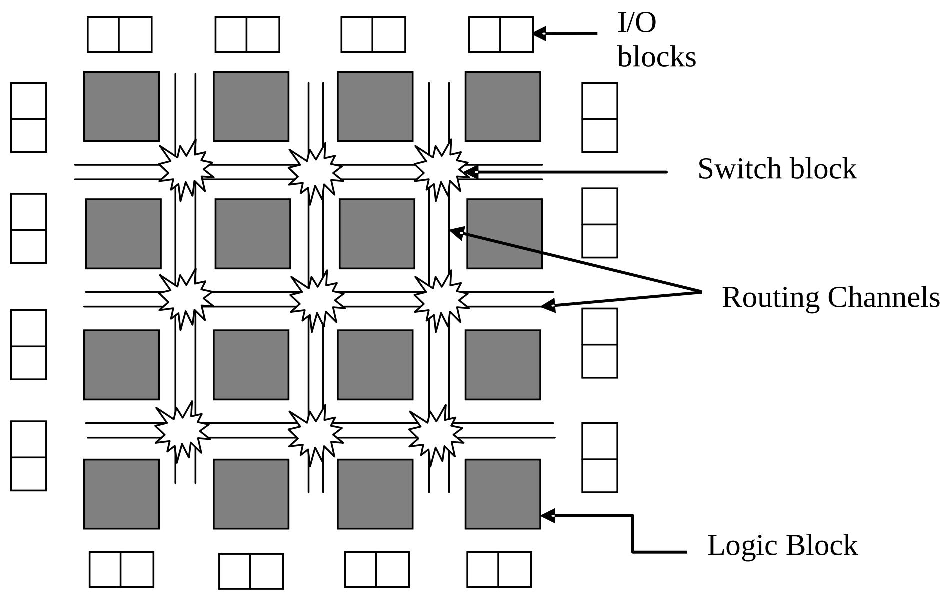

Figure 2.2: Architecture of an Island Style FPGA [12]

As shown in Figure 2.2, “channels” of routing wires surround the logic block on all sides (hence it is known as the “Island-Style FPGA”). Logic blocks are connected to the routing wires via a connection block (not shown in Figure 2.2). The switch block is a set of programmable switches, which allows the appropriate connection between a source and the related sinks (see Figure 2.2). Logic blocks are typically grouped in clusters and the connections made within these grouping are known as intra-cluster connections. Conversely, the connections made between different clusters are known as the *inter-cluster* connections. Generally, the intra-cluster connections are faster than *inter-cluster* connections. The following section describes how a user programs the device to implement a design.

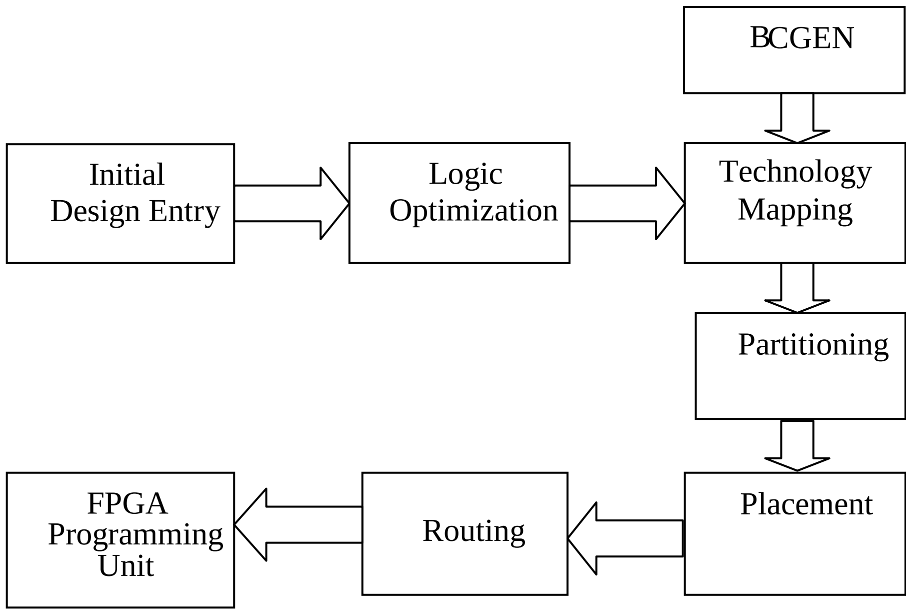

Figure 2.3: Typical CAD flow

2.2 CAD Tools

An FPGA user designs a digital circuit using either a Hardware Description Language (HDL) or schematic entry. Subsequently, the CAD tools are used for converting from this high level designs (HDL and schematic) into a bit stream file which is used to program the FPGA. Typical CAD tool flow is shown in Figure 2.3. In CAD tools, high level synthesis converts the VHDL/Verilog circuit into the register transfer level (RTL) description. Logic optimization performs required logic simplification. In the subsequent step, a netlist of logic gates is mapped into a netlist of logic blocks (LUTs + FFs) to implement the logic function, which is known as technology mapping. Then, the logic blocks are packed into the clusters. Next, the clusters are placed onto a virtual mapping of the device. The cluster placement is optimized such that the connection length between the connected clusters will be minimized. Once the location of the cluster has been decided, the router determines the specific routing resources (wire tracks and switches) that should be used to connect all the logic blocks' input and output pins as required by the circuit. Finally, the circuit is converted into the bit stream, which is used to program an FPGA.

Placement and routing are important aspects in CAD tools for FPGAs. There have been many algorithms developed for efficient placement and routing. To test these CAD algorithms accurately, large benchmark circuits are required to map onto an FPGA. Most researchers use the MCNC (Microelectronics Center of North Carolina) benchmark circuits for this purpose. However, the largest benchmark circuit is unrealistically smaller than the circuits implemented on modern commercial FPGAs. Other benchmark circuits are also available, but they also have size limitations. An efficient solution to this problem is to use randomly generated benchmark circuits. These randomly generated circuits behave like the real circuits. This study explores functionality of the BCGEN (existing benchmark circuit generator) and presents an improvement for the large dataflow circuits of the existing BCGEN. These circuits usually use a common format known as the Berkeley Logic Interchange Format (BLIF) to encapsulate the netlist of the circuit.

2.3 BCGEN

Mark [10] has developed a benchmark circuit generator (BCGEN) that generates the benchmark circuits by combining a number of randomly selected logic blocks from the given MCNC library. The benchmark circuit is generated using hierarchical SoC technique. In today's FPGAs, the implemented circuits contain Intellectual Property (IP) modules, including processor modules that are connected using buses and on-chip networks. Similarly, BCGEN generates the benchmark circuits that can have many types of IP modules connected using bus, star or dataflow network connection pattern. The circuits are generated using different hierarchical levels and different network connection patterns based on the user requirement. The library of MCNC benchmark circuits and the user constraint file are used for building the synthetic benchmark circuits.

The MCNC benchmark circuit library is divided into different sub-categories such as processors, interfaces, and controllers. The final circuit construction is done in three stages. In the first stage, the values for primary parameters such as hierarchical depth, number of networks, number of leaf modules and bus width are selected. These primary parameters are important to decide the shape and the size of the final circuit. In the second stage, the circuit is constructed by selecting the size and the network pattern to connect the modules at each hierarchical level. Finally, the leaf module pins are connected as appropriate to other module pins to realize the desired benchmark circuit model.

For the bus network type, one sub-module is selected as a master and all other sub-modules are selected as slaves. All these slave modules are connected to the master module using the bus network interface. In the star network type, one of the modules is selected as a head sub-module and all other modules are selected as tail sub-modules. The input and output communications between these modules are handled by an algorithm [13].

The dataflow network connection is illustrated in Figure 2.4. In this network pattern, connections are made between adjacent sub-modules. However, if a module has more output connections (e.g. module Q) than the input connections of the next module (e.g. module R), the additional connections can be skipped to the subsequent module (e.g. module T) because the sub-module (module T) has more input connections than the output connections of the preceding module (module S). Moreover, the connections can be made as a feedback loop between the stages (e.g. the Q and R sub-modules). Finally in the first/last stage, any unconnected I/Os (if feedback of stage skipping is not possible) are become primary I/Os for the final generated circuit. The resultant generated circuits are then validated using the T-V pack and VPR [11].

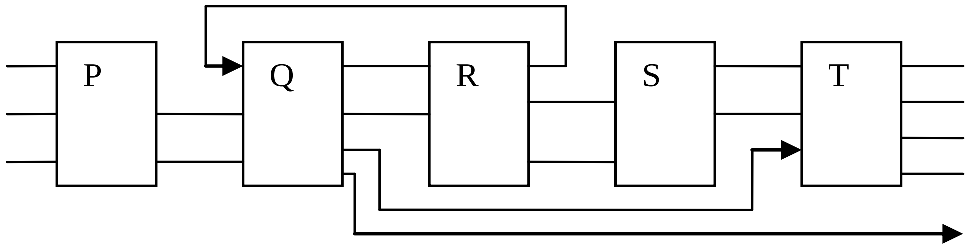

Figure 2.4: Dataflow connection in original BCGEN

2.4 Existing Synthetic Benchmark Circuit Generators

This section describes the properties of the existing synthetic benchmark circuit generators:

- GEN
- GNL
- Other generators

Now, each of these generators is discussed in detail.

2.4.1 GEN

GEN benchmark generator was originally designed by Hutton [7] in 1997. The CIRC tool analyzes the existing circuit and provides information about the circuit characteristics. The tool uses MCNC benchmark circuits as input circuits and measures the circuit properties in the form of a specification file. Subsequently, GEN uses this specification file as an input and builds a “clone” (circuit) according to the user specification. The post routing results of the cloned circuit are compared with the result of the original circuits using the VPR tool. Later on the GEN benchmark generator was modified to produce both combinational and sequential logic circuits [14]. Additionally, Kundarewich et al. [15] extended the GEN such that it can use partitioning information to develop the hierarchical circuits.

2.4.2 GNL

This generator, developed by Stroobandt et al. [8] in 1999, generates the benchmark circuit using an analytical method. This method is based on the rent rule and ratio of the circuit’s multi-terminal nets. The tool uses the user-defined constraints such as the value of rent parameter, the ratio of input and output pins, and the number of input and output pins. The generator uses bottom up clustering approach to generate circuits. However, the delay characteristics are not controlled using the tool.

2.4.3 Other Generators

The method described by Tom et al. [16] in 2005 generates larger circuits by randomly stitching the BLIF circuits. This method generates a circuit with a large number of inputs and outputs. Furthermore, Grant et al. [17] uses an edge swapping method to generate benchmark circuits. This method tries to preserve properties of the original circuits in the newly generated circuit. Moreover, Harlow’s method [18] generates a set of Binary Decision Diagrams (BDDs). One more generator, Partgen is developed by Pistorius et al. [19] in 2000. This generator generates circuits from the library, which already has several kind of circuits such as regular and combinational, memory, controller and irregular, and combinational.

3: CHAPTER 3

Data Streaming Model

In this chapter, the details of our contributions are discussed. The modifications done in the existing dataflow pattern of the BCGEN are also explained.

3.1 Updates in the BCGEN Data Streaming Model

As described in Chapter 2, the dataflow network of the original BCGEN (see Figure 2.4) is designed such that the additional connections to those directly connected sub-modules can be either skipped or connected as a feedback loop if input or output pins remain unconnected on the sub-module. Remaining unconnected inputs and outputs are converted to adjacent module’s primary input or output pins. Because of this implementation, the resultant circuit has a large number of I/Os. To restrict the number of I/Os in the resultant circuit, additional multiplexer and demultiplexer modules are inserted in between the existing sub-modules. Furthermore, data buffering capability is added by using Shift Register Logic First In-First Out (SRL FIFO) [10]. This results in a more accurate and scalable representation of the dataflow communication pattern by stitching multiplexers, demultiplexers, and FIFO modules between the original modules to represent the Data Streaming.

3.2 Details of data streaming modification

The proposed modification in the existing dataflow connection is illustrated in Figure 3.1.

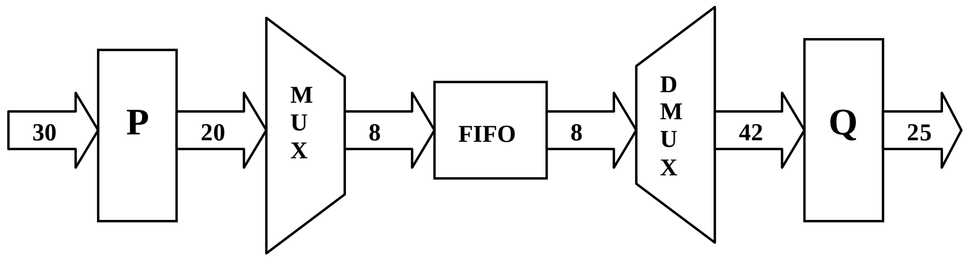

Figure 3.1: Proposed dataflow connection in the BCGEN

As shown in Figure 3.1, the P and Q modules are selected from the MCNC library for the dataflow network connection using the original BCGEN. However, our proposed modification inserts a multiplexer, demultiplexer and FIFO between the P and Q modules. The P module has 30 input pins and 20 output pins. On the other hand, the Q module has 42 input pins and 25 output pins. To connect modules P and Q using the dataflow connection (see Figure 3.1), the module Q has 22 more input connections than the output connections of the module P. In the original BCGEN, these additional input connections are connected to the additional output connections of any subsequent module that has more output connections than the following module or as primary outputs of the final circuit. In the original implementation, the former is the more likely, such that the connections are made by either feedback loop or skipping the modules.

In our approach, the feedback loops and skipping between the stages in the dataflow implementation are avoided. To simplify this, all the module outputs are first converted (using multiplexer or demultiplexer) into the user defined bus width. Similarly, all the module inputs are converted (using demultiplexer or multiplexer) from the bus width. Additionally, a FIFO is connected between these mux/demux or demux/mux modules that is designed according to the databus width. By using this approach, the inequality of the input and output connections between the modules can be efficiently handled without using feedback loops and skipping between stages. This reduces the number of input and output connections in the final circuit.

3.2.1 Multiplexer

The multiplexer module is designed such that the number of connections between two modules are reduced according to the system requirement. The multiplexer unit is mainly used where the output connections of a module are more than the input connections of a subsequent module. The multiplexer can be customized to obtain any number of output connections from the given number of input connections. For example, if two BLIF modules are selected from the MCNC library to join side by side using the BCGEN's data path algorithm. One module has 32 output pins and the subsequent module has 23 input pins. Hence, nine extra connections should be reduced to efficiently connect the two modules. This is easily achieved using different combinations of the multiplexers (as shows in Figure 3.2).

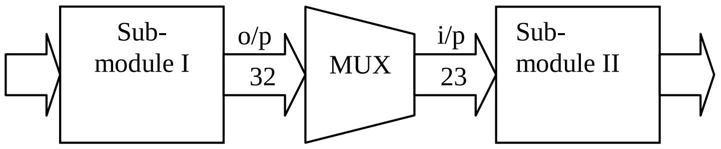

Figure 3.2: Modules connected using multiplexer

3.2.2 Demultiplexer

The demultiplexer module is designed such that it increases the number of connections between two modules according to the system requirement. The demultiplexer unit is mainly used where the number of output connections of a module is less than the input connections to subsequent module. Any number of output connections can be generated from the given number of input connections. For example, the demultiplexer is used if two BLIF modules are selected from the MCNC library to join side by side using the BCGEN's data path algorithm where one module has 23 output pins and the subsequent module has 32 input pins. Hence, additional nine connections should be generated to efficiently connect the two modules. This is easily achieved using different combinations of the demultiplexers (as shows in Figure 3.3).

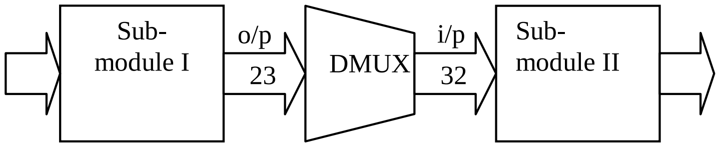

Figure 3.3: Modules connected using demultiplexer

3.2.3 SRL FIFO

Shift Register logic FIFO (SRL FIFO) is mainly used for data buffering and data overflow prevention. The VHDL code for the SRL FIFO is obtained from an open cores website [10]. This FIFO from the open core website is 8-bit wide and 32 bit long. However, it has been modified such that the user specified bus-width is used as the FIFO width in the proposed system. Furthermore, the clock and the reset signals of the FIFO are connected to the system clock and reset signals to synchronize with the over-all system.

4: CHAPTER 4

Updates to the BCGEN Code Base

This chapter gives a detailed description of the BCGEN software organization and highlights our contribution to BCGEN. The programming language for the original BCGEN is C, within which we made all our modifications to implement our proposed data streaming functionality. The updated source code of the modified BCGEN is given in the Appendix. We have added approximately 1165 lines of code in the original BCGEN to implement modified dataflow communication pattern and size of our code is approximately 41 KB.

4.1 Software Organization

Figure 4.1 describes how BCGEN generates synthetic circuits based on user constraints. As shown in the figure, BCGEN has two sets of inputs for the circuit generation, the first is the library of the MCNC benchmark circuits and the other is the user constraints file. The user is able to specify different primary parameters, such as the number of networks, the number of leaf modules, the hierarchy depth and bus width in the form of a user constraints file. These parameters describe the overall size and shape of the generated circuit. If the user does not provide any of these parameters, BCGEN uses the default parameters for the circuit's generation. The library of MCNC benchmarks contains different BLIF modules including processors, interfaces and controllers, which are used by BCGEN as the component modules for the final circuit.

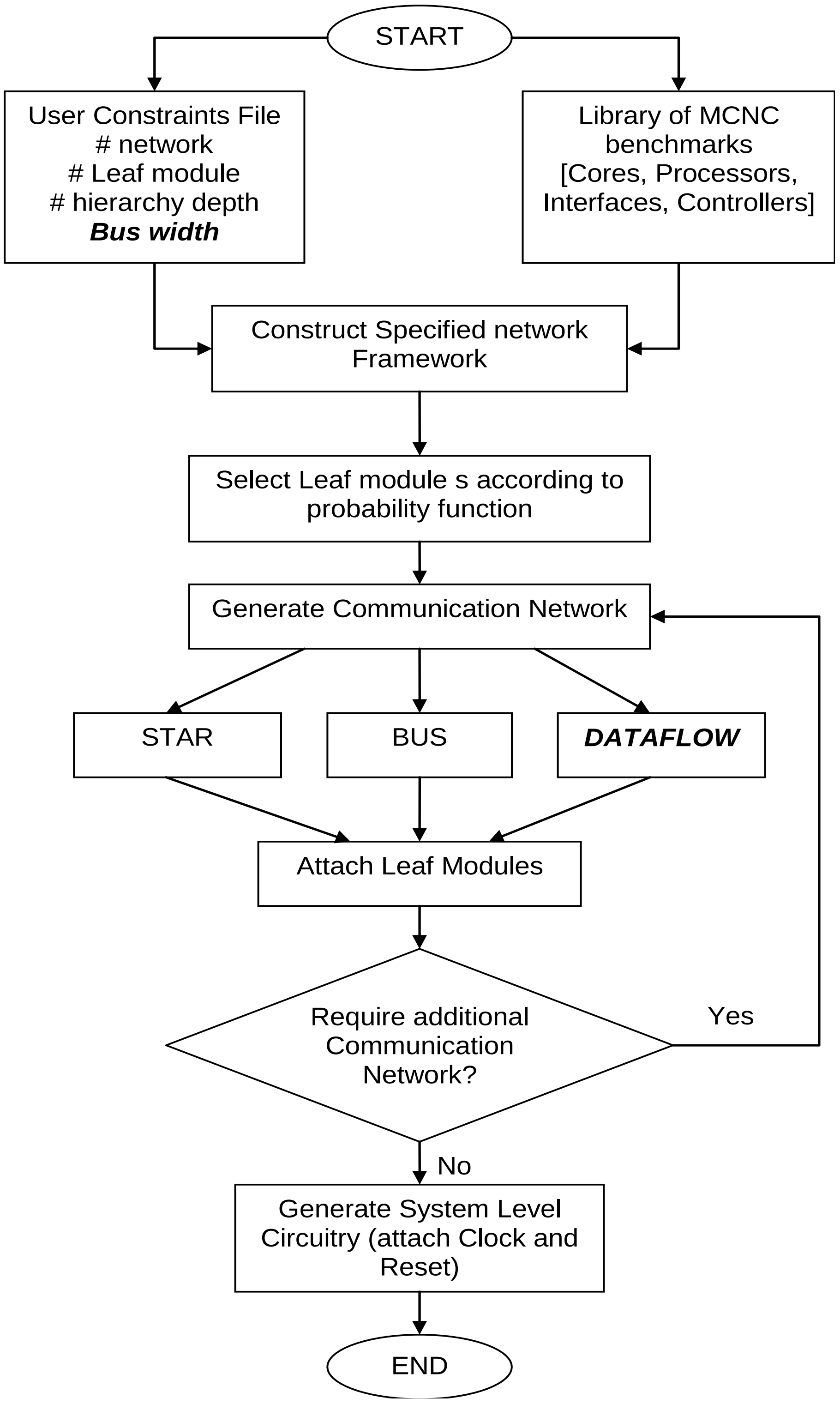
START
User Constraints File
# network
# Leaf module
# hierarchy depth
Bus width
Library of MCNC benchmarks [Cores, Processors, Interfaces, Controllers]
Construct Specified network Framework
Select Leaf module s according to probability function
Generate Communication Network
STAR
BUS
DATAFLOW
Attach Leaf Modules
Require additional Communication Network?
Yes
No
Generate System Level Circuitry (attach Clock and Reset)
END

Figure 4.1: Software organization flow for the BCGEN

Using these two sets of inputs, BCGEN constructs the user specified network framework by arranging different randomly selected modules from the MCNC library in the form of a tree. During this framework arrangement, BCGEN assigns the top hierarchy level to a network and the remaining networks are assigned to a random hierarchy level. Now, the algorithm selects each leaf module for the networks according to the probability function and generates different communication patterns depending on the user’s specifications. The user is able to choose any combination from the available star, bus and dataflow communication patterns in a single design. The star, bus and dataflow network patterns generated according to the original BCGEN algorithm [13] is detailed in Section 2.3.

After generating the appropriate communication network, the algorithm attaches different leaf modules to the specified network and checks whether additional communication patterns are required. If additional communication patterns are required, then the algorithm goes back to generate another communication pattern as shown in Figure 4.1. After all the required communication patterns are generated, the algorithm generates system level circuitry and attaches the clock and reset signals to the final generated circuit. The format of the final generated circuit is BLIF, which is used as an input to T-V pack.

4.2 Our Modifications

Our modifications to the original BCGEN logic flow are highlighted in Figure 4.1 using bold and italics font. Detailed descriptions of our modification to the dataflow network generation are described in Chapter 3. The other significant modification was to enable the dataflow communication pattern to support fixed bus widths. Based on the original method of stitching dataflow modules together, bus widths between modules were random. By using the multiplexers and demultiplexers to reduce or increase the number of I/Os, the bus width specified in the user constraint file is used as the width of the FIFO block. The different benchmark circuits generated using the dataflow communication pattern for our modified version and the original version of BCGEN are compared in the next Chapter.

5: CHAPTER 5

Testing and Discussion of Results

In this chapter, the details of our test parameters, results and the evaluation of our updates to the dataflow communication pattern of the original BCGEN are provided. To do this, we generated a set of circuits using both our updated version and the original version of BCGEN and compared the results of both circuit generation methods. The user constraints file and the MCNC library remain same for both versions. We have used T-VPACK and VPR5.0 [11] for mapping each circuit into a minimum-sized FPGA.

5.1 Methodology

In this section, the details of the test parameters used for verification of both modified and old BCGEN are described. We have used the same test parameters as the ones used for testing the original BCGEN. Hence, we can have precise comparison of the performance of both the BCGEN implementations. First of all, a clustered architecture is used, where each cluster (i.e. logic block) contains four 4-input LUTs and four FFs. Each cluster has 10 inputs and 4 outputs. The routing segments are uniform spanning four logic blocks in this architecture. T-VPACK is used to pack LUTs and FFs into the logic blocks. Generally, VPR recognizes the '.net' format so T-VPACK converts a netlist of blif format to the '.net' format. Throughout the testing process, clustering, placement and routing are timing-driven. For placement and routing, each circuit is first converted into a netlist (.net) format using T-VPACK. Finally, the circuit is placed and routed into an FPGA architecture using VPR tool.

5.2 Results

The results obtained for 12 circuits generated with the dataflow communication pattern using the modified BCGEN, as well as from the original BCGEN, are tabulated in Table 5.1. In Column 2, rows with ‘M’ (Modified version) denote our results and rows with ‘O’ (Original version) denote the results obtained for the original BCGEN. Column 3 and 4 in Table 5.1 show that the number of I/Os generated using our version of BCGEN in the final generated circuits are fewer than the number of I/Os generated using the original BCGEN. Figure 5.1 shows a graphical representation of I/Os, where I/P BCGEN Modified indicates inputs generated using our version, I/P BCGEN Original indicates inputs generated using BCGEN original dataflow pattern and O/P BCGEN Modified gives outputs generated using our versions whereas O/P BCGEN Original givers outputs generated using BCGEN Original dataflow pattern. This demonstrates that the addition of the multiplexers and demultiplexers in between the existing modules has the desired effect of reducing the number of I/Os.

Rent’s rule is given in Equation 1 below for the given sub-circuit, which is known as the relationship between the number of external I/Os and number of modules in the given sub-circuit.

$$\mathrm{A} = \kappa \mathrm{B}^{\rho} \tag{1}$$

Table 5.1: Results from our work and original BCGEN for the Dataflow network

Circuit	BCGEN Version	Input	Output	Rent Parameter	No. of Clusters	Avg. Net Length	Channel Width	Critical Path Delay
1	Mod.	47	11	0.466	984	8.82	18	41
	Orig.	47	39	0.656	93	6.82	13	18.7
2	Mod.	14	11	0.429	936	8.35	19	37.9
	Orig.	17	11	0.529	45	5.78	10	20
3	Mod.	31	15	0.461	961	8.44	19	44.3
	Orig.	31	40	0.581	74	5.55	11	19.6
4	Mod.	22	18	0.531	1949	8.97	21	52.7
	Orig.	44	45	0.574	159	7.81	16	36.5
5	Mod.	46	56	0.519	503	7.68	17	42
	Orig.	70	56	0.545	239	6.98	17	25.5
6	Mod.	29	20	0.521	1741	8.93	19	41.9
	Orig.	46	55	0.6	328	8.29	17	37
7	Mod.	71	116	0.701	9682	21.32	48	85.1
	Orig.	71	277	0.707	8742	21.96	45	86.8
8	Mod.	257	26	0.582	2708	14.48	32	62.2
	Orig.	257	238	0.651	1737	19.27	27	70
9	Mod.	76	138	0.698	8654	22.43	48	85.6
	Orig.	76	259	0.708	7565	25.57	50	102
10	Mod.	230	8	0.693	9722	16.39	44	225
	Orig.	230	418	0.705	7820	18.08	44	227
11	Mod.	261	306	0.725	13142	19.48	45	226
	Orig.	261	633	0.729	12791	20.47	45	226
12	Mod.	169	22	0.700	9079	19.50	48	82.8
	Orig.	169	231	0.708	7589	21.27	47	82.2

In Equation 1, *A* is the number of external pins in a given sub-circuit after partitioning and *B* is the number of modules in a given sub-circuit after partitioning where κ is known as the Rent's constant and ρ is known as the Rent parameter. The Rent parameter indicates the complexity of interconnects in a circuit. The Rent parameter depends on the circuit structure and selected partitioning algorithm. In the original BCGEN, the Rent parameter is calculated using a recursive Fiduccia-Mattheyses partitioning algorithm. We have used the same method to calculate the Rent parameter for the final circuits generated using the modified BCGEN. To reduce the congestion in the circuit layout, the Rent parameter should be lower. The results given in Table 5.1 and graphical representation of results drawn in Figure 5.2 clearly show that the Rent parameter generated from our circuits is less than the circuits generated using the original BCGEN because of reduced I/Os using the modified BCGEN. Lower Rent parameters reduce congestion in the layout.

Average post-routing net length affects the number of routing resources used by a net to connect to the logic blocks. Net length is defined as the number of consecutive clusters covered by a net. From the tabulated results (Table 5.1), the results obtained using the modified BCGEN have higher average net lengths because these circuits have greater internal connectivity due to added multiplexers, demultiplexers and FIFO blocks in between the original sub-modules during our circuit generation. Results generated for No. Of Clusters is graphically represented in Figure 5.3, where BCGEN original represents the result of number of clusters for original version and BCGEN modified represents the result of number of clusters for our modified version. To route a circuit, VPR uses only the required routing resources, resulting in an FPGA architecture with a minimum channel width. As mentioned above, higher average net length requires more routing resources, which tends to increase the minimum channel width. Therefore, our circuits result in mapping with larger channel widths. The maximum speed for a circuit is generally determined by the critical path delay, which is predominantly due to its routing. As such, the critical path delay for the modified BCGEN is more than the original implementation as they have longer internal nets (see Avg. Net Length, Table 5.1).

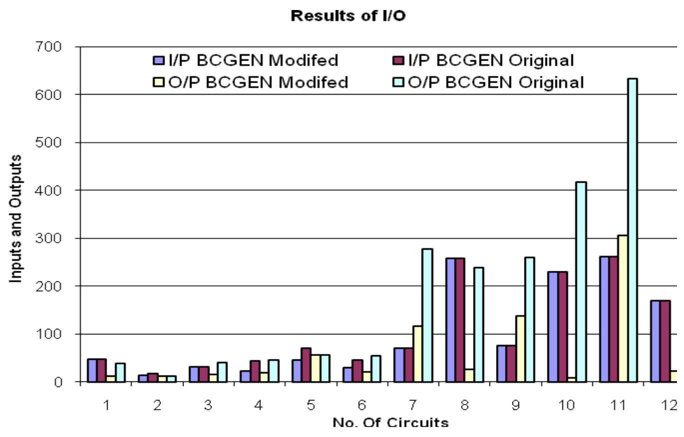

Figure 5.1 Results of I/O

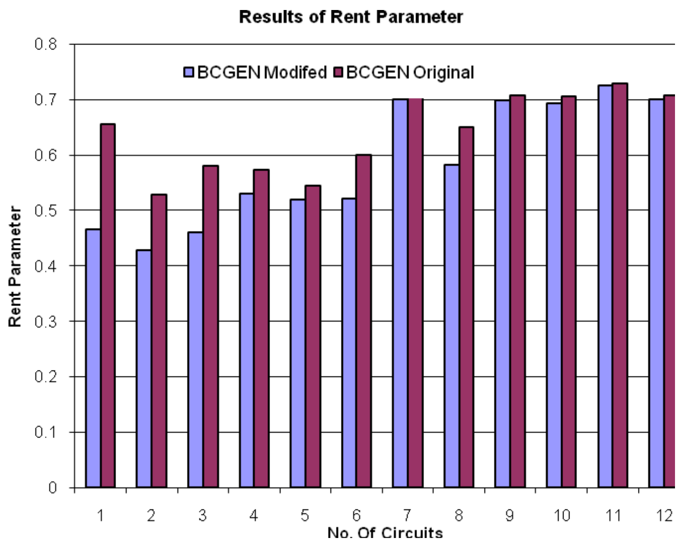

Figure 5.2 Results of Rent Parameters

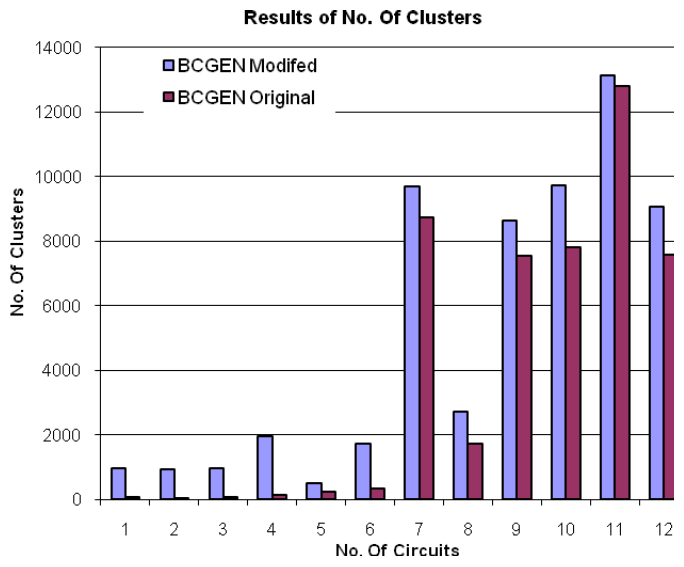

Figure 5.3 Results of No. Of Clusters

6: CHAPTER 6

Conclusions and Future Work

This chapter concludes our work and provides some suggestions for possible future work.

6.1 Conclusions

Our work improves the dataflow communication pattern of the original BCGEN to better model real circuits. This is achieved by inserting multiplexer and demultiplexer modules between the original modules (stages). Using the proposed approach, the number of I/Os in the generated circuit are reduced using the modified BCGEN. Additionally, the data buffering capability is also added by inserting the FIFO modules. As a result, the Rent parameter using the modified BCGEN is reduced by limiting the number of I/Os in the final generated circuit. This should allow circuit sizes to scale to better represent actual SoCs. Furthermore, the inclusion of buffering between stages is a key component in actual data-flow circuits again improving the quality of the synthetic data streaming benchmarks.

6.2 Future work

For future work, the star communication pattern should be updated to include real Network-on-Chip (NoC) architectures (mesh, torus etc.) as well as the ability to represent application specific topologies in the BCGEN. BCGEN should also include the ability to generate circuits comprising multiple clock domains. Finally, in the original BCGEN, bus networks support only single master architectures whereas, based on design trends, multi-master bus architectures would be useful.

REFERENCES

[1] Xilinx Inc, “Enabling Wireless-Solution around the world,” Online: http://www.xilinx.com/esp/wireless/index.htm

[2] M.Leeser, S. Loric, E. Miller, H. Yu and M. Trepanier, “Parallel-Beam Backprojection: An FPGA Implementation Optimized for Medical Imaging,” *Journal of VLSI Signal Processing*, 39(3):295-311, 2005.

[3] G. R. Goslin, “A guide to Using Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance,” *Xilinx Inc*, 1995.

[4] S. Sharma and W. Chen, “Model-Based Design to Accelerate FPGA Development for Automotive Applications,” *SAE International Journal of Passenger Cars*, 150-158, October 2009.

[5] R. Bianchil and A. H. Reali, “Implementing Computer-Vision Algorithms on Hardware: an FPGA/VHDL–based vision System for Mobile Robot,” *RoboCup 2001: Robot Soccer World Cup V, Springer-Verlag*, 281-286, 2002.

[6] D. McGrath, “FPGA Market to Pass $2.7 Billion by '10. In-Stat Says,” EE Times, May 24, 2006.

[7] M. Hutton, “The Circuit Characterization and Generation Project at the University Of Toronto,” http://www.eecg.toronto.edu/_mdhutton/gen/index.html.

[8] D. Stroobandt, J. Depreitre, and J. Van Campenhout, “Generating new benchmark designs using a multi-terminal net model,” *INTEGRATION: the VLSI Journal*, 27(2):113–129, 1999.

[9] C. Mark, A. Shui, S. Wilton, "A System-Level Stochastic Circuit Generator for FPGA Architecture Evaluation,*" in International Conference on Field-Programmable Technology*, December 2008.

[10] Open Cores FIFO Source Code: http://opencores.org/project,srl_fifo

[11] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W.M. Fang, and J. Rose, “VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling,” *Proceeding of the ACM/SIGDA international symposium on FPGAs*, pages 133-142, 2009.

[12] V. Betz, J. Rose, and A. Marquardt, “*Architecture and CAD for Deep-Submicron FPGAs,”* Norwell, MA: Kluwer, 1999.

[13] C. Mark, “A System-Level Synthetic Circuit Generator for FPGA Architectural Analysis Project at the University of British Columbia,” 2008. Online: http://www.ece.ubc.ca/~stevew/circuit/circuit_agreement.html

[14] M. Hutton, J. Rose, and D. Corneil, “Automatic generation of synthetic sequential benchmark circuits,” *IEEE Transactions on Computer-Aided Design*, 21(8):928–940, 2002.

[15] P. Kundarewich and J. Rose, “Synthetic circuit generation using clustering and iteration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp.869–887, June 2004.

[16] M. Tom and G. Lemieux, “Logic block clustering of large designs for channel-width constrained FPGAs,” *Proceedings of the 42nd annual Design Automation Conference*, 2005.

[17] D. Grant and G. Lemieux, “Perturb+mutate: Semi-synthetic circuit generation for incremental placement and routing,” *ACM Transactions on Reconfigurable Technology and System*s, 1(3): 1-24, 2009.

[18] J. Harlow and F. Brglez, “Synthesis of ESI equivalence class combinational circuit mutants,” *Technical Report 1997-TR@CBL-07-Harlow*, North Carolina State University, October 1997. Also available at http://www.cbl.ncsu.edu/publications.

[19] J. Pistorius, E. Legai, and M. Minoux, “Partgen: A generator of very large circuits to benchmark the partitioning of FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1314–1321, 2000.

APPENDIX

Our Modified Source code for BCGEN dataflow pattern

As discussed in Chapter 4, here is the source code we added/updated to implement our desired changes. Details are given in terms of the specific file listed below:

- GENERATE_CIRCUIT.C

This is an existing file from the original BCGEN, to which we have added code to update the dataflow pattern. When invoked, it adds multiplexers, demultiplexers and FIFOs between existing modules to reduce the number of I/Os and to include data buffering capabilities.

- MODIFIED_DATAFLOW.C

This is our additional code for the modified data flow pattern to reduce the number of I/Os and to add the data buffering capability by adding multiplexers, demultiplexers and FIFOs between existing modules.

- DATA_MUX.C

This code is used to generate multiplexers and demultiplexers to produce required number of outputs from the given number of inputs.

GENERATE_CIRCUIT.C

```
/* This is existing case in Original BCGEN where we have added call for our modified
method for BCGEN*/
case e_DATAFLOW:
//original code for DATAFLOW... here...
//retval = generate_dataflow_subcircuit(fpout, inst, structure, handle);
//clock test code..ADDED MODIFIED VERSION OF DATAFLOW
retval = generate_dataflow_subcircuit_clocktest(fpout, inst, structure, handle);
break;
```

MODIFIED_DATAFLOW.H

```
//header file for method description
circuit_t *generate_dataflow_subcircuit_clocktest(FILE *fpout, instance_t *inst,
structure_t *structure, char *handle);
```

MODIFIED_DATAFLOW.C

```
#include "../include.h"
//janki's code ..
circuit_node_t *generate_dataflow_multiplexer(FILE *fpout,int flag,char *handle,int
no_input,int no_output);
circuit_t *generate_dataflow_fifo(FILE *fpout,int flag,char *handle,int no_input,int
no_output);
void create_fifo(int in_no1,int out_no1,char *name);

//CODE FOR MODIFIED DATAFLOW COMMUNICATION PATTERN
circuit_t *generate_dataflow_subcircuit_clocktest(FILE *fpout, instance_t *inst,
structure_t *structure, char *handle)
{
//here for test flag =0 then multiplex signals(means input number in data_mux.c is greater
than output...
//and when flag =1 then demultiplex signals...mean input number is  smaller than
output....
//when flag =2 then add fifo..
```

```
        int flag;
        // variable from original code
        int buswidth;
        int temp;
        int clock;
        int fifo_flag;
        int i, j, k, m, n; // counters
        int node_index; // static
        int total_nodes; // constant
        char *name; // string holder
        char *copy_handle;
        int counter;//added for modified version for increamenting node value...
        int rem_opins, common, min; // local variables
        int *inter_level_empty_input_pins, *inter_level_empty_output_pins;
        BOOL finished_inputs, clock_pin, irupt_ctrl;
        int ffs[BUFFERLENGTH], num_ffs;
        structure_node_t *s_node; // pointers
        circuit_t *circuit,*fifo_circuit;
        circuit_node_t *node, *irupt_ctrl_node;
        circuit_node_t *fifo_circuit_node;
        circuit_edge_t *edge;
        int reset_source_index, reset_latch_node;
        int irupt_sink_index, irupt_latch_node, irupt_ctrl_node_index;
        circuit_node_t *node_1,*node_2,*node_3;
        structure_node_t *fifo_node; // added for modified version (for fifo)

        //added for FIFO (modified version)...
        int fifo_counter,*f_node;
        f_node = (int*) my_malloc(sizeof(int));
        fifo_counter=0;

        // index is allocated after order is generated
        fifo_node = (structure_node_t *) my_malloc (1 * sizeof (structure_node_t));
        fifo_node->circuit = NULL;
        fifo_node->numNodesI = 0;
        fifo_node->numNodesO = 0;
        fifo_node->nodesI = NULL;
        fifo_node->nodesO = NULL;
        fifo_node->rem_ipins = 0;
        fifo_node->rem_opins = 0;
        fifo_node->clock = FALSE;
        //end of node initialization for fifo...

        fifo_flag =0;
        num_ffs = 0;
        total_nodes = structure->num_blocks + structure->num_structures;
        node_index = total_nodes;
        s_node = structure->nodes;
        name = (char *) my_malloc(STRING*sizeof(char));
copy_handle =(char *) my_malloc(STRING*sizeof(char)); //mem allocation for copy_handle
        inter_level_empty_output_pins = (int *) my_malloc (structure->length * sizeof
        (int));
        inter_level_empty_input_pins = (int *) my_malloc (structure->length * sizeof
        (int));
        buswidth = inst->constraints.bus_width;
        counter =0;
        for (i = 0; i < structure->length; i ++)
        {
               inter_level_empty_output_pins[i] = 0;
               inter_level_empty_input_pins[i] = 0;
        }
        buswidth = inst->constraints.bus_width;
        printf("\n bus width =%d",buswidth);
        sprintf(name, "s_%d", structure->index);
        circuit = circuit_alloc_and_init(name, total_nodes, 0, 0, 0);
        reset_source_index = -1;
        reset_latch_node = -1;
        irupt_sink_index = -1;
        irupt_latch_node = -1;
        for (i = 0; i < total_nodes; i ++)
        {
```

```
		if (s_node[i].substructure == inst->reset_source)
		{
			reset_source_index = i;
		}
		if (s_node[i].substructure == inst->interrupt_sink)
		{
			irupt_sink_index = i;
		}
		if (structure->nodes[i].type == e_BLOCK)
		sprintf(name, "b_%d", ((block_t *)structure->nodes[i].substructure)-
		>index);
		else
		sprintf(name, "s_%d", ((structure_t *)structure->nodes[i].substructure)-
		>index);
node = circuit_node_alloc_and_init_from_subcircuit(structure->nodes[i].circuit, name, 0,
0, &i, &i);
		circuit_set_node(circuit, node, i);
	}

	// generate outputs from the previously indicated sources to the indicated sinks
	for (i = 0; i < total_nodes; i ++)
	{
		if (s_node[i].numNodesO == 0)
		{
			// if it is at the end of the sequence allocate the output nodes
			if (s_node[i].sequence == structure->length - 1)
			{
				for (k = 0; k < s_node[i].rem_opins; k ++)
				{
node_index = generate_node_add_external_o(circuit, node_index, i);
				}
				s_node[i].rem_opins = 0;
			}
			// else
		// this is a node with no outputs, leave until later to maybe fill in gaps
		}//end of if
		else if (s_node[i].numNodesO == 1)
		{
			counter++;
			sprintf(copy_handle,"%s_%d",handle,counter);
			printf("\n name of copy_handle =%s",copy_handle);
			if(s_node[i].rem_opins > buswidth) //demultiplex
			{
				flag =0;
			}
			else //multiplex..
			{
				flag =1;

			}
node_1=generate_dataflow_multiplexer(fpout,flag,copy_handle,s_node[i].rem_opins,buswidth);
		circuit_add_node(circuit,node_1);

			for (j = 0; j < s_node[i].rem_opins; j ++)
				{
				edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
				circuit_add_edge(circuit, edge);
				circuit_edge_set_source_node(edge, 0,i);
				circuit_edge_set_sink_node(edge, 0, node_index);
				circuit_node_add_out(circuit_get_node(circuit, i), edge-
				>net);
				circuit_node_add_fanin(circuit_get_node(circuit,
				node_index), edge->net);
				}
				node_index++;

			///add fifo here... between two node... code for test...
			counter++;
			sprintf(copy_handle,"%s_%d",handle,counter);
			flag =2;
			f_node[fifo_counter] = node_index;
```

```
                    fifo_counter =fifo_counter+1;
                    f_node = (int*) my_realloc(f_node,sizeof(int)*(fifo_counter+1));
                    //here node_index will be 3 ...
//store node_index value to some variable so that it can be use to add reset and clock...
later on..
fifo_circuit = generate_dataflow_fifo(fpout,flag,copy_handle,buswidth,buswidth);
//check for clock....
//if circuit does not have clock this function returns -1 otherwise it returns clock pin
number
                    clock = circuit_has_clock_pin(fifo_circuit);
                    if (clock != -1)
                    {
                    fifo_node->clock = TRUE;
                    circuit_swap_input_order(fifo_circuit, clock, fifo_circuit-
                    >num_inputs - 1);
                    }

                    //end of check for clock
                    // convert into a node
fifo_circuit_node = circuit_node_alloc_and_init_from_subcircuit(fifo_circuit, "FIFO", 0,
0, NULL, NULL);
                    circuit_add_node(circuit,fifo_circuit_node);
                    fifo_flag=1;
                    for (j = 0; j < buswidth; j ++)
                    {
                           edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                           //original
                           circuit_add_edge(circuit, edge);
                           circuit_edge_set_source_node(edge, 0, node_index-1);
                           circuit_edge_add_sink(edge, node_index);
                           circuit_node_add_out(circuit_get_node(circuit, node_index-
                           1), edge->net);
                           circuit_node_add_fanin(circuit_get_node(circuit,node_index),
                           edge->net);
                    }
                    for(j=0;j<2;j++) //which will add two additional input to fifo...
                    {
                    circuit_edge_add_sink(edge, node_index);
                    circuit_node_add_fanin(circuit_get_node(circuit,node_index), edge-
                    >net);
                    }
                    node_index++;
                    //end of fifo..
                    //here (nodeindex-1) = fifo node number ...
                    for(j=0;j<3;j++)//connect three output from fifo to cindy's block..
                    {
                           edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                           circuit_add_edge(circuit, edge);
                           circuit_edge_set_source_node(edge, 0, node_index-1);
                           circuit_edge_add_sink(edge, s_node[i].nodesO[0]->index);
                           circuit_node_add_out(circuit_get_node(circuit, node_index-
                           1), edge->net);
                           circuit_node_add_fanin(circuit_get_node(circuit,
                           s_node[i].nodesO[0]->index), edge->net);

                    }
                    s_node[i].nodesO[0]->rem_ipins =s_node[i].nodesO[0]->rem_ipins -3;
                    //because of connection from fifo to cindy's node..
                    if(buswidth >s_node[i].nodesO[0]->rem_ipins)
                    {
                           flag =0;
                    }
                    else
                    {
                           flag =1;
                    }
                    counter++;
                    sprintf(copy_handle,"%s_%d",handle,counter);
node_3 =generate_dataflow_multiplexer(fpout,flag,copy_handle,buswidth,s_node[i].nodesO[0]-
>rem_ipins);
```

```
                    circuit_add_node(circuit,node_3);
                    for (j = 0; j < buswidth; j ++)
                    {
                    edge = circuit_edge_alloc_and_init(1, 0, 1, NULL); //original
                    circuit_add_edge(circuit, edge);
                    circuit_edge_set_source_node(edge, 0, node_index-1);
                    circuit_edge_add_sink(edge, node_index);
                    circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                    >net);
                    circuit_node_add_fanin(circuit_get_node(circuit,node_index), edge-
                    >net);
                    }

                    for (j = 0; j < s_node[i].nodesO[0]->rem_ipins; j ++)
                    {
                    edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                    circuit_add_edge(circuit, edge);
                    circuit_edge_set_source_node(edge, 0, node_index);
                    circuit_edge_add_sink(edge, s_node[i].nodesO[0]->index);
                    circuit_node_add_out(circuit_get_node(circuit, node_index), edge-
                    >net);
                    circuit_node_add_fanin(circuit_get_node(circuit,
                    s_node[i].nodesO[0]->index), edge->net);
                    }
                    s_node[i].nodesO[0]->rem_ipins = 0;
                    s_node[i].rem_opins = 0;
                    node_index++;
            }//end of else if

            else// outputs to many blocks
            {
             for (j = 0; j < s_node[i].numNodesO; j ++)
                    {
                            if (s_node[i].rem_opins != 0)
                            {
                                    ounter++;
                                    printf(copy_handle,"%s_%d",handle,counter);
                                    printf("\n name of copy_handle =%s",copy_handle);
                                    if(s_node[i].rem_opins > buswidth) //demultiplex
                                    {
                                            flag =0;
                                    }
                                    else //multiplex..
                                    {
                                            flag =1;
                                    }
node_1 generate_dataflow_multiplexer(fpout,flag,copy_handle,s_node[i].rem_opins,buswidth);
circuit_add_node(circuit,node_1);

                                    for (k = 0; k < s_node[i].rem_opins; k ++)
                                    {
                                    edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                                    circuit_add_edge(circuit, edge);
                                    circuit_edge_set_source_node(edge, 0, i);
                                    circuit_edge_set_sink_node(edge, 0, node_index);
                                    circuit_node_add_out(circuit_get_node(circuit,
                                    s_node[i].index), edge->net);
                                    circuit_node_add_fanin(circuit_get_node(circuit,
                                    node_index), edge->net);
                                    }
                                    node_index++;

                    ///add fifo here... between two node... code for test...
                                    counter++;
                                    sprintf(copy_handle,"%s_%d",handle,counter);
                                    flag =2;
                                    f_node[fifo_counter] = node_index;
                                    fifo_counter =fifo_counter+1;
f_node = (int*) my_realloc(f_node,sizeof(int)*(fifo_counter+1));
fifo_circuit = generate_dataflow_fifo(fpout,flag,copy_handle,buswidth,buswidth);
                                    //check for clock....
```

```
//if circuit does not have clock this function returns -1 otherwise it returns clock pin
//number
                                     clock = circuit_has_clock_pin(fifo_circuit);
                                     if (clock != -1)
                                     {
                                     fifo_node->clock = TRUE;
                                     circuit_swap_input_order(fifo_circuit, clock,
                                     fifo_circuit->num_inputs - 1);
                                     }

                      //end of check for clock
                      // convert into a node
fifo_circuit_node = circuit_node_alloc_and_init_from_subcircuit(fifo_circuit, "FIFO", 0,
0, NULL, NULL);
                      circuit_add_node(circuit,fifo_circuit_node);
                      fifo_flag=1;
                      for (j = 0; j < buswidth; j ++)
                      {
                             edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                             circuit_add_edge(circuit, edge);
                             circuit_edge_set_source_node(edge, 0, node_index-1);
                             circuit_edge_add_sink(edge, node_index);
                             circuit_node_add_out(circuit_get_node(circuit, node_index-
                             1), edge->net);
                             circuit_node_add_fanin(circuit_get_node(circuit,node_index),
                             edge->net);
                      }
                      for(j=0;j<2;j++) //which will add two additional input to fifo...
                      {
                      circuit_edge_add_sink(edge, node_index);
                      circuit_node_add_fanin(circuit_get_node(circuit,node_index), edge-
                      >net);
                      }
                      node_index++;
                      //end of fifo..for second node...
                      //here (nodeindex-1) = fifo node number ...
                      for(j=0;j<3;j++)//connect three output from fifo to cindy's block..
                      {
                      edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                      circuit_add_edge(circuit, edge);
                      circuit_edge_set_source_node(edge, 0, node_index-1);
                      circuit_edge_add_sink(edge, s_node[i].nodesO[0]->index);
                      circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                      >net);
                      circuit_node_add_fanin(circuit_get_node(circuit,
                      s_node[i].nodesO[0]->index), edge->net);
                      }
s_node[i].nodesO[0]->rem_ipins =s_node[i].nodesO[0]->rem_ipins -3; //because of connection
//from fifo to cindy's node..
                      //for second node...

                      if(buswidth >s_node[i].nodesO[j]->rem_ipins)
                             flag =0;
                      }
                      else
                      {
                             lag =1;
                      }
node_2 =generate_dataflow_multiplexer(fpout,flag,copy_handle,buswidth,s_node[i].nodesO[j]-
>rem_ipins);
                      circuit_add_node(circuit,node_2);
                      for (k = 0; k < buswidth; k ++)
                      {
                      edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                      circuit_add_edge(circuit, edge);
                      circuit_edge_set_source_node(edge, 0, i);
                      circuit_edge_set_sink_node(edge, 0, node_index);
                      circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                      >net);
                      circuit_node_add_fanin(circuit_get_node(circuit, node_index), edge-
                      >net);
```

```
                        }
                for (k = 0; k < s_node[i].nodesO[j]->rem_ipins; k ++)
                {
                edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                circuit_add_edge(circuit, edge);
                circuit_edge_set_source_node(edge, 0, i);
                circuit_edge_set_sink_node(edge, 0, s_node[i].nodesO[j]->index);
                circuit_node_add_out(circuit_get_node(circuit, node_index), edge->net);
                circuit_node_add_fanin(circuit_get_node(circuit, s_node[i].nodesO[j]-
                >index), edge->net);
                }

                s_node[i].nodesO[j]->rem_ipins = 0;
                s_node[i].rem_opins = 0;
                node_index++;
                }//end of if.
                else
                {
                        s_node[i].numNodesO = j;
                        s_node[i].nodesO[j]->numNodesI --;
                }//end of else
                }//end of for..
                } //end of else
        }//end of for...

// intra_level assignments
for (i = 0; i < structure->length; i ++)
{
        for (j = structure->order[i]; j < structure->order[i + 1]; j ++)
        {
                        if (s_node[j].rem_opins != 0)
                        {
                        // the last nodes in the sequence should never reach here b/c opins
                        // should always be asserted
                for (m = structure->order[i + 1]; m < structure->order[i + 2]; m ++)
                // search for input pins
                        {
                                if (s_node[m].rem_ipins != 0)
                                        {
                                        counter++;
                                        sprintf(copy_handle,"%s_%d",handle,counter);
                                        printf("\n name of copy_handle =%s",copy_handle);

                                                if(s_node[j].rem_opins > buswidth)
                                                //demultiplex
                                                {
                                                        flag =0;
                                                }
                                                else //multiplex..
                                                {
                                                        flag =1;

                                                }

node_1=generate_dataflow_multiplexer(fpout,flag,copy_handle,s_node[j].rem_opins,buswidth);
                circuit_add_node(circuit,node_1);
                for (n = 0; n < s_node[j].rem_opins; n ++)
                        {
                        edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, j);
                        circuit_edge_set_sink_node(edge, 0, node_index);
                        circuit_node_add_out(circuit_get_node(circuit, s_node[j].index),
                        edge->net);
                        circuit_node_add_fanin(circuit_get_node(circuit, node_index), edge-
                        >net);
                        }
                        node_index++;
                        counter++;
                        sprintf(copy_handle,"%s_%d",handle,counter);
                        flag =2;
                        f_node[fifo_counter] = node_index;
                        fifo_counter =fifo_counter+1;
                        f_node = (int*) my_realloc(f_node,sizeof(int)*(fifo_counter+1));
        fifo_circuit = generate_dataflow_fifo(fpout,flag,copy_handle,buswidth,buswidth);
                        clock = circuit_has_clock_pin(fifo_circuit);
                        if (clock != -1)
                        {
                        fifo_node->clock = TRUE;
                        circuit_swap_input_order(fifo_circuit, clock, fifo_circuit-
                        >num_inputs - 1);
                        }

fifo_circuit_node = circuit_node_alloc_and_init_from_subcircuit(fifo_circuit, "FIFO", 0,
0, NULL, NULL);
                        circuit_add_node(circuit,fifo_circuit_node);
                        fifo_flag=1;
                        for (j = 0; j < buswidth; j ++)
                        {
                        edge = circuit_edge_alloc_and_init(1, 0, 1, NULL); //original
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, node_index-1);
                        circuit_edge_add_sink(edge, node_index);
                        circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                        >net);
                        circuit_node_add_fanin(circuit_get_node(circuit,node_index), edge-
                        >net);
                        }
                        for(j=0;j<2;j++)
                        {
                        circuit_edge_add_sink(edge, node_index);
                        circuit_node_add_fanin(circuit_get_node(circuit,node_index), edge-
                        >net);
                        }
                        node_index++;
                        for(j=0;j<3;j++)
                        {
                        edge = circuit_edge_alloc_and_init(1, 0, 1, NULL);
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, node_index-1);
                        circuit_edge_add_sink(edge, s_node[i].nodesO[0]->index);
                        circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                        >net);
                        circuit_node_add_fanin(circuit_get_node(circuit,
                        s_node[i].nodesO[0]->index), edge->net);
                        }
                        s_node[i].nodesO[0]->rem_ipins =s_node[i].nodesO[0]->rem_ipins -3;

                        //for second node...

                        if(buswidth >s_node[m].rem_ipins)
                        {
                                flag =0;
                        }
                        else
                        {
                        flag =1;
                        }
node_2=generate_dataflow_multiplexer(fpout,flag,copy_handle,buswidth,s_node[m].rem_ipins);
circuit_add_node(circuit,node_2);
                        for (n = 0; n < buswidth; n ++)
                        {
                        edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, i);
                        circuit_edge_set_sink_node(edge, 0, node_index);
                        circuit_node_add_out(circuit_get_node(circuit, node_index-1), edge-
                        >net);
                        circuit_node_add_fanin(circuit_get_node(circuit, node_index), edge-
                        >net);
                        }
                        for (n = 0; n < s_node[m].rem_ipins; n ++)
                        {
                        edge = circuit_edge_alloc_and_init(1, 1, 1, NULL);
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, i);
                        circuit_edge_set_sink_node(edge, 0, s_node[m].index);
                        circuit_node_add_out(circuit_get_node(circuit, node_index), edge-
                        >net);
                        circuit_node_add_fanin(circuit_get_node(circuit, s_node[m].index),
                        edge->net);
                        }
                        s_node[m].rem_ipins = 0;
                        s_node[j].rem_opins = 0;
                        node_index++;
                        }//end of if..
                        }// for input pins in the next level
                        break; // ie no more empty ipins in the next level to be allocated
                }
        }

}
        node_index = generate_structure_add_external_io(structure, circuit, node_index);
        // INTERRUPT
        irupt_ctrl = TRUE;
        if (structure->level != inst->constraints.hier_depth){
// HACK - THIS ASSUMES THERE IS ALWAYS AT LEAST 2 NODES ON EACH NETWORK
                if (irupt_sink_index == -1){
                        if (total_nodes == 1){
                                irupt_ctrl = FALSE;
                        }
                        else{
irupt_ctrl_node = generate_interrupt_controller_node(fpout, circuit, node_index,
total_nodes, handle);
                                circuit_add_node(circuit, irupt_ctrl_node);
                                irupt_ctrl_node_index = node_index;
                                node_index ++;
                                node_index = generate_node_add_external_o(circuit,
                                node_index, irupt_ctrl_node_index);
                        }
                }
                else{
                        i = total_nodes - 1 + (structure->level != 0); // -1 for sink, +1
                        //for the upper level
                        if (i > 1){
                        irupt_ctrl_node = generate_interrupt_controller_node(fpout,
                        circuit, node_index, i, handle);
                        circuit_add_node(circuit, irupt_ctrl_node);
                        irupt_ctrl_node_index = node_index;
                        node_index ++;

                                if (structure->level != inst->level_interrupt_sink){
                                        edge = circuit_edge_alloc_and_init(1, 1, 1,
                                        "irupt");
                                        circuit_add_edge(circuit, edge);
                                        generate_new_edge_connection(circuit, edge,
                                        irupt_ctrl_node_index, irupt_sink_index);
                                }
                                else{
                                        irupt_latch_node = node_index;
                                        node_index =
                                        generate_node_add_flipflop_to_output(circuit,
                                        node_index, irupt_ctrl_node_index);
                                        edge = circuit_edge_alloc_and_init(1, 1, 1,
                                        "irupt");
                                        circuit_add_edge(circuit, edge);
                                        generate_new_edge_connection(circuit, edge,
                                        irupt_latch_node, irupt_sink_index);
                                }
                                if (structure->level != 0){
                                        node_index = generate_node_add_external_i(circuit,
                                        node_index, irupt_ctrl_node_index);
                                }
                        }
                        else{ // only happens if there are 2 nodes and on the 0th level
                                irupt_ctrl = FALSE;
                                if(structure->level == inst->level_interrupt_sink){
                                        irupt_latch_node = node_index;
                                        node_index =
                                        generate_node_add_flipflop_to_input(circuit,
                                        node_index, irupt_sink_index);
                                }
                        }
                }
                for (i = 0; i < total_nodes; i ++){
                        if (i != irupt_sink_index){
                                sprintf(name, "b_%d_irupt", i);
                                edge = circuit_edge_alloc_and_init(1, 1, 1, name);
                                circuit_add_edge(circuit, edge);
                                if (!irupt_ctrl){
                                        if (irupt_latch_node == -1){
                                                if (irupt_sink_index == -1){
                                circuit_remove_edge(circuit, circuit->num_edges - 1);
                                node_index = generate_node_add_external_o(circuit,
                                node_index, i);
                                                }
                                                else{
                                        generate_new_edge_connection(circuit, edge, i,
                                        irupt_sink_index);
                                                }
                                        }
                                        else{
                                        generate_new_edge_connection(circuit, edge, i,
                                        irupt_latch_node);
                                        }
                                }
                                else
                                        generate_new_edge_connection(circuit, edge, i,
                                        irupt_ctrl_node_index);
                                }
                }
        }
        else{
                if (irupt_sink_index != -1){
                        irupt_latch_node = node_index;
                        node_index = generate_node_add_flipflop_to_input(circuit,
                        node_index, irupt_sink_index);
                        node_index = generate_node_add_external_i(circuit, node_index,
                        node_index - 1);
                }
        }

        // RESET
        if (reset_source_index == -1)//this cond.. does not satisfy second time..
        {
                node_index = generate_node_add_external_i(circuit, node_index, 0);
                edge = circuit_get_edge(circuit, circuit->num_edges - 1);
                for (i = 1; i < total_nodes; i ++){
                        circuit_edge_add_sink(edge, i);
                        circuit_node_add_fanin(circuit_get_node(circuit, i), edge->net);
                }
        }
        else{
                if (structure->level == 0){ // if the source is at the top level
                        reset_latch_node = node_index;
                        node_index = generate_node_add_flipflop_to_output(circuit,
                        node_index, reset_source_index);
                        edge = circuit_edge_alloc_and_init(1, 0, 1, "reset");
                        circuit_add_edge(circuit, edge);
                        circuit_edge_set_source_node(edge, 0, reset_latch_node);
                        circuit_node_add_out(circuit_get_node(circuit, reset_latch_node),
                        edge->net);
                }
                else{ // if not at the top level
```

```
if (structure->level == inst->level_reset_source)
 // add a latch at the source to prevent combinational cycles
                        {//this executes first time...
                        reset_latch_node = node_index;
                        node_index = generate_node_add_flipflop_to_output (circuit,
                        node_index, reset_source_index);
                        node_index = generate_node_add_external_o(circuit, node_index,
                        node_index - 1);
                        edge = circuit_get_edge(circuit, circuit_get_edge(circuit, circuit-
                        >nodes[reset_latch_node]->outs[0])->net);
                        }
                        else{ // add the output to the upper level
                        node_index = generate_node_add_external_o(circuit, node_index,
                        reset_source_index);
                        edge = circuit_get_edge(circuit, circuit->num_edges - 1);
                        }
                }
                for (i = 0; i < total_nodes; i ++) // add sinks to the rest of the nodes on
                //this level
                {
                        if (i != reset_source_index)
                        {
                        circuit_edge_add_sink(edge, i);
                        circuit_node_add_fanin(circuit_get_node(circuit, i), edge->net);
                        }
                }
                if(fifo_flag ==1)
                {
                        for(j=0;j<fifo_counter;j++)
                        {
                        circuit_edge_add_sink(edge, f_node[j]);
                        circuit_node_add_fanin(circuit_get_node(circuit, f_node[j]), edge-
                        >net);
                        }
                }


        }

        clock_pin = FALSE;

        for (i = 0; i < total_nodes; i ++)
        {
                if (s_node[i].clock == TRUE)
                {
                        if (!clock_pin)
                        {
                        node_index = generate_node_add_external_i(circuit, node_index, i);
                        edge = circuit->edges[circuit->num_edges - 1];
                        clock_pin = TRUE;
                        }
                        else
                        {
                        circuit_edge_add_sink(edge, i);
                        circuit_node_add_fanin(circuit_get_node(circuit, i), edge->net);
                        }
                }
        }

                if (fifo_node->clock == TRUE)
                {
                        for(j=0;j<fifo_counter;j++)
                        {
                        circuit_edge_add_sink(edge, f_node[j]);
                        circuit_node_add_fanin(circuit_get_node(circuit, f_node[j]), edge-
                        >net);
                        }
                }


        if (irupt_latch_node != -1){
                if (!clock_pin){
                node_index = generate_node_add_external_i(circuit, node_index,
                irupt_latch_node);
                clock_pin = TRUE;
                }
                else{
                edge = circuit_get_edge(circuit, circuit->num_edges - 1);
                circuit_edge_add_sink(edge, irupt_latch_node);
                circuit_node_add_fanin(circuit_get_node(circuit, irupt_latch_node), edge-
                >net);
                }
        }
        if (reset_latch_node != -1){
                if (!clock_pin){
                node_index = generate_node_add_external_i(circuit, node_index,
                reset_latch_node);
                clock_pin = TRUE;
                }
                else{
                edge = circuit_get_edge(circuit, circuit->num_edges - 1);
                circuit_edge_add_sink(edge, reset_latch_node);
                circuit_node_add_fanin(circuit_get_node(circuit, reset_latch_node), edge-
                >net);
                }
        }
        for (i = 0; i < num_ffs; i ++){
                if (!clock_pin){
                node_index = generate_node_add_external_i(circuit, node_index, ffs[i]);
                clock_pin = TRUE;
                }
                else{
                edge = circuit_get_edge(circuit, circuit->num_edges - 1);
                circuit_edge_add_sink(edge, ffs[i]);
                circuit_node_add_fanin(circuit_get_node(circuit, ffs[i]), edge->net);
                }
        }
        circuit_sanity(circuit, fpout, TRUE);
        free(f_node);

        free(name);
        free(inter_level_empty_output_pins);
        free(inter_level_empty_input_pins);
        return circuit;

}
//end of code to change...

circuit_node_t *generate_dataflow_multiplexer(FILE *fpout,int flag,char *handle,int
no_input,int no_output)
{
FILE *fp;
char *cmd_string, *filename;
circuit_t *mux_circ;
circuit_node_t *mux_node,*demux_node;
char *name;
int clock;
int count;
name =(char *) my_malloc(2*STRING*sizeof(char));
cmd_string = (char *) my_malloc(2*STRING*sizeof(char));
filename = (char *) my_malloc(2*STRING*sizeof(char));
count =0;
printf("\n number of input =%d",no_input);
printf("\n number of output =%d",no_output);
if(flag==0)
{
        printf(name,"mux_%s",handle);
        sprintf(filename, "multiplexer_%s", handle);
        create_mux( no_input,no_output,filename);
        #ifdef WIN32
        SetCurrentDirectory("C:\\Docume~1\\jjp11\\desktop\\research\\bcgen\\bcgen\\BUS_GEN
        ERATOR");
        sprintf(cmd_string, "perl module_gen_janki.pl --filename=%s --entityname=mux_%s --
        design=%s > %s.output 2>&1",filename, filename,filename,filename);
        system (cmd_string);
        SetCurrentDirectory("..");
        #endif
        sprintf(filename, "BUS_GENERATOR/multiplexer_%s.blif", handle);
        fp = fopen(filename, "r");
        if (fp == (FILE *) NULL) fprintf(fpout,"Error generating multiplexer block\n");
        else
        {

                mux_circ = read_blif(fp, fpout, filename, FALSE, TRUE);
                fclose(fp);
                if (mux_circ == (circuit_t *)NULL)
                {
                        fprintf(fpout,"Error reading mux block\n");
                }
        }
        // convert into a node
mux_node = circuit_node_alloc_and_init_from_subcircuit(mux_circ, name, 0, 0, NULL, NULL);
        }//end of if for flag =0

// flag =1 then demultiplex signals...mean input number is  smaller than output....
        if (flag ==1)
        {
                sprintf(name,"demux_%s",handle);
                sprintf(filename, "demultiplexer_%s", handle);
                create_mux(no_input,no_output,filename);
                #ifdef WIN32
        SetCurrentDirectory("C:\\Docume~1\\jjp11\\desktop\\research\\bcgen\\bcgen\\BUS_GEN
        ERATOR");
        sprintf(cmd_string, "perl module_gen_janki.pl --filename=%s --entityname=mux_%s --
        design=%s > %s.output 2>&1", filename,filename,filename,filename);

                system (cmd_string);
                SetCurrentDirectory("..");
                #endif
                sprintf(filename, "BUS_GENERATOR/demultiplexer_%s.blif", handle);
                fp = fopen(filename, "r");
                if (fp == (FILE *) NULL) fprintf(fpout,"Error generating demultiplexer
                block\n");
        else
        {

                mux_circ = read_blif(fp, fpout, filename, FALSE, TRUE);
                fclose(fp);
                if (mux_circ == (circuit_t *)NULL)
                {
                        fprintf(fpout,"Error reading demux block\n");
                }
        }
        // convert into a node
mux_node = circuit_node_alloc_and_init_from_subcircuit(mux_circ, name, 0, 0, NULL, NULL);

        }//end of if for flag =1

// flag =1 then demultiplex signals...mean input number is  smaller than output....

free(cmd_string);
free(filename);
return mux_node;

}//end of generate_dataflow_multiplexer function...


circuit_t *generate_dataflow_fifo(FILE *fpout,int flag,char *handle,int no_input,int
no_output)
{

FILE *fp;
char *cmd_string, *filename;
```

```
circuit_t *mux_circ;
char *name;
int clock;
int count;
name =(char *) my_malloc(2*STRING*sizeof(char));
cmd_string = (char *) my_malloc(2*STRING*sizeof(char));
filename = (char *) my_malloc(2*STRING*sizeof(char));
count =0;
printf("\n number of input =%d",no_input);
printf("\n number of output =%d",no_output);
//flag ==2 ..for fifo core...
        if (flag ==2)
        {
        sprintf(name,"fifo_%s",handle);
        sprintf(filename, "fifo_core_%s", handle);
        create_fifo(no_input,no_output,filename);
        #ifdef WIN32
        SetCurrentDirectory("C:\\Docume~1\\jjp11\\desktop\\research\\bcgen\\bcgen\\BUS_GEN
        ERATOR");
        sprintf(cmd_string, "perl module_gen_janki.pl --filename=%s --entityname=mux_%s --
        design=%s > %s.output 2>&1", filename,filename,filename,filename);
        system (cmd_string);
        SetCurrentDirectory("..");
        #endif
        sprintf(filename, "BUS_GENERATOR/fifo_core_%s.blif", handle);
        fp = fopen(filename, "r");
        if (fp == (FILE *) NULL)
        {
        fprintf(fpout,"Error generating fifo block\n");
        }
        else
        {
        mux_circ = read_blif(fp, fpout, filename, FALSE, TRUE);
        fclose(fp);
        if (mux_circ == (circuit_t *)NULL)
        {
                fprintf(fpout,"Error reading  fifo block\n");
        }
        }

        }//end of if for flag =2
free(cmd_string);
free(filename);
return mux_circ;
}//end of function
```

//CODE FOR SRL FIFO (VHDL code borrowed from OPEN CORE)..

```
void create_fifo(int in_no1, int out_no1,char *name)
{
        int in_no,out_no;
        char *name_copy;
        FILE *f;
        SetCurrentDirectory("C:\\Docume~1\\jjp11\\desktop\\research\\bcgen\\bcgen\\BUS_GEN
        ERATOR");
        name_copy = (char *) my_malloc(2*STRING*sizeof(char));
        printf("\n name of the file =%s",name);
        strcpy(name_copy,name);
        strcat(name_copy,".vhd");
        in_no =in_no1;
        out_no = out_no1;
        f= fopen(name_copy,"w");
        if(f==NULL)
        {
        printf("An error has occurred.\n");
        return 1;
        }
        //start of vhdl file printing........
        fprintf(f,"library IEEE;\n");
        fprintf(f,"use IEEE.STD_LOGIC_1164.ALL;\n");
        fprintf(f,"use IEEE.STD_LOGIC_ARITH.ALL;\n");
        fprintf(f,"use IEEE.STD_LOGIC_UNSIGNED.ALL;\n");
        fprintf(f,"use IEEE.NUMERIC_STD.ALL;\n");
        fprintf(f,"\n entity %s is \n",name);
        fprintf(f," \ngeneric(width : integer := %d); -- set to how wide fifo is to
        be",in_no);
        fprintf(f,"\nport( \n");
        fprintf(f,"\ndata_in      : in     std_logic_vector (width -1 downto 0);");
        fprintf(f,"\ndata_out     : out    std_logic_vector (width -1 downto 0);");
        fprintf(f,"\nreset        : in     std_logic;");
        fprintf(f,"\n write        : in     std_logic;");
        fprintf(f,"\n read         : in     std_logic;");
        fprintf(f,"\nfull         : out    std_logic;");
        fprintf(f,"\n half_full    : out    std_logic;");
        fprintf(f,"\ndata_present : out    std_logic;");
        fprintf(f,"\nclk          : in     std_logic");
        fprintf(f,"\n);");
        fprintf(f,"\n end %s ;",name);
        fprintf(f,"\n architecture behavioural of %s is ",name);
        fprintf(f,"\n constant srl_length  : integer := 32;    -- set to srl 'type' 16 or
        32 bit length");
        fprintf(f,"\n constant pointer_vec : integer := 5;    -- set to number of bits
        needed to store pointer = log2(srl_length)");
        fprintf(f,"\n type    srl_array   is array ( srl_length - 1  downto 0 ) of
        STD_LOGIC_VECTOR ( WIDTH - 1 downto 0 );");
        fprintf(f,"\n signal   fifo_store            : srl_array;");
        fprintf(f,"\nsignal  pointer            : integer range 0 to srl_length - 1;");
        fprintf(f,"\nsignal pointer_zero        : std_logic;");
        fprintf(f,"\nsignal pointer_full        : std_logic;");
        fprintf(f,"\nsignal valid_write         : std_logic;");
        fprintf(f,"\nsignal half_full_int       : std_logic_vector( pointer_vec - 1 downto
        0);");
        fprintf(f,"\nsignal empty               : std_logic := '1';");
        fprintf(f,"\nsignal valid_count         : std_logic ;");
        fprintf(f,"\nbegin");
        fprintf(f,"\n-- Valid write, high when valid to write data to the store.");
        fprintf(f,"\nvalid_write <= '1' when ( read = '1' and write = '1' )  ");
        fprintf(f,"\n or  ( write = '1' and pointer_full = '0' ) else '0';");
        fprintf(f,"\n-- data store SRL's");
        fprintf(f,"\ndata_srl :process( clk )");
        fprintf(f,"\nbegin");
        fprintf(f,"\nif rising_edge( clk ) then");
        fprintf(f,"\nif valid_write = '1' then");
        fprintf(f,"\nfifo_store <= fifo_store( fifo_store'left - 1 downto 0) & data_in;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend process;");
        fprintf(f,"\ndata_out <= fifo_store( pointer );");
        fprintf(f,"\nprocess( clk)");
        fprintf(f,"\nbegin");
        fprintf(f,"\nif rising_edge( clk ) then");
        fprintf(f,"\nif reset = '1' then");
        fprintf(f,"\nempty <= '1';");
        fprintf(f,"\nelsif empty = '1' and write = '1' then");
        fprintf(f,"\nempty <= '0';");
        fprintf(f,"\nelsif pointer_zero = '1' and read = '1' and write =   '0' then");
        fprintf(f,"\nempty <= '1';");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend process;");
        fprintf(f,"\n--       W      R      Action");
        fprintf(f,"\n--       0      0      pointer <= pointer");
        fprintf(f,"\n--       0      1      pointer <= pointer - 1 Read, but no write, so
        less data in counter");
        fprintf(f,"\n--       1      0      pointer <= pointer + 1 Write, but no read, so
        more data in fifo");
        fprintf(f,"\n--       1      1      pointer <= pointer            Read and write,
        so same number of words in fifo");
        fprintf(f,"\nvalid_count <= '1' when (");
        fprintf(f,"\n(write = '1' and read = '0' and pointer_full = '0' and empty = '0'
        )");
        fprintf(f,"\nor");
```

```
        fprintf(f,"\n(write = '0' and read = '1' and pointer_zero = '0' )");
        fprintf(f,"\n) else '0';");
        fprintf(f,"\nprocess( clk )");
        fprintf(f,"\nbegin");
        fprintf(f,"\nif rising_edge( clk ) then");
        fprintf(f,"\nif valid_count = '1' then");
        fprintf(f,"\nif write = '1' then");
        fprintf(f,"\npointer <= pointer + 1;");
        fprintf(f,"\nelse");
        fprintf(f,"\npointer <= pointer - 1;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend if;");
        fprintf(f,"\nend process;");
        fprintf(f,"\n-- Detect when pointer is zero and maximum");
        fprintf(f,"\npointer_zero <= '1' when pointer = 0 else '0';");
        fprintf(f,"\npointer_full <= '1' when pointer = srl_length - 1 else '0';");
        fprintf(f,"\n  -- assign internal signals to outputs");
        fprintf(f,"\nfull <= pointer_full;  ");
        fprintf(f,"\nhalf_full_int <= std_logic_vector(to_unsigned(pointer,
        pointer_vec));");
        fprintf(f,"\nhalf_full <= half_full_int(half_full_int'left);");
        fprintf(f,"\ndata_present <= not( empty );");
        fprintf(f,"\nend behavioural;");
        fclose(f);
        }
```

DATA_MUX.C

```
#include "../include.h"

//CODE TO CREATE MULTIPLEXER AND DEMULTIPLEXER

void create_mux(int in_no1, int out_no1,char *name);

void create_mux(int in_no1, int out_no1,char *name)
{
int in_no,out_no,div1_result,div2_result,div3_result,mod,diff,value,rem_line,diff1;
        int flag,check;
        int val,temp_val,no,M;
        int var,counter;
        int *input =(int*)malloc(sizeof(int));
        int *output=(int*)malloc(sizeof(int));
        int copy_input,copy_output;
        int i,j,count,temp_value;
        char *mux_name,*name_copy,*copy_mux_name;
        FILE *f,*f1;
        SetCurrentDirectory("C:\\Docume~1\\jjp11\\desktop\\research\\bcgen\\bcgen\\BUS_GEN
        ERATOR");
        name_copy = (char *) my_malloc(2*STRING*sizeof(char));
        mux_name = (char *) my_malloc(2*STRING*sizeof(char));
        copy_mux_name = (char *) my_malloc(2*STRING*sizeof(char));
        printf("\n name of the file =%s",name);
        strcpy(name_copy,name);
        sprintf(mux_name,"mux_%s",name_copy);
        strcat(name_copy,".vhd");
        strcpy(copy_mux_name,mux_name);
        strcat(copy_mux_name,".vhd");
        //code for 2 to 1 mux...
        in_no =in_no1;
        out_no = out_no1;
        *input =in_no;
        *output = out_no;
        //case 1 when where *input > *output...
        if(*input>*output)
        {
                //start of mux.vhd...
                f1=fopen(copy_mux_name,"w");
                if(f1==NULL)
                {
                        printf("An error has occurred.\n");
                        return 1;
                }
                fprintf(f1,"library IEEE;\n");
                fprintf(f1,"use IEEE.STD_LOGIC_1164.ALL;\n");
                fprintf(f1,"use IEEE.STD_LOGIC_ARITH.ALL;\n");
                fprintf(f1,"use IEEE.STD_LOGIC_UNSIGNED.ALL;\n");
                fprintf(f1,"\n entity %s is \n",mux_name);
                fprintf(f1,"port\n(\n");
                fprintf(f1,"X : in std_logic;\n");
                fprintf(f1,"Y :in std_logic;\nS : in std_logic;\n ");
                fprintf(f1,"Z : out std_logic \n);\nend %s;\n ",mux_name);
                fprintf(f1,"architecture Behavioral of %s is \n ",mux_name);
                fprintf(f1,"begin \n  ");
                fprintf(f1," Z <= X when S = '0' else Y; \n ");
                fprintf(f1,"end Behavioral;\n ");
                fclose(f1);
                //end of mux.vhd file....
//now create another file for reducing the signal by multiplexing...by connecting to
//mux.vhd using port map....
                f= fopen(name_copy,"w");
                if(f==NULL)
                {
                        printf("An error has occurred.\n");
                        return 1;
                }
                //start of vhdl file printing........
                fprintf(f,"library IEEE;\n");
                fprintf(f,"use IEEE.STD_LOGIC_1164.ALL;\n");
                fprintf(f,"use IEEE.STD_LOGIC_ARITH.ALL;\n");
                fprintf(f,"use IEEE.STD_LOGIC_UNSIGNED.ALL;\n");
                fprintf(f,"\n entity %s is \n",name);
                //end of file printing
                printf("welcome to program...");
        //----------------PRINTING TO FILENAME.VHD FILE-----------------------------------
        fprintf(f, "generic \n( \n M  : integer := %d ;\n N  : integer := %d\n );", in_no ,
        out_no);
        fprintf(f,"\nport(\nA : in std_logic_vector(0 to %d); \nB : out std_logic_vector(
        0 to %d)\n);",in_no-1,out_no-1);
                fprintf(f,"\n end %s;",name);
                fprintf(f,"\narchitecture Behavioral of %s is",name);
                fprintf(f,"\ncomponent %s",mux_name);
                fprintf(f,"\n port(\nX : in std_logic;\nY :in std_logic;\nS : in
                std_logic;\nZ : out std_logic\n);");
                fprintf(f,"\nend component;");
                no=2;
                count =1;
                M=in_no%2;
                for(i=in_no/2;i>=1;i=i/2)
                {
                        if(M!=0)
                        {
                                out_no = out_no-1;
                        }
                        if(out_no >=i)
                        {
                                break;
                        }
                        count=count+1;
                        M=i%2;
                }//end of for...
                printf("\n count =%d",count);
                printf("\n out no =%d",out_no);
                temp_val=1;
                for(i=1;i<=count;i++)
                {//generate internal signals
                fprintf(f,"\nsignal X_%d  : std_logic_vector(0 to %d);",i,(in_no/temp_val-
                1));
                        temp_val=temp_val*2;
                }
                fprintf(f,"\nbegin");
                fprintf(f,"\n J :for i in 0 to %d generate",in_no-1);
                fprintf(f, "\nX_1(i) <= A(i);");
                fprintf(f,"\nend generate J;");
                val =2;
                for(i=1; i<=count;i++)
                {
                        M= in_no%2;
                        if(i>1)
                        {
//gives in_no input to multiplexer...
fprintf(f,"\nG%d: for i in 0 to %d generate",i-1,in_no-1);
fprintf(f,"\nU%d : %s port map(X =>X_%d(i*2) ,Y => X_%d((i*2)+1), S =>X_%d(i*2), Z =>
X_%d(i));",i-1,mux_name,i-1,i-1,i-1,i);
                        fprintf(f,"\nend generate G%d;",i-1);
                        }
                        if(M!=0)
                        {
                                fprintf(f,"\nB(%d) <= X_%d(%d);",*output-1,i,in_no-1);
                                *output = *output-1;
                        }
                        in_no = in_no/2;
                        val =val*2;
                }//end of for loop...

                diff = (in_no*2) - out_no;
                value =diff *2;//number of input goes to multiplexer...
                rem_line = (in_no*2) -value;//out_no = rem_line +diff.
                fprintf(f,"\n G: for i in 0 to %d generate",diff-1);
                fprintf(f,"\n U : %s port map(X =>X_%d(i*2) ,Y => X_%d((i*2)+1), S
                =>X_%d(i*2),Z => B(i));",mux_name,count,count,count);
                fprintf(f,"\nend generate G ;");
                j=value;
                for(i = diff ;i <out_no;i++)
                {

                        fprintf(f, " \n B(%d) <= X_%d(%d);",i,count,j);
                        j++;
                }
                fprintf(f,"\n end Behavioral;");
                fclose(f); //CLOSE THE FILE
        }//end of case 1..*input > *output..

        //case 2.. where *input1 < * input2 THEN demultiplex signals....
        if(*input<=*output)
        {
                diff1= *output-*input;
                flag =1;
                check=1;
                counter =0;
                var =0;
                f= fopen(name_copy,"w");
                if(f==NULL)
                {
                        printf("An error has occurred.\n");
                        return 1;
                }
                //start of vhdl file printing........
                fprintf(f,"library IEEE;\n");
                fprintf(f,"use IEEE.STD_LOGIC_1164.ALL;\n");
                fprintf(f,"use IEEE.STD_LOGIC_ARITH.ALL;\n");
                fprintf(f,"use IEEE.STD_LOGIC_UNSIGNED.ALL;\n");
                fprintf(f,"\n entity %s is \n",name);
                //means I need to increase wires upto diff1....from the *input..
                //---------------------PRINTING TO FILENAME.VHD FILE--------------------
                fprintf(f, "generic \n(  \n M : integer := %d ;\n N  : integer := %d\n );",
                in_no , out_no);
                fprintf(f,"\nport(\nA : in std_logic_vector(0 to %d);\n B : out
                std_logic_vector( 0 to %d)\n);",in_no-1,out_no-1);
                fprintf(f,"\n end %s;",name);
                fprintf(f,"\narchitecture Behavioral of %s is\n",name);
                fprintf(f,"begin\n ");
        //if diff1 is zero means both are same values ...so just connect from A to B...
```

```
                if(diff1 == 0)
                {
                        var= var+1;
                        fprintf(f,"\n B(0) <= A(0) xor A(1);\n");
                        fprintf(f,"K%d:for i in 1 to %d generate \n",var,in_no-1);
                        fprintf(f,"B(i) <= A(i);\n ");
                        fprintf(f," end generate;\n");
                }

//else means diffference is greater than zero so we have to make copy of //wires...which
//can connect to B
                else
                {
                        fprintf(f,"\n B(0) <= A(0) xor A(1);\n");
                        do
                        {
                                var = var +1;
                                if(check ==1)
                                {
                                        fprintf(f,"K%d:for i in 1 to %d generate
                                        \n",var,in_no-1);
                                }
                                else
                                {
                                        fprintf(f,"K%d:for i in 0 to %d generate
                                        \n",var,in_no-1);
                                }
                                check =0;
                                fprintf(f,"B(i+%d) <= A(i);\n",counter);
                                fprintf(f," end generate;\n");
                                if(flag ==0)
                                {
                                        diff1=diff1-in_no;
                                }
                                flag=0;
                                counter=counter+(in_no);
                        }while(diff1 > in_no);

                        if(diff1<0)
                        {
                                diff1 = (-diff1);
                        }
                        var = var +1;
                        fprintf(f,"K%d:for i in 0 to %d generate \n",var,(diff1-1));
                        fprintf(f,"B(i+%d) <= A(i);\n",counter);
                        fprintf(f," end generate;\n");
                }//end of else
        fprintf(f,"end behavioral;\n");
        fclose(f);
        }//end of if... case 2.....

}//end of function create_mux...
```