HIGH LEVEL SYNTHESIS OF KAHN PROCESS NETWORKS (KPN) FOR STREAMING APPLICATIONS

By

Attiya Mahmood

Submitted to the department of Computer Engineering in fulfillment of the requirements for the degree of

Masters in Science

In

Computer Engineering

Thesis Supervisor

Dr. Shoab A. Khan

College of Electrical and Mechanical Engg,
National University of Sciences and Technology
2009

ACKNOWLEDGEMENT

All praises and thanks to **ALMIGHTY ALLAH,** Who made this difficult work possible. This research work is the product of His blessings. I would like to express my heartfelt gratitude with love to my family especially to my parents for being so supportive and caring throughout the course of this study, my brothers and my sister without whom I would never be able to achieve this milestone.

I extend my appreciation and thanks to my supervisor, Dr. Shoab Ahmed Khan, Professor, Department of Computer Engineering, EME NUST, for his long lasting guidance, positive criticism and kind cooperation offered to me during my MS tenure. I feel I may never be able to pay him the deserved thanks. I wish to express my heartiest thanks to Mr. Ali Abbas, PhD Student, College of Electrical and Mechanical Engg, NUST Rawalpindi, for his kind cooperation and constructive suggestions at every step of my research.

The constant advice and encouragement of my friends and the work of early scientists which blazed this study is also highly acknowledged.

Last but not the least, I appreciate all those who remember me in their prayers and encourage me throughout the deals of my life.

ATTIYA MAHMOOD

Dedicated to …..

My mother, brothers and sister

who always encourage me to go for another degree!

DECLARATION

I hereby declare that no portion of the work referred to in this research work has been submitted or published in support of an application for another degree or qualification of this of any other university or other institute of learning. If any act of plagiarism found, I am fully responsible for every disciplinary action taken against me.

ABSTRACT

Streaming Applications usually run in parallel or in series that incrementally transform a stream of raw input data into a stream of processed output data. The only issue in streaming application is project realization time. This issue poses a design challenge to break such an application into distinguishable blocks and then to map them into independent hardware processing elements. For this, there is required a generic controller that automatically maps such a stream of data into independent processing elements without any dependencies and manual considerations. In my thesis work, we have designed and developed a framework of Kahn Process Networks (KPN) for such streaming applications that will be mapped on MPSoC. This is designed in such a way that we have a generic C-based compiler that will take the mapping specifications as an input from the user and then it will automate these design constraints and automatically generate the synthesized RTL optimized code for specified application.

TABLE OF CONTENTS

ACKNOWLEDGEMENT I

DEDICATION II

DECLARATION III

ABSTRACT IV

TABLE OF CONTENTS VII

LIST OF FIGURES VII

CHAPTER 1 INTRODUCTION 1

1.1 RELATED WORK 2

CHAPTER 2 LITERATURE REVIEW 4

SECTION I 5

2.1 MODELING DSP DESIGNS 5

2.1.1 LANGUAGE DRIVEN DESCRIPTION 5

2.2 FLOW GRAPHS 6

2.2.1 BLOCK DIAGRAM 7
2.2.2 SIGNAL FLOW GRAPHS (SFG) 7
2.2.3 DATA FLOW GRAPHS (DFG) 8
2.2.3.1 Synchronous Data Flow Graphs (SDFG) 10
2.2.3.2 Single Rate SDFG and Multi-Rate SDFG 11
2.2.3.3 Homogeneous Synchronous Data Flow Graphs (HSDF) 12
2.2.4 PARAMETERIZED SYNCHRONOUS DATA FLOW GRAPHS (PSDF) 13
2.2.5 CYCLO-STATIC SYNCHRONOUS DATA FLOW GRAPHS (CSDF) 14
2.2.6 PARAMETERIZED CYCLO-STATIC SYNCHRONOUS DATA FLOW GRAPHS (PCSDF) 14
2.2.7 DYNAMIC DATA FLOW GRAPHS (DDF) .... **ERROR! BOOKMARK NOT DEFINED.**

SECTION II 16

2.4 KAHN PROCESS NETWORK: 19

2.4.1 KAHN PROCESS NETWORK MODEL 19
2.4.2 KPN FOR MODELING STREAMING APPLICATIONS 21
2.4.3 RESTRICTIONS OF KPN MODEL: 22

2.4.4 CUSTOMIZED KPN MODEL: 23
2.4.5 EFFICIENT KPN SCHEDULING 25

CHAPTER 3 SYSTEM DESIGN 27

3.1 SYSTEM MODEL: 27

3.2 THESIS ORGANIZATION: 29

3.3 ALGORITHM: 31

CHAPTER 4 EXPERIMENTAL RESULTS 34

EXAMPLE 1: 34

DESCRIPTION: 34
CONFIGURATION FILE SPECIFICATIONS 34
COMPILER OUTPUT 36
CODE GENERATION TOOL SNAPSHOT 36
VERILOG SOURCE CODE OF GIVEN DSP DESIGN GENERATED BY C-COMPILER: 37

EXAMPLE 2 60

DESCRIPTION: 60
CONFIGURATION FILE SPECIFICATIONS 60
COMPILER OUTPUT 62
CODE GENERATION TOOL SNAPSHOT 62
VERILOG SOURCE CODE OF GIVEN DSP DESIGN GENERATED BY C-COMPILER: 63

CHAPTER 5 CONCLUSION AND FUTURE WORK 95

APPENDIX A SOURCE CODE OF C-BASED COMPILER 95

C-CODE OF GIVEN DSP DESIGN: 96

REFERENCES 137

LIST OF FIGURES

FIGURE 2-1: GENERAL ORGANIZATION OF DSP ALGORITHM REPRESENTATION 6
FIGURE 2-2: BLOCK DIAGRAM 7
FIGURE 2-3: SIGNAL FLOW GRAPHS 8
FIGURE 2-4: BASIC DFG MODEL 9
FIGURE 2-5: BASIC DFG MODEL SHOWING NODES GENERAL CHARACTERISTICS 11
FIGURE 2-6: PSDF MODEL 13
FIGURE 2-7: PSDF EXAMPLE 13
FIGURE 2-8: GENERALIZED PSDF MODEL 14
FIGURE 2-9: GENERALIZED PCSDF MODEL 14
FIGURE 2-10: GENERAL KPN MODEL 21
FIGURE 2-11: KPN MODELING FOR STREAMING APPLICATIONS 22
FIGURE 2-12: GENERAL MPSOC MODEL 24
FIGURE 3-1: SYSTEM MODEL 29
FIGURE 3-2: THESIS ORGANIZATION 30

CHAPTER 1

INTRODUCTION

Streaming applications are usually represented as a set of simultaneous processes that take a stream of input data and then transforms them into processed output stream of data. Implementing such an application on hardware poses a large challenging modeling problem. For this, KPN is the best ever representation to model such applications on hardware. KPN is a set of independent processes that communicate through point-to-point fashion over unbounded buffers with blocking Read and Non-blocking Write. This provides a very simple mechanism to map an application on hardware or software as KPN. The Reads and Writes also elevate the design from the use of complicated schedules. By this, streaming applications are mapped on independent processes working autonomously after acquiring sufficient data samples on its input buffers. Such data samples are called tokens in KPN’s terminology and such an execution is called firing of tokens.

Streaming applications are usually mapped on FPGA and ASIC. The main idea behind this work is to propose a system in which streaming applications will be broken down into set of separate independent processing elements (these processing elements will be set of FPGAs or ASICs), mapped through KPN and inter-process communication between these processing elements will be performed through NOC Switch. This paper demonstrates the mapping constraints on these processing elements and then finally generates a generic customized controller that automatically maps these applications on hardware. By this, the designer can simply add any number of available processors in streaming applications and automatically map different types of streaming applications on hardware without any manual settings and dependencies.

*1.1 **<u>RELATED WORK</u>***

No C-based compiler has yet been introduced in research that can automatically generate RTL synthesized KPN model based on the specifications of streaming application. But there are so many design issues in KPN implementation. A lot of research has been undergone in KPN buffer sizing, artificial deadlock detection and real time scheduler for KPN.

In 1974, Kahn proposed semantics of simple language for parallel programming. This was his PhD. thesis work in which he proposed a parallel computation model where any application can be modeled into set of concurrent independent processes with unbounded FIFOs (First In First Out) buffers at its inputs and outputs. Theses independent concurrent processing elements can be executed on any parallel processing units without incurring any overhead and dependencies. His main contribution of work is illustrated in [1]. [2] introduced some new features of KPN with regards to its task level parallelism and its deterministic behavior. In real sense, no memory allocation can be unbounded, so a lot of research has been made in optimum buffer sizing for KPN implementation. [3] proposed an automatic buffer sizing for KPN on MpSoC. They proposed an idea of automatic buffer sizing by starting with some fixed sizing and incrementally increase buffer sizing wherever needed. In KPN, FIFO read is blocking and FIFO write is non-blocking based on the assumption of unbounded buffer size. When we start imposing the impact of finite memory sizes then an artificial deadlock issue arises. [4] demonstrated the effectiveness of KPN in media and signal processing applications and presented the method of effective and bounded execution of KPN. [5,6] deals with this artificial deadlock detection when all the processes in the process network are blocked then they claim of finding the effective solution.[7] suggested a new idea of an early detection of artificial deadlocks in the process network of eclipse shape. In recent so many years, KPN has been modified in the set of different DSP designs because it is compositional and it allows parallelism. The output of the KPN is independent of the flow of sequence of execution. [8] presented the idea of designing and analysis of DSP designs using Kahn process networks. [9] proposed the idea of basic transformation of basic DSP designs into Kahn networks, but he did not focus on the task level decomposition of the particular DSP design and also the automatic controller for it. [10] diminished the concept of artificial deadlock in process networks and proposed a design of real time scheduler for process networks on multiprocessor system on chip. Because of KPN’s effectiveness, it is consistently used for mapping streaming (Audio or Video) applications on MPSoC. Compaan and Laura in [11] projected a system design where they take an application written in Matlab and automatically give the transformation which can be mapped on to target platform. YAPI in [12] provided a C++ interface that gives KPN implementation on single processor. [13] offered the idea of KPN exploration on multiprocessor system on chip.

In this thesis work, my field of interest lies in streaming application mapping. For that, I have proposed an architecture in which I have designed a C-based compiler that can automate the design constraints and automatically generates the synthesized HDL implementation of KPN that is application independent.

CHAPTER 2

LITERATURE REVIEW

Now days, there is continuous trend of technology advancement in the field of embedded multiprocessor on chip (MPSOC). For the real-time streaming application, it is always desirable to cater the delay and jitter in transmission. The embedded system must be very efficient in terms of its speed and its area consumption. To ensure its high speed, several architectures were proposed in which multiprocessor-on-a-chip is one of them. MPSOC ensures its fast processing with the help of multiprocessors core on a singe chip. Any application is broken down into set of sub-units and each sub-unit is actually mapped on these individual independent processors. The inter-processor communication is guaranteed in MPSOC by the help of NOC (Network-on-chip) switch. This is how MPSOC works. My thesis work is to figure out the efficient ways of automatically mapping any application on MPSOC. Implementing a digital signal processing (DSP) applications on the MPSOC is a complicated problem. My literature starts with how many ways of representing DSP designs graphically. Data Flow Graphs (DFG) is the most commonly used graphs for representing DSP designs. Then, Kahn Process Networks (KPN) is considered an excellent model for modeling any applications on MPSOC, because of its so many implementation easiness like task-level parallelism, explicit behavior and determinism.

This chapter is organized as follows. Section I lists the several ways of graphically representing DSP designs. Section II focuses on the detailed aspects of KPN and also its modifications with respect to DSP constraints.

SECTION I
DSP DESIGN REPRESENTATIONS

2.1 MODELING DSP DESIGNS

There are commonly two methods that are used for representing DSP designs. Flow graphs are totally different in the sense that they particularly focus on the graphical view of DSP designs.

- Language Driven Description
- Graphics/ Flow Graphs Specifications

2.1.1 LANGUAGE DRIVEN DESCRIPTION

Language driven specification relies on the languages that represent DSP designs. They are used for software development. Usually languages are interpretive or executable. Interpretive language is a MATLAB that provides the flexibility to designer but usually does not produce an optimized version of the design. There are so many techniques to make the MATLAB code an efficient one like loops reduction and so on and on. Executable languages are critical in designing but they promote the efficient representation of design because at here, designer actually works at the memory usage and actively deals with register, so fast and efficient design is possible in executable languages as compared to interpretive driven MATLAB.

Figure 2-1 shows the block diagram of general organization of representing DSP algorithms. This diagram demonstrates that generally, the DSP designs can be represented by two ways. It can be represented by an executable language description or by using flow graphs specifications. Next, the diagram shows that the language can be an executable or an interpretive one. Also it sub-categorizes the flow graphs in to so many ways that are application dependent. As flow graphs can be a block diagram, signal flow graphs (Representation in terms of signals information), the data flow graphs (representing graphs by data driven strategy), the Parameterized data flow graphs (parameterize some variables for history information of node), Cyclo-static graphs (that ultimately maps in to a periodic fashion). Also data flow graphs can more be cauterized in terms of

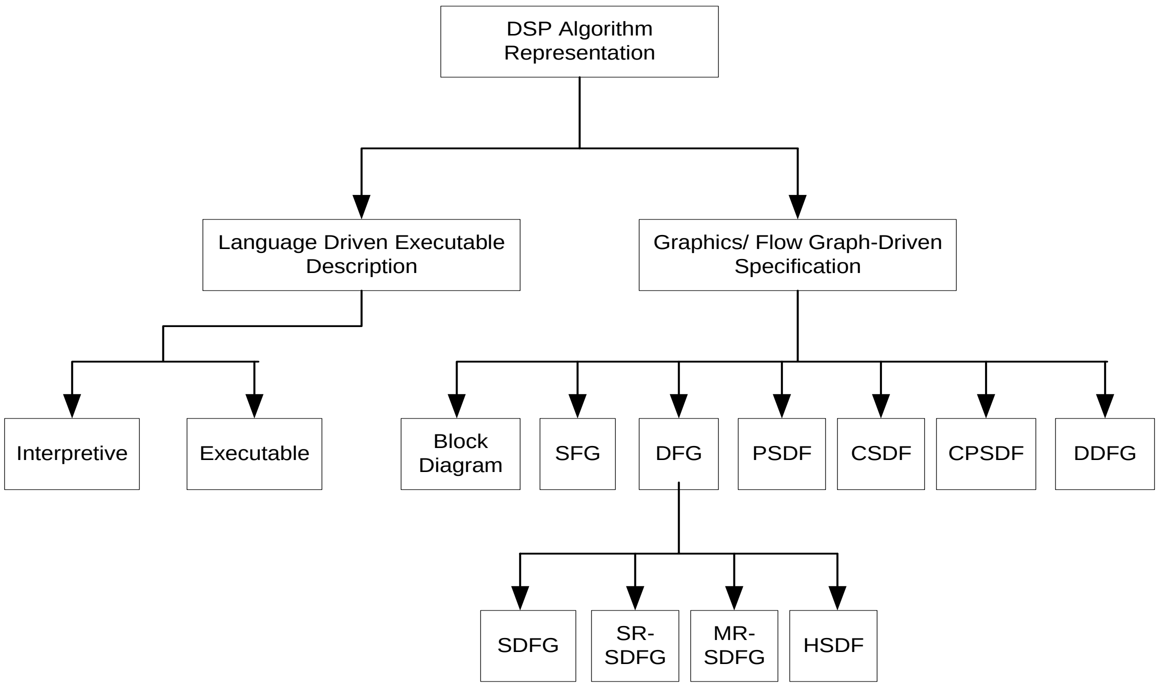

DSP Algorithm Representation
Language Driven Executable Description
Graphics/ Flow Graph-Driven Specification
Interpretive
Executable
Block Diagram
SFG
DFG
PSDF
CSDF
CPSDF
DDFG
SDFG
SR-SDFG
MR-SDFG
HSDF

Figure 2-1: General Organization of DSP Algorithm Representation

2.2 <u>FLOW GRAPHS</u>

As language driven specification is a way of representing DSP designs but it does not usually ensure visibility of design units. Flow graphs are always considered the better way of representing any applications. Based on this visual interpretation, it is generally well suitable with the perspective of mapping the design on to several processors because independent sub-units can easily be moved to separate computing platform. Also, this visual representation allows the designer to re-use the components in the form of design instances. This helps in hardware/software co-simulation and assessment of design space in HW/SW for better portioning.

There are so many types of graphs that are used for representing applications.

2.2.1 <u>Block Diagram</u>

Block diagram is a very simple form of representing designs. It has set of functional blocks communicated through a set of edges ensuring a form of connectivity between these functional blocks. These functional blocks are usually the adders, multipliers and delay elements in the circuit design. These adders, multipliers and delay elements are the basic building blocks of representing DSP designs.

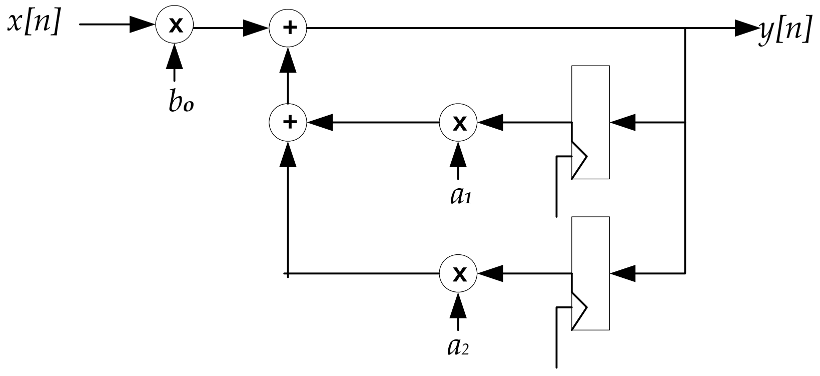


Figure 2-2: Block Diagram

2.2.2 <u>Signal Flow Graphs (SFG)</u>

Signal Flow Graphs are well thought-out the simple form of the block diagram. In SFG, the multiplier unit (that is either multiplying with a constant or a delay element) is replaced by the edge of the block diagram and the addition, subtraction and input-output relationships are ensured by the nodes.

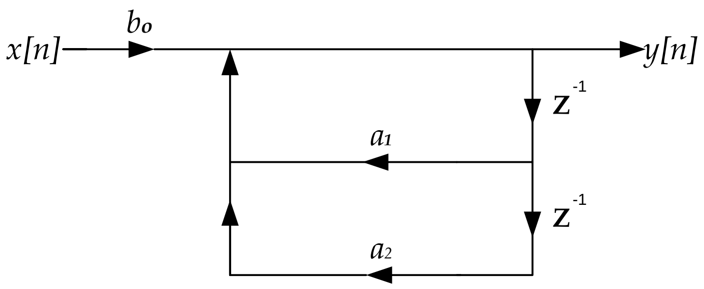


Figure 2-3: Signal Flow Graphs

SFG and block diagram are attractive for representing DSP algorithms but when there is the issue of mapping architecture, they are not well-suited because they are not focusing on any synchronization issues and data consumption and production at each node. In short, the designer can not put any limitations on the node's execution to ensure synchronization.

2.2.3 __Data Flow Graphs (DFG)__

The highly computational workstations have been a reasonable choice for intensive multimedia applications. The major demand from these workstations is the high throughput and high performance. These requirements had been for years but, with the advancements of commercial workstations, these issues are resolved. Like the hardware development costs are sharing server market, software development costs are abridged because operating systems gives upgradeability, portability and maintainability.

Kahn Process Networks (KPN) is a data-flow modeling technique that is used to model various streaming based multimedia and signal processing applications. KPN can also run in event driven applications and multi-rate systems. The key features of KPN lies in their parallelism and their communication in a specific task is explicit; means they are very suitable in multiprocessor environment. The major advantage of KPN is that they allow processes the asynchronous construct, by this the process can work independently and concurrently.

A matter of concern for my thesis is particularly a mapping of DSP designs on any reconfigurable computing platform. Data Flow graphs are the most widely used data flow graph model used for representing DSP designs. This is the flavor of DSP that the application can easily be broken into sub units. These subunits are formed in such a way that they can easily be worked independently and executed on parallel computing platform with out the threat of any data coherency issues. In such a scenario, DFG is the best model of representation. In DFG, an application is represented by a set of computing units also called nodes or vertices and set of edges. These nodes are interconnected to one another by the help of directed graphs. Each node has its corresponding edges at the input and output. Each node is defined with the corresponding number of data values also called tokens. Node will not execute as long as it finds sufficient number of data values or tokens at its respective input edges. After firing the sufficient number of tokens, the node takes throughput number if clock cycles for its internal processing. For storing these particular number of tokens on the input and output edges, we have an associated FIFO buffers for temporarily holding these data tokens. Each edge also has its associated algorithmic delays. Figure 2-4 showing the DFG model in which we have four nodes or actors A, B, C, and D. e1, e2, e3, e4 and e5 are the connected edges working as a mean of communication channel among these processing units. These edges have set of FIFO buffers for temporarily holding data values or tokens.

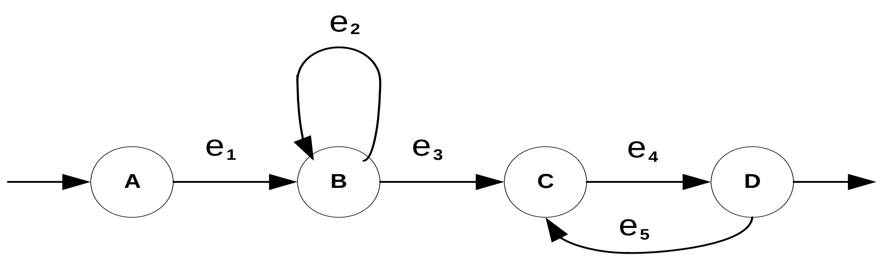


Figure 2-4: Basic DFG Model

Data flow graphs is my point of concern because they are well suited especially for representing signal processing applications. In any DSP designs, the subunits are usually multi-rate systems and their representation as a DFG is straightforward. Also, DFG representation allows the very cost-effective solution in terms of hardware reuse. Also this representation works on component basis so testing, verification and module level optimization can be carefully performed.

2.2.3.1 <u>Synchronous Data Flow Graphs (SDFG)</u>

As data flow graphs are particularly focusing on the directed graphs in which we have number of processing elements (Actors) and their associated connectivity information. When the actors are executed, they consume required number of tokens at all the input edges and after processing, send the processed data tokens to the output buffer. When this rate of consumption and production is defined at compile time, then the data flow graphs are called the Synchronous data flow graphs (SDF). This helps in the provision of so much optimization techniques. In short, in SDF, the number of token consumption and production at each link is fixed or constant. This easily helps in predicting the pattern or flow of sequence of operation and memory constraints at compile time. Also in SDF, the run time behavior is very predictable.

In Multimedia and streaming applications, usually, the application is running on fixed sampling rate ensuring the fixed or constant number of data tokens at the input and output edges. Thus for signal processing applications, each processing unit has predefined number of samples for consumption and production. For example, a decimator that is just throwing out data samples. It is defined by a rate. If it is decimating by 2 means that it will take two samples at its input edge and passes one sample to its output and throwing the second one. In this case the rate of consumption and production ratio is 2:1. Figure shows that in SDF, the nodes are represented by their names and execution time. Each edge is represented at its head and tail by the consumption and the production token rates. In this figure, we have three nodes A, B and C having execution time represented in circles as T1 and T2 and T3. Also each edge is marked with the data or token consumption and production rates. Each node will not execute its processing as long as it finds sufficient tokens on its input buffers. Also there is another parameter that can be labeled on SDF i.e. algorithmic delays. These algorithmic delays specify that Node B will take three iterations old data tokens for its firings. Conclusively, SDF is the most commonly used model for representing DSP designs that is particularly my point of concern in my thesis.

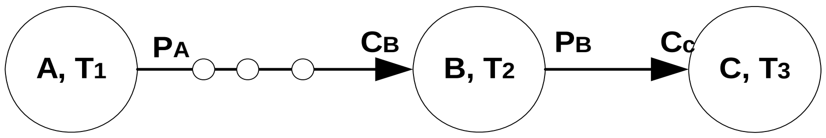


Figure 2-5: Basic DFG Model showing Nodes general characteristics

2.2.3.2 Single Rate SDFG and Multi-Rate SDFG

In Single Rate SDFG, the number of tokens consumption at the input of the node is same as the number of tokens produced by this node at its output edge. While in case of multi-rate SDFG, the rate of consumption and production parameters are not same. These specifications are relatively very important especially for designing purpose. If it is assumed that the graph is a single-rate one then the optimization techniques can easily be engaged. But in my thesis work, I am focusing on DSP designs that are usually multi-rate in nature, so optimization is a big design challenge and a key research issue as well.

2.2.3.3 Homogeneous Synchronous Data Flow Graphs (HSDF)

Homogeneous SDF is a special case of SDF. In HSDF, the rate of consumption and production parameters at each node is one. This is important because it gives the information of throughput (Number of cycles taken by node for its execution). Also, it reduces the complexity of hardware at the expense of large number of node's occurrence. Based on this occurrence, task level parallelism is possible and application can easily be run on so many parallel computing units.

2.2.4 Parameterized Synchronous Data Flow Graphs (PSDF)

SDF is particularly ineffective to dynamic behavior of node. Parameterized SDF comes in this case where structured and dynamic parameter changes are required. PSDF graph is comprised of PSDF actors and PSDF edges. The combination of PSDF actors and PSDF edges control the functionality of a node and also different configuration settings like rate of consumption and production and data flow characteristics of node. Each PSDF system consists of three graph parameters that are separately controlled, the init graph, the sub-init graph and the body graph. Init and sub-init graph control the configuration settings of bode graph while the body graph controls the main functional specifications of the processing unit or actor (Node). These init and sub-init graph reserves the previous history of node's execution as well for efficient modeling of dynamism. Figure shows a mapping of simple SDF to the parameterized Data Flow graph. Figure depicts that each node is parameterized by the three set of graphs. A.Init and A.Subinit controls the functionality of node A's body by setting the body parameters and A.body is resided with the actual functionality of node.

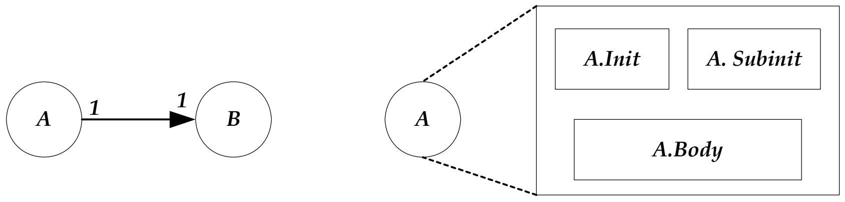


Figure 2-6: PSDF Model

2.2.5 Cyclo-Static Data Flow Graphs (CSDF)

Cyclo-Static SDF is one of most powerful extension of SDF. In CSDF, the rate of token consumption and production varies but according to the fixed periodic pattern. This periodic pattern is called the phase of the actor. This form is particularly suitable for signal processing techniques because of variable nature of data firing rate. For example an application having nodes X, Y and Z. Following the diagram shows that each node is having different number of data firing rate according to specific function execution order of the node. Node X, Y and Z have execution time of [1,3], [2,4] and [3,7] respectively. Based on specific function's execution, each node is consuming and producing varying number of data tokens.

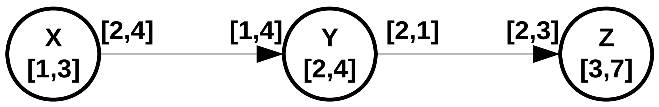


Figure 2-7: PSDF Example

Figure shows generic way of cyclo-static data flow graphs representing DSP designs modeling in such a way that each node has an execution sequence $f_j(i)$……$f_j(P_j)$ of length P_j. Each $f_j(i)$ is called the phase of the sequence. For each actor A_j, the node executes the sequence $f_j(j\%P_j)$ and produce the token rate specified by the sequence $X_j(j\%P_j)$.

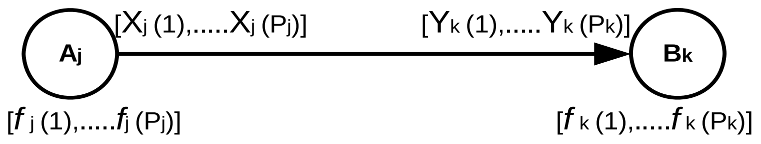


Figure 2-8: Generalized PSDF Model

2.2.6 Parameterized Cyclo-Static Data Flow Graphs (PCSDF)

As CSDF is a meta-modeling technique but it requires fixed compile time information of node's execution time and tokens consumption and production rates. Each node has a series of periodic sequences for its execution. Parameterized CSDF allows the better optimization techniques in terms of hardware reconfiguration by using dynamic behavior of node at run time. Figure depicts that, In PCSDF, each node or actor is parameterized for its functionality but here the data behaviors of each node vary cyclically. In fact, it is a cyclic pattern that does for parameterization of node.

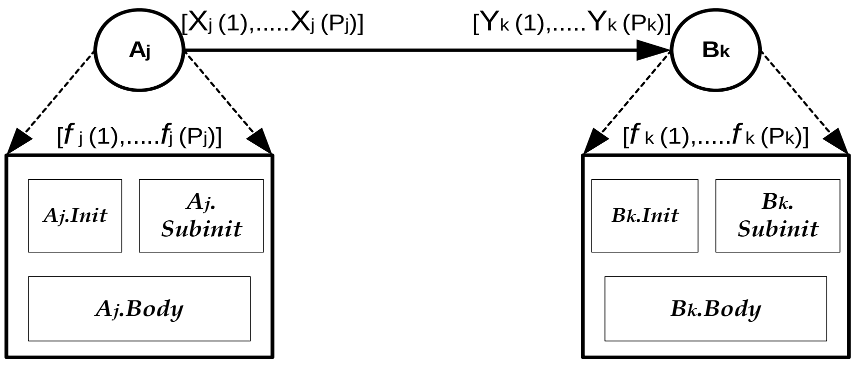


Figure 2-9: Generalized PCSDF Model

Theoretically, in PCSDF, there are two controlling parameters that are particularly concerned in the data flow behavior of PCSDF. These are

2 Period of cycle of each phase with respect to each actor.

3 Rate of consumption and production parameter for each phase with respect to each actor.

2.2.7 Dynamic Data Flow Graphs (DDFG)

In Dynamic Data Flow graphs, the number of tokens produced and consumed at run time. This gives better optimization techniques and efficient memory usage.

SECTION II
PROCESS NETWORKS MODELS

High performance systems in terms of cost, power, area, compilation time or throughput and limited memory usage are always the measures for all system designers especially on any hardware configurable platform. The main aim of any system designer is to efficiently use the available resources and keep the system at the level of the design specifications. For multimedia streaming and signal processing applications, compilation time and memory usage are the main point of threats for designers. Usually, the designer has limited memory available and its efficient use is dependent on the process's context switches. To cater this huge problem, there exist so many process network models that take cares of specific parameters of the design requirements. If we talk about particularly the multimedia and signal processing applications, timeliness and limited memory are the points of concern because, for example, delayed transmission will not make any sense in any videoconferencing session and also streaming applications deal with large number of video and audio frames and efficient memory usage come for their storage purpose. Based on these design requirements, Process Network (PN) models are particularly considered best models of computation for such applications. Usually, signal processing applications deal with real infinite amount of data samples. PN must be capable enough to automatically schedule such a continuous stream of data in a given amount of data. There are so many types of process network models but Kahn Process Network (KPN) is the commonly used one. Here we will discuss it in more detail.

2.3 **Process Network Model**

PN is a model of computation in which we have set of processing units like nodes or processes. These processes take continuous stream of data packets called tokens from infinite length FIFO (First-In First Out) buffers and write the transformed or processed data on to FIFO buffers on infinite length. KPN (Kahn Process Network is a most commonly used model of computation for streaming applications. In this typical KPN model, As FIFO buffers are assumed to be of infinite length, so writing to buffers is non-blocking but there may be the case where the process attempts to read the data from empty queue so we have a blocking read operation. In short, KPN is a computational model in which all the processing elements can run concurrently with blocking read and non-blocking write operation.

Here are the some key features of KPN model:

- Its determinism-- means it does not affect the functional behavior of application only the topology of network changes.
- The execution order does not matter—application is explicit
- Facilitate the components reuse and design complexity.
- All the nodes have their own FIFO buffers so there is no concept of global memory and it is well suitable for multiprocessor architectures—it allows task level parallelism
- Good optimization techniques can be prevailed because of easy component's handling
- All the nodes have automatic synchronization because of blocking read operation

Before further going on the details of KPN, We list some basic definitions that will be commonly used for KPN understandings.

2.3.1 **Output Completeness** – the output of this modeling must yield the same results as are defined by the algorithm. This is very important in the sense that for every raw data, we must acquire the transformed processed data according to the proposed algorithm.

2.3.2 **Execution Order** – Refers to the matter of reading from and writing into the FIFO buffers. As is defined above that the execution order is not an issue in the case of KPN because any sequence of execution will yield the same desired output. This execution order can be defined at compile time or run time. Generally, the execution order is either static or dynamic.

- **Static** – Execution order is first defined and then that order is followed in network execution. In short, the execution of processes is fixed and defined at compile time.

- **Dynamic –** Execution order is not fixed and usually defined at run time. It goes continuously changing according to network conditions.

2.3.3 **Boundedness –** The execution order helps nodes in determining the sequence of their proper execution but this cause the nodes to wait for control that will be given to them at respective times. Because of this, unconsumed tokens will be accumulated on FIFO buffers waiting for their turn of execution. Boundedness ensures that these tokens are bounded for the complete execution of the program. More specifically the network may be strictly bounded, bounded or unbounded.

2.3.4 **Strictly Bounded –** Unconsumed tokens on all FIFO queues give rise to complete execution.

- **Bounded –** Unconsumed tokens on all FIFO queues must at-least give rise to one complete execution.
- **Unbounded –** Unconsumed tokens on all FIFO queues do not give any single complete execution.

2.3.5 **Termination** – Termination relates to the number or amount of data values that are processed by the network. If all the input buffers are taking finite amount of data then at some later time, the program will definitely terminate. But if any singe input FIFO buffer takes continuous stream of data then the program will never terminate.

2.3.6 **Memory Allocation and Buffer Sizing** – Memory reservation is a definite issue especially for streaming multimedia applications because of their heavy storage requirements. As the basic KPN implementation lies on this concept that each processing element is connected with the FIFO buffers of infinite length but the memory allocation of infinite length is impossible to do. When KPN gets optimized according to desired specifications then the issue comes of the optimum size of FIFO buffers. Optimum size of the FIFO buffers is a big challenge of today's research.

2.3.7 **Artificial Deadlocks –**Artificial deadlock is a case where node gets blocked just because of the insufficient memory allocation of FIFO buffers. There are usually two drawbacks of KPN implementation with respect to memory. There may be more memory allotted between network processes or not enough memory allocation. When node gets insufficient memory then it gets blocked not because of inadequate tokens on FIFO buffers but because of small buffer sizes. There are so many researches that have been carried out in this regard. One option is that to start with some fixed size of buffers and then based on dynamic conditions/network traffic conditions, the size of buffers are varied. Ultimately all the buffers automatically reach to optimum buffer size range.

2.4 KAHN PROCESS NETWORK:

Kahn Process Network (KPN) is a computational model that provides the facility of parallel computation i.e. concurrency. At first, the application is divided into set of sub-units. It is the designer choice that how effectively the application can be broken down into components. As I have already figure out earlier that this reduces the system's complexity and the ability of component's reuse. Once the application is broken down, then it is mapped to any reconfigurable hardware platform. To configure them we have some process network models among which KPN is considered the best one.

2.4.1 KAHN PROCESS NETWORK MODEL

KPN model is commonly represented as directed graph in which all the nodes or actors are generally represented as processing elements. These processing elements are any processors that will perform some form of dedicated task. The actual component's functionality is performed at here. The inter-communication among these processing elements is ensured by designing a proper topology. In this topology, we have a complete list of connectivity and the complete FIFO buffers requirements. In KPN, each component of any streaming application i.e. processing elements communicates with one another by a set of infinite length FIFO buffers. Conclusively, network is described as a graph **G = (V, E, F),** where

V= Vertex or Node or processing element

E= Connected edges between nodes

F= Functionality defined in the network element.

Figure 2.10 demonstrates the basic KPN model in which an application is broken down into set of processing units and their communication is performed through set of FIFOs. Each processing unit is defined with its name and number of cycles or time units, they take in execution. Also, each processing node is connected to set of FIFOs at its input and output. These nodes will not execute as long as it finds desired number of RC (Rate of Consumption) tokens on its input FIFO buffers. The processing units will check on its input FIFOs, when it acquires sufficient tokens, it will start executing. The control is given to this processing unit as long as it executes. After its execution, RP (Rate of Production) number of processed tokens will be written on its output buffers. In this model, Node ‘A’ is taking the continuous stream of data. When its input FIFO buffer ‘F1’ will store two tokens, then process A will fire and takes eight cycles for its execution. After completing this processing, it will write two, three and one tokens on FIFOs ‘F2’, ‘F3’ and ‘F4’. Process ‘B’ and ‘C’ will execute when they find two and one tokens on its input FIFO buffers i.e. ‘F2’ and ‘F4’. Process ‘D’ will not execute until it finds two, three and one tokens on its input buffers i.e. ‘F5’, ‘F3’ and ‘F6’. Process ‘D’ is continuously writing data to its output buffer ‘F7’. This is how streaming applications are mapped through KPN structure.

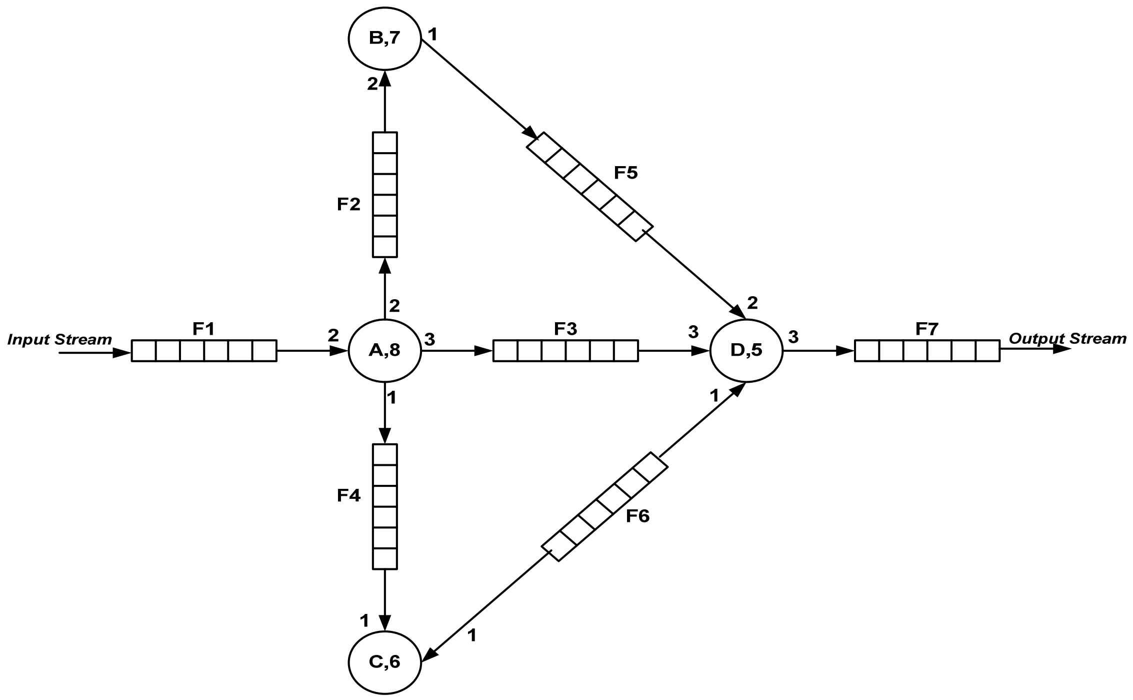


Figure 2-10: General KPN Model

2.4.2 <u>KPN FOR MODELING STREAMING APPLICATIONS</u>

Figure 2-11 shows the very basic example of JPEG compression. Implementing JPEG is a good example to explain effectiveness of KPN in streaming applications. The raw image taken from the source is saved in FIFO ‘F1’. Node ‘1’ performs the RGB to YCbCr conversion and stores the transforms image to FIFO ‘F2’. The Node ‘2’ waits to perform the conversion to take place and once FIFO ‘F2’ acquires this data, it fires and computes DCT and writes the result in FIFO ‘F3’.Now Node ‘3’ and ‘4’ sequentially fire and compute Quantization and Entropy coding and write data in FIFO ‘F4’ and ‘F5’. This is how any streaming application can be mapped through KPN structure without incurring any overheads.

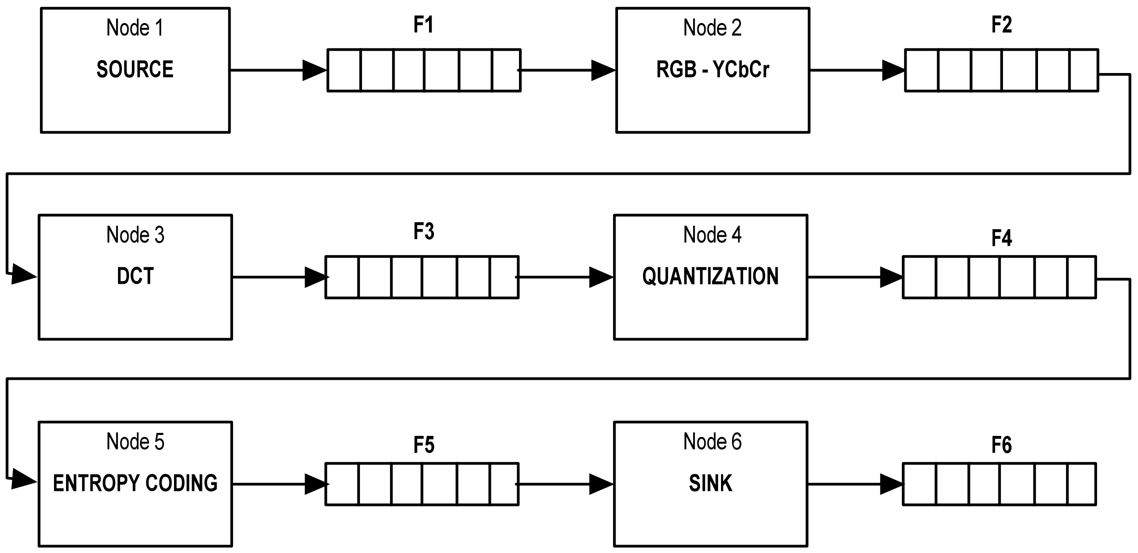


Figure 2-11: KPN modeling for streaming applications

2.4.3 ***<u>RESTRICTIONS OF KPN MODEL:</u>***

KPN follows the strict behavior of FIFO operation for buffering data. My thesis work is particularly focusing on KPN implementation for multimedia streaming applications. Particularly speaking, these applications do not follow this stringent behavior of FIFO operations causing some limitations in classical KPN model. Thus, a modified KPN will be desirable for streaming applications. Some of the major restrictions happen because of strict FIFO behavior are

- Reading of data or tokens from the FIFO buffers require strict FIFO operation but there are so many signal processing techniques that do not require such a stringent behavior for their execution. For example, if the application mode is performing decimation in time then it doe not need all the consecutive data tokens. This is one of the major flaws because of classical KPN implementation.

- There are so many signal processing applications that require the data multiple times. For example, in convolution, the node performs point to point multiplication and then additive sum to generate one output sample. After that, there is a simple shift by one in time domain and then again node performs this same operation using the old previous stored data tokens. But in this standard implementation, once the data gets read from the FIFO buffer then it is flushed out from the memory without considering any behavior of node at run time.

- There may be some cases when the node does not have need of the data currently stored/available in FIFO buffer but the node can not take the desired data until this useless information will first extract from the buffer. Thus node does not need that where data is read sparsely.

2.4.4 ***CUSTOMIZED KPN MODEL:***

They are so many solutions of handling this problem occur just because of typical FIFO behavior. The very simple solution is use a local memory provided to each network node. This local memory is meant for keeping the copy of data that node is expecting to use in near future.

KPN model comes in so many implementation module among them MPSoC (Multi-Processors System on Chip) is considered the best embedded system design for multimedia streaming applications. In MPSoC, we have some form of KPN to model the problem and then it can automatically be transformed to MPSOC. Figure 2-12 demonstrates the basic MPSOC representation. In MPSOC, the KPN is used to model the application on the set of independent processing hardware. This application is modeled in such a way that all the processing elements or processors can run independently using their own local memory. The intercommunication among these processors is ensured through a NOC (Network On Chip) switch bar. Each processing element or processor is generally called a tile. Each tile has its own local memory M, the memory controller MC and set of FIFO buffers. The communication among FIFOs lying on different processors is ensured by a cross bar switch, a P2P network or a shared bus or a more elaborated form of NOC (Network-On-Chip) component bar. Each tile also has a logic called Communication Controller CC.

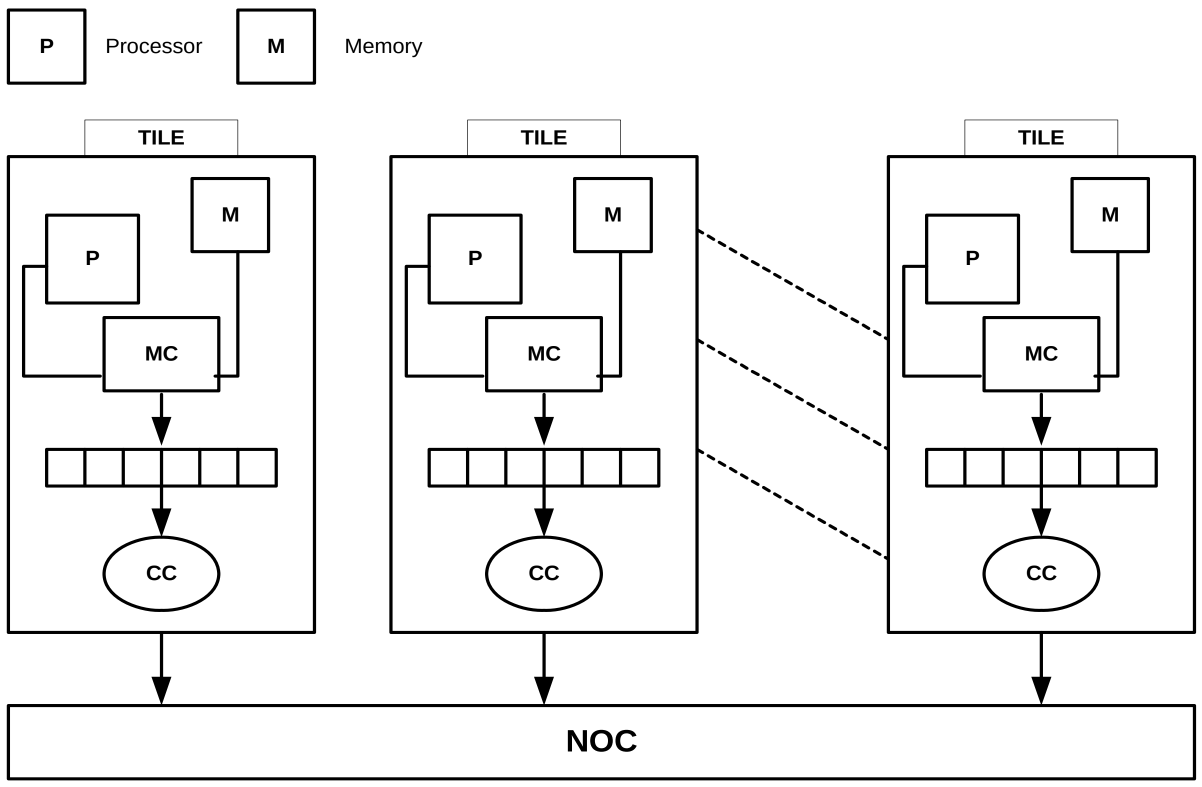


Figure 2-12: General MPSoC Model

This is where my thesis work actually starts. I am particularly concerned about the mapping of multimedia streaming applications on to this processing hardware i.e. MPSOC. I have been assigned a task that to design a generic controller that automatically and efficiently maps any kind of streaming applications on to the reconfigurable platform i.e. MPSOC. There are some researches that have been carried out in efficient execution of process networks and the design of real time scheduler for KPN on multiprocessor system.

2.4.5 ***EFFICIENT KPN SCHEDULING***

All the applications require their efficient mapping on to the multiprocessor environment. Thus a generic controller or scheduler is an essential component for designer. The main role of scheduler is to provide the control to the particular node for their turn to execution. Usually this scheduling can be performed statically or dynamically. Static scheduling though is a simpler one which is defined at compile time but no doubt it does not give the efficient algorithm implementation. Also, static scheduler for particularly streaming multimedia applications is not feasible because of built-In dynamism in such applications.

Dynamism is a main point of concern for my thesis. It requires the efficient modeling of ready processes at run time. There are usually two approaches for determining the set of ready processes at run time. These approaches are broadly categorized as

- Demand Driven Scheduling
- Data Driven Scheduling

In Demand driven scheduling, the scheduling comes by the actual demand of data. As the demand arises then the ready processes start reading data form input FIFO buffers. Generally the demand is originated from the output node. As the demand is propagated, the set of ready processes become active and they try to read data from their input buffers but if they try to read data from some empty buffers then the ready processes go to blocked mode and wait until they acquire the sufficient tokens in all their input FIFO buffers. Meanwhile the set of all the ready processes start their execution if they have required number of data tokens. The main objective of the demand driven scheduling is to perform execution only when is needed.

In data driven scheduling, the ready processes always keep themselves in the polling state. It continuously checks the input buffers. When the node gets desired number of tokens on all its input buffers then it starts performing its execution. This scheduling technique is meant for continuous node’s execution as long as it has required number of tokens. The process will only stop when it will have no longer data available on its input buffers. Both techniques have their respective disadvantages. The main disadvantage of data driven approach is that it will allow process to run with out considering this fact that whether it is required for output node. There may be the case that the intermediate nodes continuously run and data gets overwhelmed in the intermediate buffers. The main drawback of the demand driven is that it will run give control to processes only when is needed but it requires so many context switches and complexity in hardware for flooding the demand information in network’s topology.

The final approach is to design a data-driven scheduler with bounded FIFO size. The classical KPN focuses on the unlimited memory size which is non-realizable. The fixed buffer size makes modification in basic KPN model. Now the process will go to blocked state not only when it is reading from empty FIFO or insufficient tokens availability case but also when it attempts to write in fully loaded FIFO. This deadlock is usually called artificial deadlock because it is an artificial one generated because of empty or full FIFO. In this final approach, the processes are scheduled on data driven strategy with efficient memory utilization but the optimum buffer sizing is a big research challenge.

CHAPTER 3

SYSTEM DESIGN

This section describes the implementation details of our research strategy. There are so many types of data flow graphs but DFG are the most commonly used representation for DSP designing perspective. Our aim is to design a generic system model that visualizes this DFG model and based on these specifications, it automatically generates a RTL high level synthesized implementation of KPN that can easily be mapped on any reconfigurable platform and also on MPSOC. Our design starts with the configuration file that lists all the necessary parameters to generate automatic controller for critical DSP designs. Also, we want that this controller must be very efficient in terms of hardware requirements i.e. it must utilize optimum size of memory buffers and other hardware units. Lastly, we really want to make certain that this controller must satisfy the flavors of KPN that makes it valuable to other existing mapping schemes.

3.1 SYSTEM MODEL:

Figure 3.1 shows the system model of our proposed scheme. In this model, we have first designed the configuration file. This file is designed in such a way that it takes all the requirement specifications for generating an automatic KPN. It list the total number of nodes, their interconnections, Rate of consumption and Rate of production parameters (These parameters the required number of token at the input and output of node to perform its processing/execution), the required number of FIFOS, information about each link, Algorithmic delays, self loop information and last but not the least the input and output streams that will carry the actual data. Input stream relates to the raw data that needs to be processed by our network and output stream is a continuous ejection of processed data from out proposed model. This configuration file is passed to our network model that is a C-based compiler. This compiler takes this configuration file as an input and then automatically generates the high level RTL based synthesized code for this particular application program. The number of files that are generated automatically by our compiler are the controller file that is managing all the intercommunication between different process nodes. It actually sends control signals to each sub unit for a certain level of synchronization. Also it generates a FIFO file that is fulfilling the basic operation of any FIFO buffer. Also this FIFO file tells the information of number of tokens resided in FIFO memory. Based on this information, the compiler manipulates the requirement that the sufficient number of tokens have been stored in this FIFO or not. If so, then the controller sends the control signals to the respective process node to start its execution and acquire the required token at its input FIFO. Also this compiler generates another set of files that are actually describing the N number of process nodes. These process nodes are actually the computation units that are the particular sub-unit of the application and these sub-units actually derive the raw input data and transform them in to the desired processed data. Our compiler focuses on these process nodes in such a way that configuration tells about the number of cycles that are needed for the execution of a particular process nodes, the compiler waits for these throughput number of clock cycles in the intention that the process node is processing at that time. In short, the control is provided to process node that then performs its execution and writes the processed output to its output FIFO buffer.

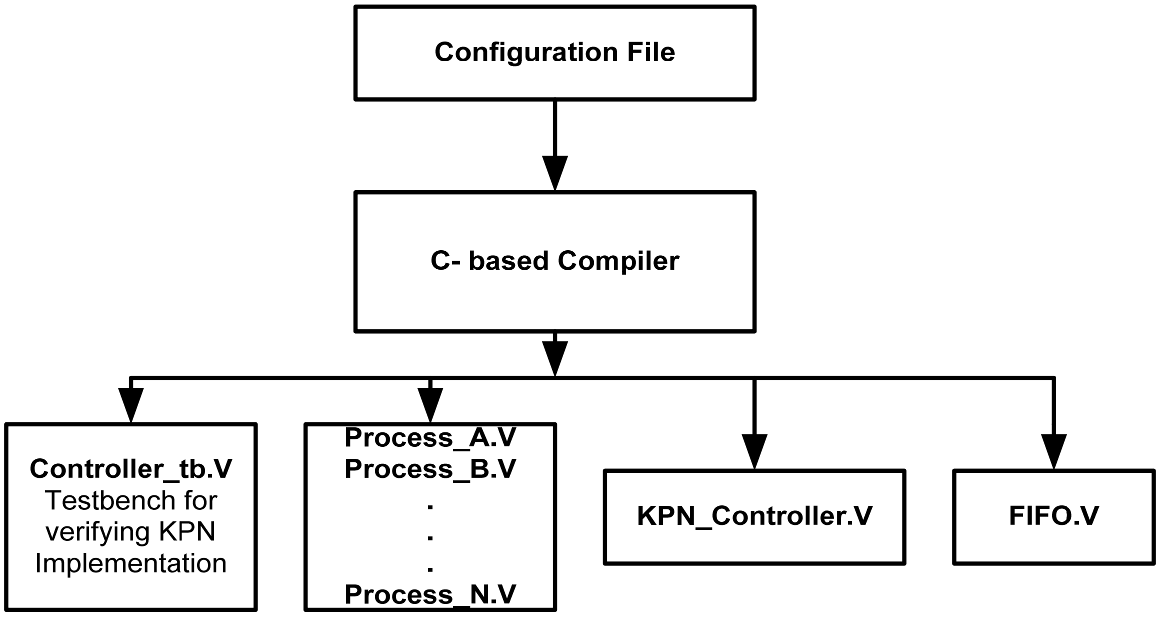


Figure 3-1: System Model

3.2 ***THESIS ORGANIZATION:***

Figure 3.2 shows my thesis work organization. At first, I put my idea to realization with the help of MATLAB tool. MATLAB was used at first to verify my designing as well as the required number of variables used for my design modeling. Then, I designed a configuration file that lists all the requirement specifications of the design. After verifying my design in MATLAB, its simulation verification and extracting parameters, I designed a manual KPN controller for a specific application i.e. for specific DFG. This was needed to extract the parameters that need to be generalized for generic KPN controller. Also, by this manual KPN controller, my design got verified on hardware or any reconfigurable platform. After that, I designed the final version of my KPN controller in Visual C++ that is actually a C-based Compiler generating an automatic synthesized controller and test bench for any given application. At the end, I calculated the performance of generalized controller with the manual controller and verified that my controller is working at the par with the manual controller that is very time consuming to design. Manual controller is specific for specific application and if design goes fail then designers have to regenerate again a new modified controller.

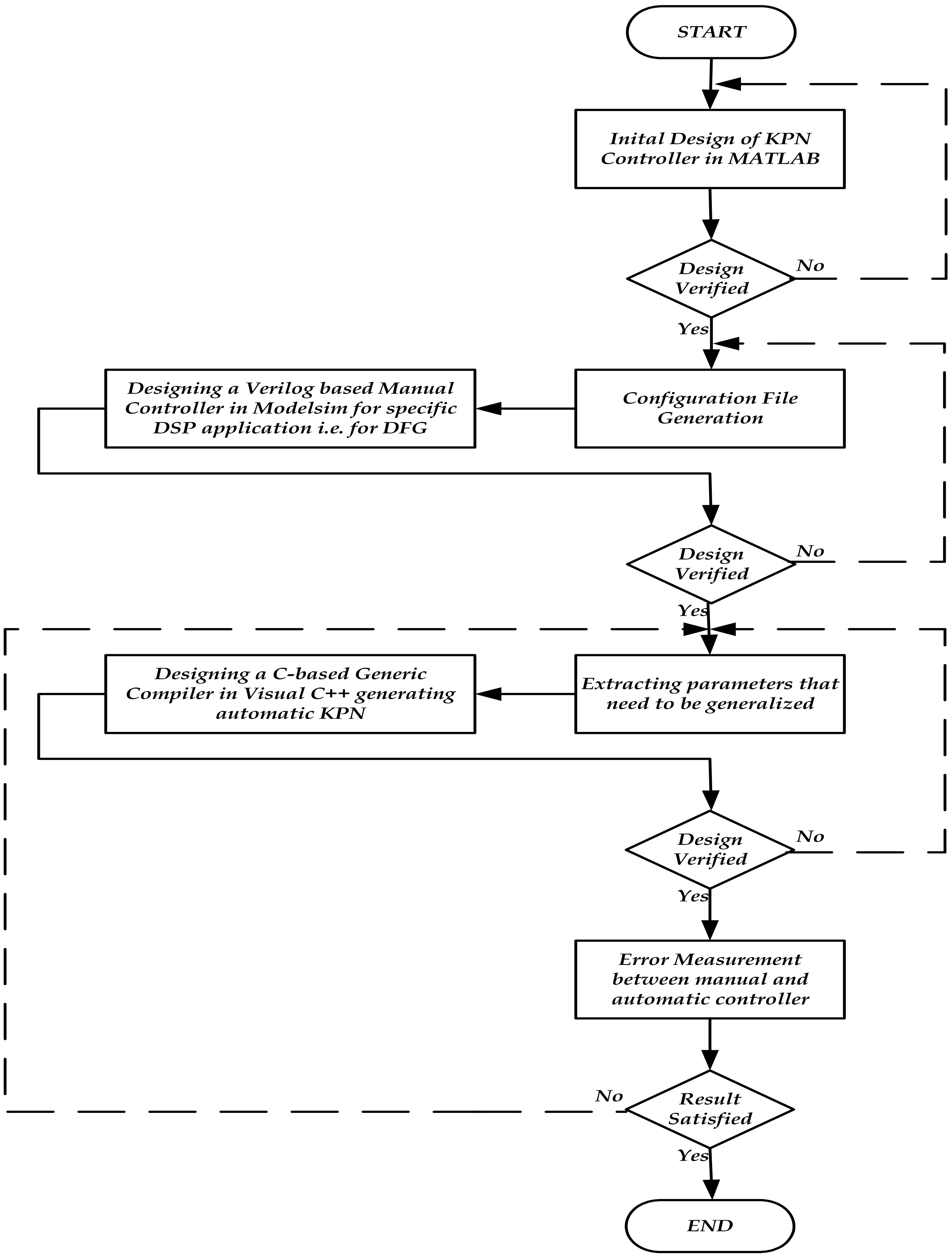

START
Inital Design of KPN Controller in MATLAB
Design Verified
No
Yes
Configuration File Generation
Designing a Verilog based Manual Controller in Modelsim for specific DSP application i.e. for DFG
Design Verified
No
Yes
Extracting parameters that need to be generalized
Designing a C-based Generic Compiler in Visual C++ generating automatic KPN
Design Verified
No
Yes
Error Measurement between manual and automatic controller
Result Satisfied
No
Yes
END

Figure 3-2: Thesis Organization

3.3 ALGORITHM:

1.) KPN_Controller(); //Main File

```
Nodes ← Number of available nodes
Links ← Total number of available links
FIFOs ← Total number of FIFOs required, storing information at links
For i ← 1 to Links
    RC[i] ← Rate of consumption parameter at link i
    RP[i] ← Rate of consumption parameter at link I
    Delay[i] ← Algorithmic delay at link i
End for
Topology Matrix Generation based on link information
For i ← 1 to Nodes
    Throughput[i] ←Execution time for node i
End for
For i ← 1 to Nodes
    Sufficient # of RC tokens found at its each connected Links
    Call FIFO_Read(); //Read Tokens from Respective FIFOs
    Done=Call Process_node(); //Functionality of node is performed
                              //that takes Throughput # of cycles
    If(Done)
        Call FIFO_Write(); //Write processed data tokens to output FIFOs
    End If
End for
```

End of Procedure "KPN_Controller();"

2.) 2.) FIFO_Read(); //Reading Data from FIFO

```
If(Read)
    If(FIFO_Empty_Flag)
        Process is Blocked
    Else
        Output ← FIFO(index)
    End If
Else
    Do Nothing
End If
```

End of Procedure "FIFO_Read()"

3.) 3.) FIFO_Write(); //Writing Data to FIFO

```
If(Write)
    If(FIFO_Full_Flag)
        Process is Blocked
    Else
        FIFO(index) ← Input Data
    End If
Else
    Do Nothing
End If
```

End of Procedure "FIFO_Write()"

4.) Process();

```
For i ← 1 to throughput
    //Processing;
End For
Done =1;
Return (Done);
```

End of Procedure "Process"

This algorithm states the functionality of these set of files that are generated automatically by my C-based compiler. These files are elaborated shortly at here.

1. ***KPN_Controller.V***

   This is my main file that is sending control signals to all the other modules and manages the complete coordination and timing constraints among each component. It is a central controller that will continuously view the status of each element and provide the control to each block when ever is desirable.

2. ***FIFO.V***

   FIFO Verilog File that provides basic FIFO operation that involves simple reads and writes into FIFO. It also gives the information of rate of consumption status of the FIFO. By, this, we can calculate whether RC tokens are accumulated in FIFO buffer or not.

3. ***Process.V***

   This compiler generates set of Verilog files depicting all the processing nodes behaviors. N processing nodes files are generated specifying number of data input units, output units and total number of time units by each processing node to perform its successful execution.

4. ***Test_Bench.V***

   This C-based compiler also generates the Verilog based test bench module that verifies the controller behavior managing all the sub units in the design.

5. ***B_Counter.V***

   This is a simple bit counter that is called by the FIFO module. This will cause the read and write pointer of FIFO to increment. This increment is performed based on the conditions that whether data has been read or not on current read pointer location and whether data has been written on current write pointer location or not.

6. ***Test_RC_RP.V***

   The node can not execute as long as it acquires sufficient number of tokens from all its respective FIFO buffers. This module checks at the abstract level that all the FIFOs connected to the particular node have adequate number of data tokens or not.

CHAPTER 4

EXPERIMENTAL RESULTS

EXAMPLE 1:

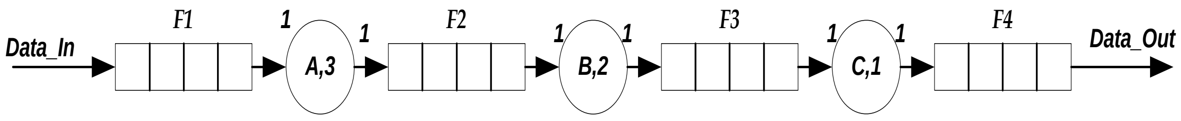


Fig. 1: Example 1

Description:

This DFG shows that there are three processing elements A, B and C. These processing elements are taking 3, 2 and 1 clock cycles for their execution. Also, these processing nodes will execute only as long as it acquires sufficient number of data tokens on their input buffers. There are four FIFO buffers for temporarily holding data values. Also, nodes are listed with their rate of consumption and production parameters. Rate of consumption is defining the number of data tokens sufficient for node to process and rate of production parameter defines the number of data values that are produced by the respective processing element.

Configuration File Specifications

Nodes=3;

FIFO_Buffers=4;

Token_Size=8;

Topology_matrix =   [1, -1, 0, 0, 0; 0, 1, -1, 0, 0; 0, 0, 1, -1, 0; 0, 0, 0, 1, -1];

Self_Loops=[0 0 0];

Algo_Delays=[0 0 0 0];

Data_In=1:20;

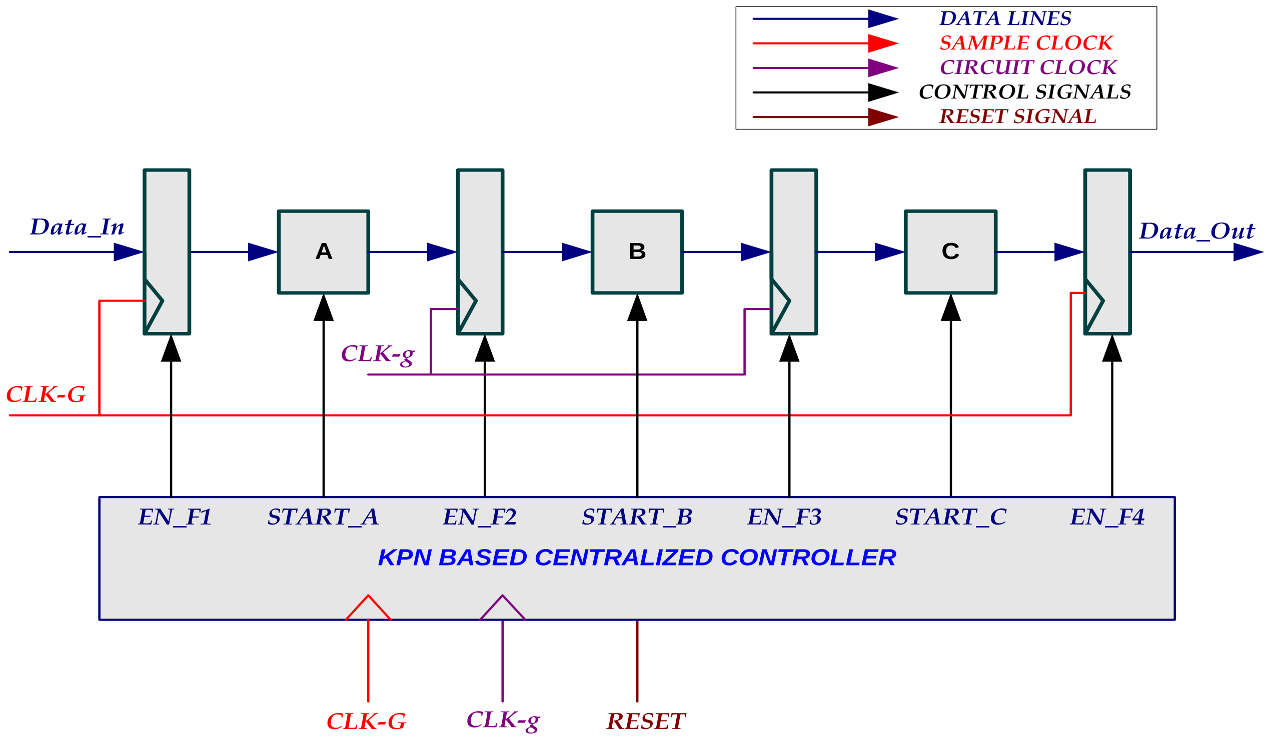
Throughput_node1=3;

Throughput_node2=2;

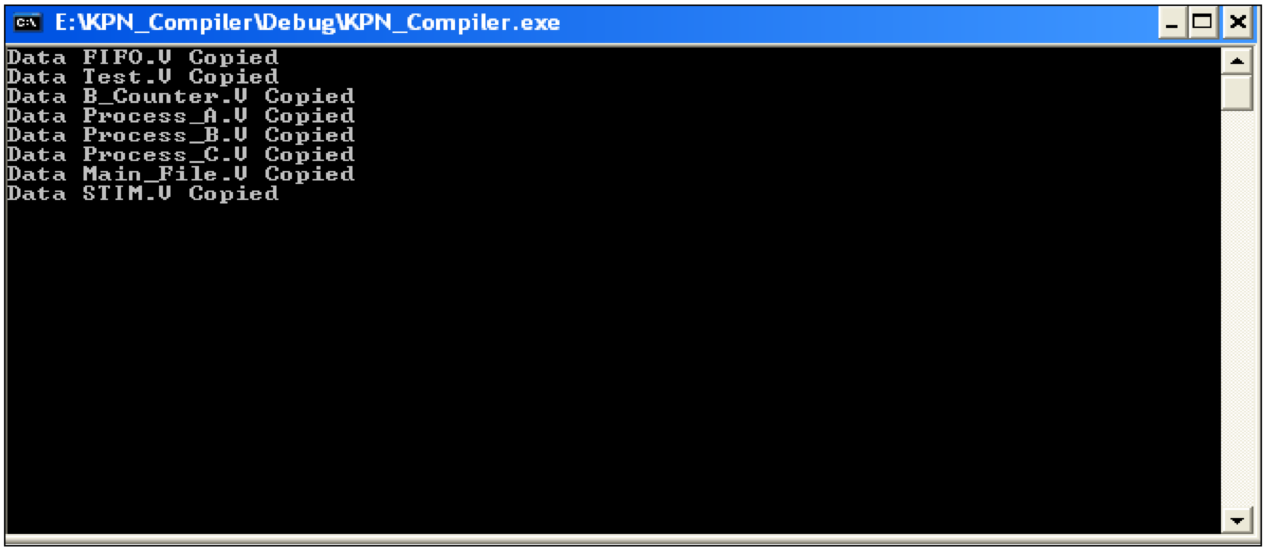
Throughput_node3=1;

Based on these specifications in MATLAB, C-based configuration file is generated that will be passed to C-based designed compiler. Then, this compiler will automatically generate synthesized HDL code of this DSP design.

<u>KPN Based Centralized Controller</u>

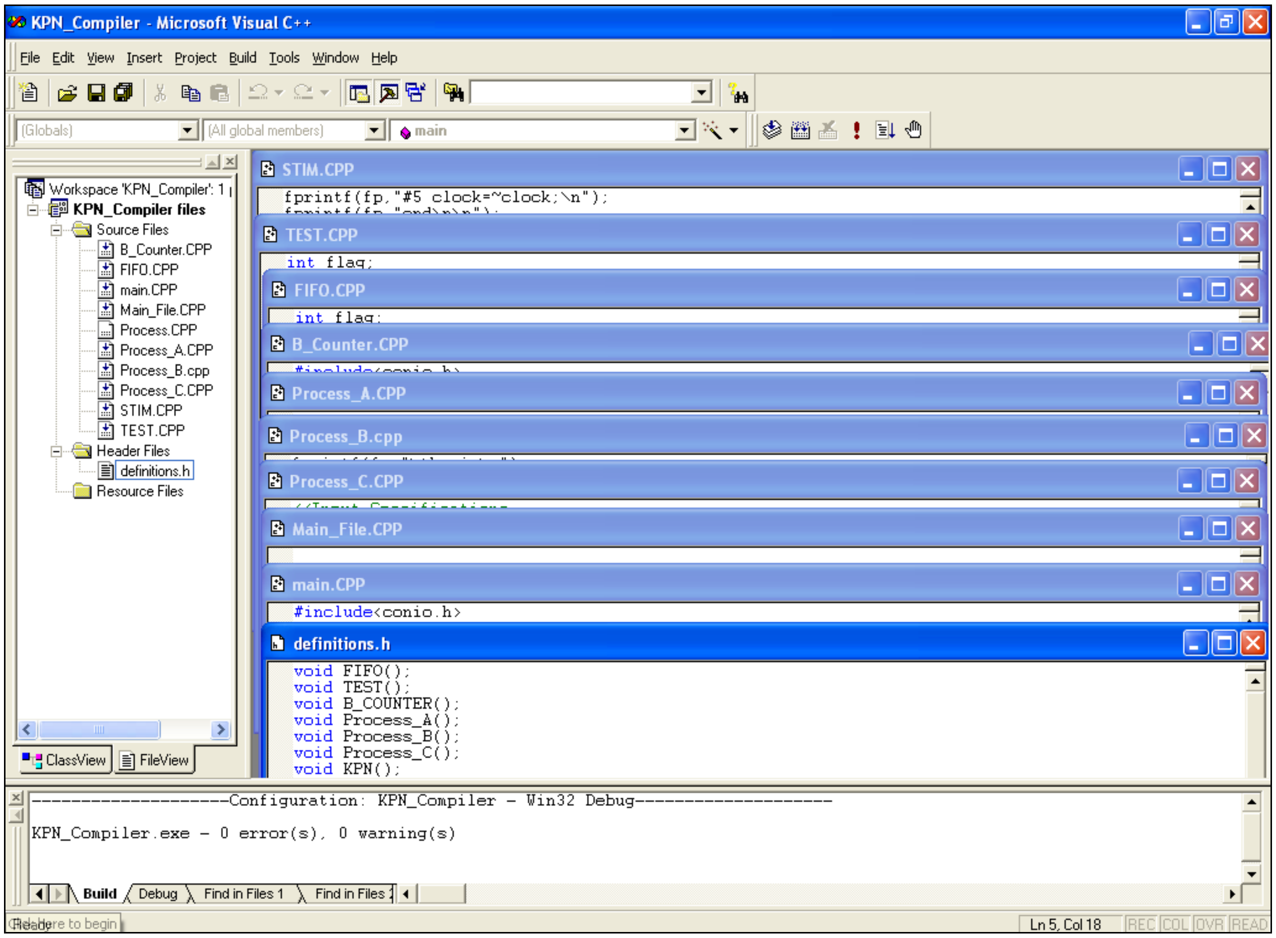


COMPILER OUTPUT



```
Data FIFO.V Copied
Data Test.V Copied
Data B_Counter.V Copied
Data Process_A.V Copied
Data Process_B.V Copied
Data Process_C.V Copied
Data Main_File.V Copied
Data STIM.V Copied
```

CODE GENERATION TOOL SNAPSHOT



```
fprintf(fp,"#5 clock=~clock;\n");
```

```
int flag;
```

```
int flag;
```

```
#include<conio.h>
```

```
void FIFO();
void TEST();
void B_COUNTER();
void Process_A();
void Process_B();
void Process_C();
void KPN();
```

```
--------------------Configuration: KPN_Compiler - Win32 Debug--------------------

KPN_Compiler.exe - 0 error(s), 0 warning(s)
```

Verilog Source Code of given DSP Design generated by C-Compiler:

1. KPN_Controller.V

```
module main_File(wr_en1,rc_n1,rc_n2,rc_n3,data_in,
                data_out,thru_n1,thru_n2,thru_n3,clk,reset);
                //Here i assumed the token size to be 08 bits
//Input Signals
input clk,reset;
//Write enable signal for fifo_1
input wr_en1;


//Throughputs for each node
input [3:0]thru_n1,thru_n2,thru_n3;
//Input Stream data will bestored at here
input [7:0]data_in;


//Rate of consumption at each link
input [23:0]rc_n1,rc_n2,rc_n3;
//Output Data
output [7:0]data_out;
reg [7:0]data_out;
//Temporary wires and Registers holding Outputs from FIFOs and processes
//and holding control signals as well
wire [7:0]d_out1,d_out2,d_out3,d_out4;
      wire [7:0]Output1,Output2,Output3;
      wire rd_en1,rd_en2,rd_en3,rd_en4,wr_en2,wr_en3,wr_en4;
```

```
reg read1,read2,read3,
   write2,write3,write4;
//Flags showing FIFO Status i.e (Fifo Full and Fifo Empty)
wire f_full_flag1,f_full_flag2,f_full_flag3,f_full_flag4,
    f_empty_flag1,f_empty_flag2,f_empty_flag3,f_empty_flag4;

//Enable the process and Done Signals showing process has completed its execution
//and ready to write data at ints output buffer
wire pr_en1,pr_en2,pr_en3,
    pr_done1,pr_done2,pr_done3;
//Pointer that stores the Difference between w_ptr and r_ptr
wire [3:0]diff_ptr1,diff_ptr2,diff_ptr3,diff_ptr4;

//Call FIFO Instances
fifo f1(diff_ptr1,d_out1,f_full_flag1,f_empty_flag1,data_in,rd_en1,wr_en1,clk,reset);
fifo f2(diff_ptr2,d_out2,f_full_flag2,f_empty_flag2,Output1,rd_en2,wr_en2,clk,reset);
fifo f3(diff_ptr3,d_out3,f_full_flag3,f_empty_flag3,Output2,rd_en3,wr_en3,clk,reset);
fifo f4(diff_ptr4,d_out4,f_full_flag4,f_empty_flag4,Output3,rd_en4,wr_en4,clk,reset);
//Process Instantiations
ProcessA A1(clk,reset,pr_en1,d_out1,Output1,thru_n1,pr_done1);
ProcessB B1(clk,reset,pr_en2,d_out2,Output2,thru_n2,pr_done2);
ProcessC C1(clk,reset,pr_en3,d_out3,Output3,thru_n3,pr_done3);

//Test Module checking sufficient tokens have acquired in each fifo to start execution
test t1(rc_n1,diff_ptr1,4'b0,4'b0,4'b0,4'b0,pr_en1);
test t2(rc_n2,4'b0,diff_ptr2,4'b0,4'b0,4'b0,pr_en2);
test t3(rc_n3,4'b0,4'b0,diff_ptr3,4'b0,4'b0,pr_en3);
//Simple assignments
//assign data_out=d_out4;
assign rd_en1=read1;
assign rd_en2=read2;
assign rd_en3=read3;
assign rd_en4=1;
assign wr_en2=write2;
assign wr_en3=write3;
assign wr_en4=write4;

/////Writing in Output Buffer at each posedge of sample clock
always @(posedge clk)
begin
        if(reset)
        begin
                data_out<=0;//write4=wr_enn4; write6=wr_enn6;
        end
        else
        begin
                data_out<=d_out4;
        end
end

//When Pr_en1 is active high or low
```

```
always @(posedge clk)
if(pr_en1)
        begin if(pr_done1)
                begin read1=1;end
        else
                begin read1=0;end
        end
else
        begin read1=0;end

        //When pr_en2 is active high or low
always @(posedge clk)
if(pr_en2)
        begin if(pr_done2)
                begin read2=1;end
        else
                begin read2=0;end
        end
else
        read2=0;

//When pr_en3 is active high or low
always @(posedge clk)
if(pr_en3)
        begin if(pr_done3)
                begin read3=1;end
        else
                begin read3=0;end
        end
else
        begin read3=0;end

///////////////Enabling Write Pointer of FIFOs after Process's execution
//When pr_done1 is active high or low
always @(pr_done1)
if(pr_done1)
        if(pr_en1)
        write2=1;
        else
        write2=0;
else
        write2=0;
//When pr_done2 is active high or low
always @(pr_done2)
if(pr_done2)
        if(pr_en2)
        write3=1;
        else
        write3=0;
else
        write3=0;
```

```
//When pr_done3 is active high or low
always @(pr_done3)
if(pr_done3)
        if(pr_en3)
        begin write4=1;end
        else
        begin write4=0;end
else
        begin write4=0;end
        endmodule
```

2. FIFO.V

```
//================================================
//fifo.v; verilog code for asynchronous FIFO
//This module describes FIFO
//================================================

module fifo(diff,d_out,f_full_flag,f_empty_flag,d_in,r_en,w_en,clk,reset);

parameter width=8; //FIFO width
parameter f_depth=16; //FIFO depth
parameter f_ptr_width=4; //because depth =16;

output [width-1:0] d_out; reg [width-1:0] d_out; //outputs
output f_full_flag,f_empty_flag;
output [3:0]diff;
input [width-1:0] d_in;
input r_en,w_en,clk;
input reset;
//internal registers,wires
wire [f_ptr_width-1:0] r_ptr,w_ptr;
reg r_next_en,w_next_en;
reg [f_ptr_width-1:0] ptr_diff;
reg [width-1:0] f_memory[f_depth-1:0];

assign diff=ptr_diff;
assign f_full_flag=(ptr_diff==(f_depth-1)); //assign FIFO status
assign f_empty_flag=(ptr_diff==0);
//instantiate address counters
b_counter r_b_counter(.c_out(r_ptr),.c_reset(reset),.c_clk(clk),.en(r_next_en));
b_counter w_b_counter(.c_out(w_ptr),.c_reset(reset),.c_clk(clk),.en(w_next_en));
//---------------------------------------------------------
always @(posedge clk) //write to memory
begin

if(reset)
        d_out<=0; //f_memory[r_ptr];
if(w_en)
begin
                if(!f_full_flag)
```

```
                    f_memory[w_ptr]<=d_in;
               end
if(r_en)
begin
                    if(!f_empty_flag)
                    d_out<=f_memory[r_ptr];
               end
end
//-------------------------------------------------------------
always @(*) //ptr_diff changes as clock changes
begin
          if(w_ptr>r_ptr)
                    ptr_diff<=w_ptr-r_ptr;
          else if(w_ptr<r_ptr)
                    ptr_diff<=((f_depth-r_ptr)+w_ptr);
          else ptr_diff<=0;
end
//-------------------------------------------------------------
always @(*) //after empty flag activated fifo read counter should not increment;
begin
          if(r_en && (!f_empty_flag))
                    r_next_en=1;
          else
                    r_next_en=0;
end
//------------------------------------------------------------
```

```
always @(*) //after full flag activated fifo write counter should not increment;
begin
        if(w_en && (!f_full_flag))
                w_next_en=1;
        else
                w_next_en=0;
end
//--------------------------------------------------------
endmodule
```

3. B_COUNTER.V

```
//===================================================
//b_counter.v; 4 bit asynchronous binary up counter
//===================================================

module b_counter(c_out,c_reset,c_clk,en);
parameter c_width=4; //counter width
output [c_width-1:0] c_out; reg [c_width-1:0] c_out;
input c_reset,c_clk,en;

always @(posedge c_clk or posedge c_reset)
        if (c_reset)
                c_out <= 0;
        else if(en)
                c_out <= c_out + 1;
endmodule
```

```
//================================================
```

4. PROCESS_A.V

```
module ProcessA(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;

//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1;
//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                 Th_Counter=Count_Inc;
           else
                 Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
else
        if(pr_en)
        pr_done=0;
        else pr_done=1;
endmodule
```

5. PROCESS_B.V

```
module ProcessB(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);
parameter width=8; //FIFO width
```

```
input [width-1:0]Input1;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;

//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1;
//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
else
        if(pr_en)
        pr_done=0;
        else pr_done=1;
endmodule
```

6. PROCESS_C.V

```
module ProcessC(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;
```

```
//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1;

//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
```

```
            pr_done=1;
            #5 Th_Counter=0;
end
else
            if(pr_en)
            pr_done=0;
            else pr_done=1;
endmodule
```

7. TEST.V

```
module test(RC,diff1,diff2,diff3,diff4,diff5,pr_en);


input [19:0]RC;
input [3:0]diff1,diff2,diff3,diff4,diff5;


output pr_en;
reg pr_en;


always @(diff1 or diff2 or diff3 or diff4 or diff5 or RC)
if(diff1 >= RC[3:0] && diff2 >= RC[7:4] && diff3 >= RC[11:8] &&
diff4>=RC[15:12] &&
   diff5>=RC[19:16])
            begin
            pr_en=1;
            end
else
```

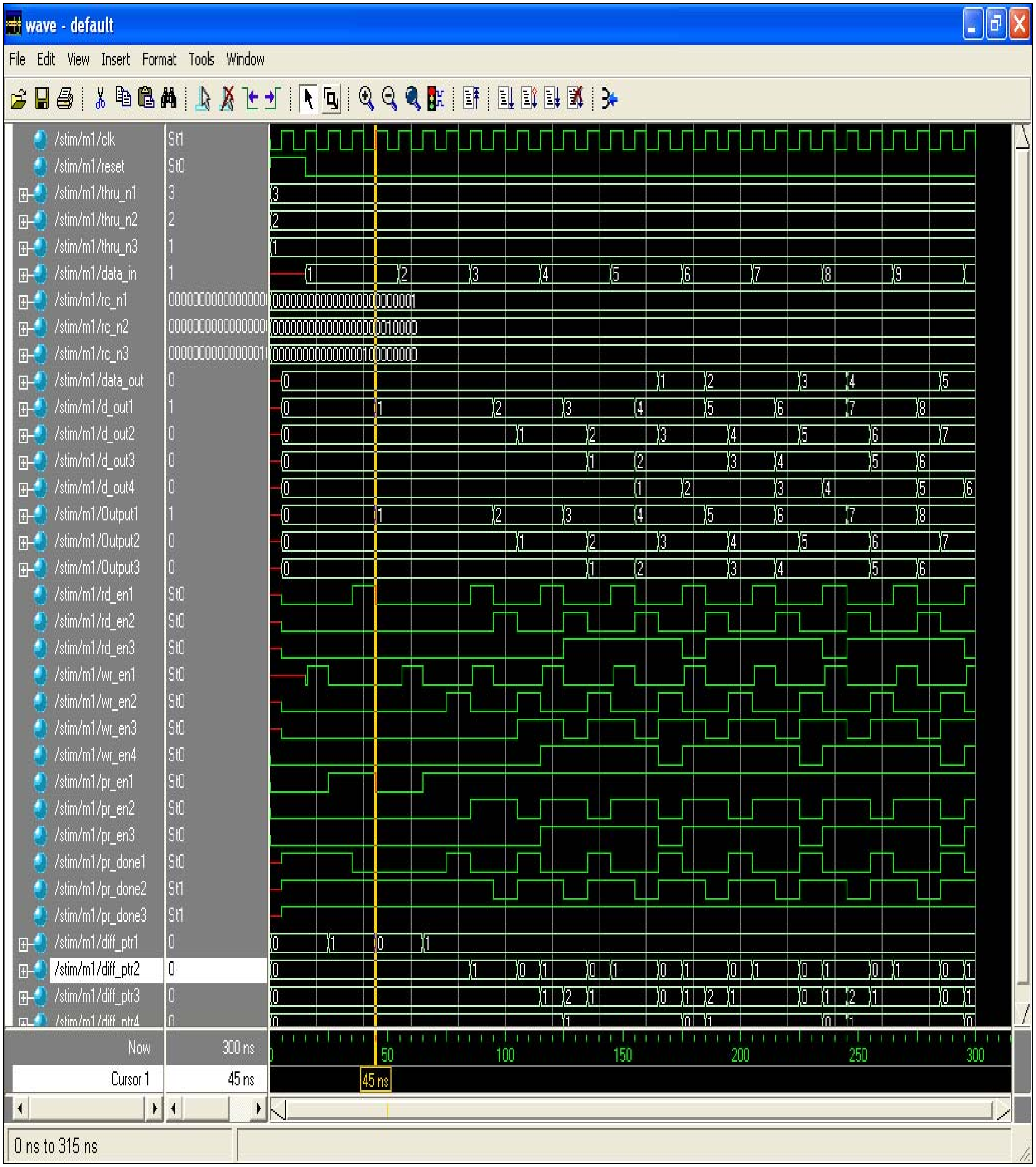
```
          begin
          pr_en=0;
          end

endmodule
```

8. STIMULUS.V

```
module stim;

reg clk,reset;
reg wr_en1;
reg [3:0]thru_n1,thru_n2,thru_n3;
reg [7:0]data_in;
reg [23:0]rc_n1,rc_n2,rc_n3;

wire [7:0]data_out;

main_File m1(wr_en1,rc_n1,rc_n2,rc_n3,data_in,data_out,
              thru_n1,thru_n2,thru_n3,clk,reset);

initial
begin
clk=0;
forever
#5 clk=~clk;
end
```

```
initial
begin
rc_n1=20'b0000_0000_0000_0000_0001;
rc_n2=20'b0000_0000_0000_0001_0000;
rc_n3=20'b0000_0000_0001_0000_0000;

thru_n1=3;thru_n2=2;thru_n3=1;


reset=1;
#15 reset=0;


data_in=1;
#10
repeat(15)
#30 data_in=data_in+1;
end


initial
#200 $stop;


initial
begin
wr_en1=1;
end
endmodule
```

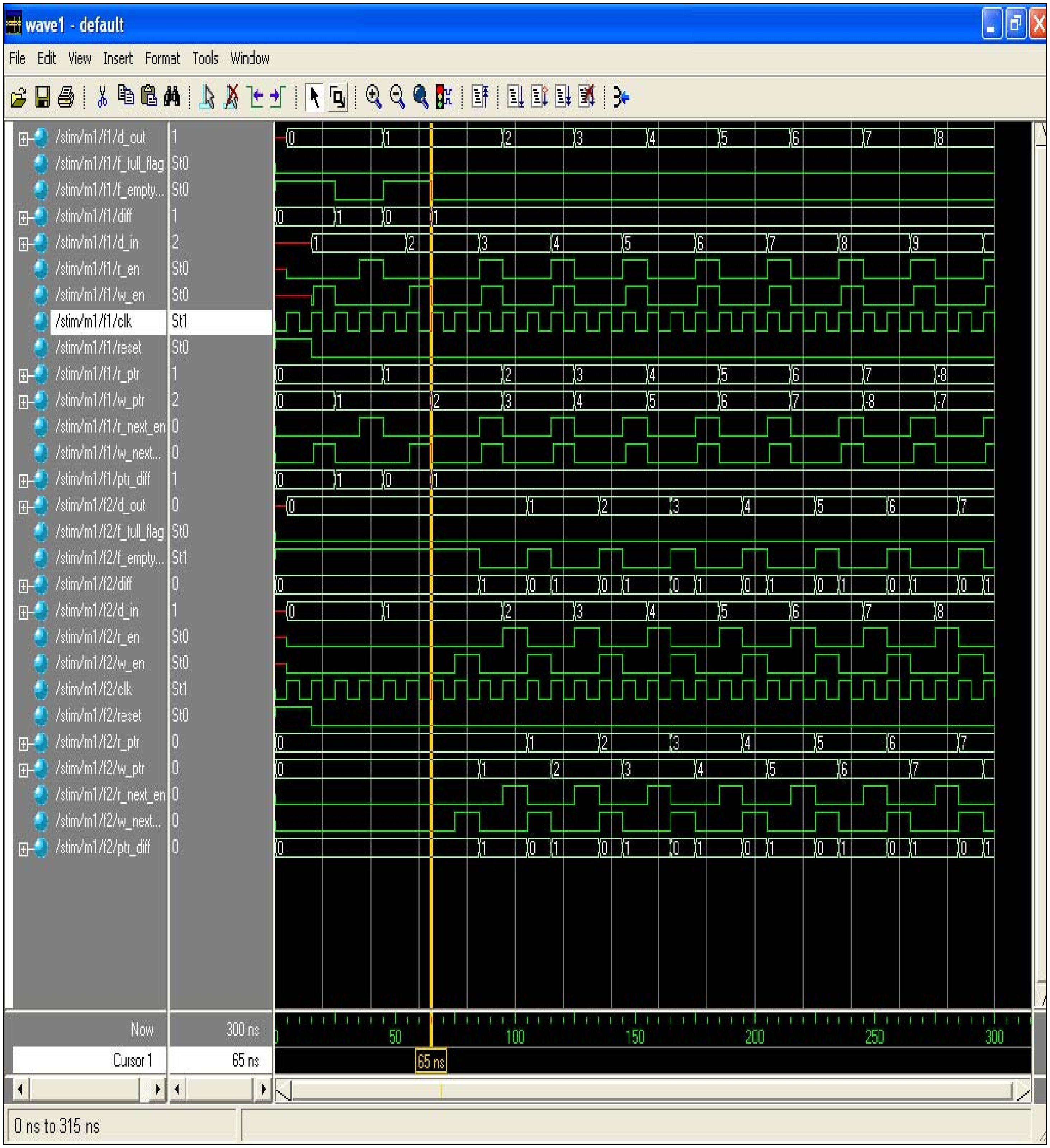
SIMULATION RESULTS



/stim/m1/clk	St1
/stim/m1/reset	St0
/stim/m1/thru_n1	3
/stim/m1/thru_n2	2
/stim/m1/thru_n3	1
/stim/m1/data_in	1
/stim/m1/rc_n1	0000000000000000
/stim/m1/rc_n2	0000000000000000
/stim/m1/rc_n3	00000000000000001
/stim/m1/data_out	0
/stim/m1/d_out1	1
/stim/m1/d_out2	0
/stim/m1/d_out3	0
/stim/m1/d_out4	0
/stim/m1/Output1	1
/stim/m1/Output2	0
/stim/m1/Output3	0
/stim/m1/rd_en1	St0
/stim/m1/rd_en2	St0
/stim/m1/rd_en3	St0
/stim/m1/wr_en1	St0
/stim/m1/wr_en2	St0
/stim/m1/wr_en3	St0
/stim/m1/wr_en4	St0
/stim/m1/pr_en1	St0
/stim/m1/pr_en2	St0
/stim/m1/pr_en3	St0
/stim/m1/pr_done1	St0
/stim/m1/pr_done2	St1
/stim/m1/pr_done3	St1
/stim/m1/diff_ptr1	0
/stim/m1/diff_ptr2	0
/stim/m1/diff_ptr3	0
/stim/m1/diff_ptr4	0
Now	300 ns
Cursor 1	45 ns

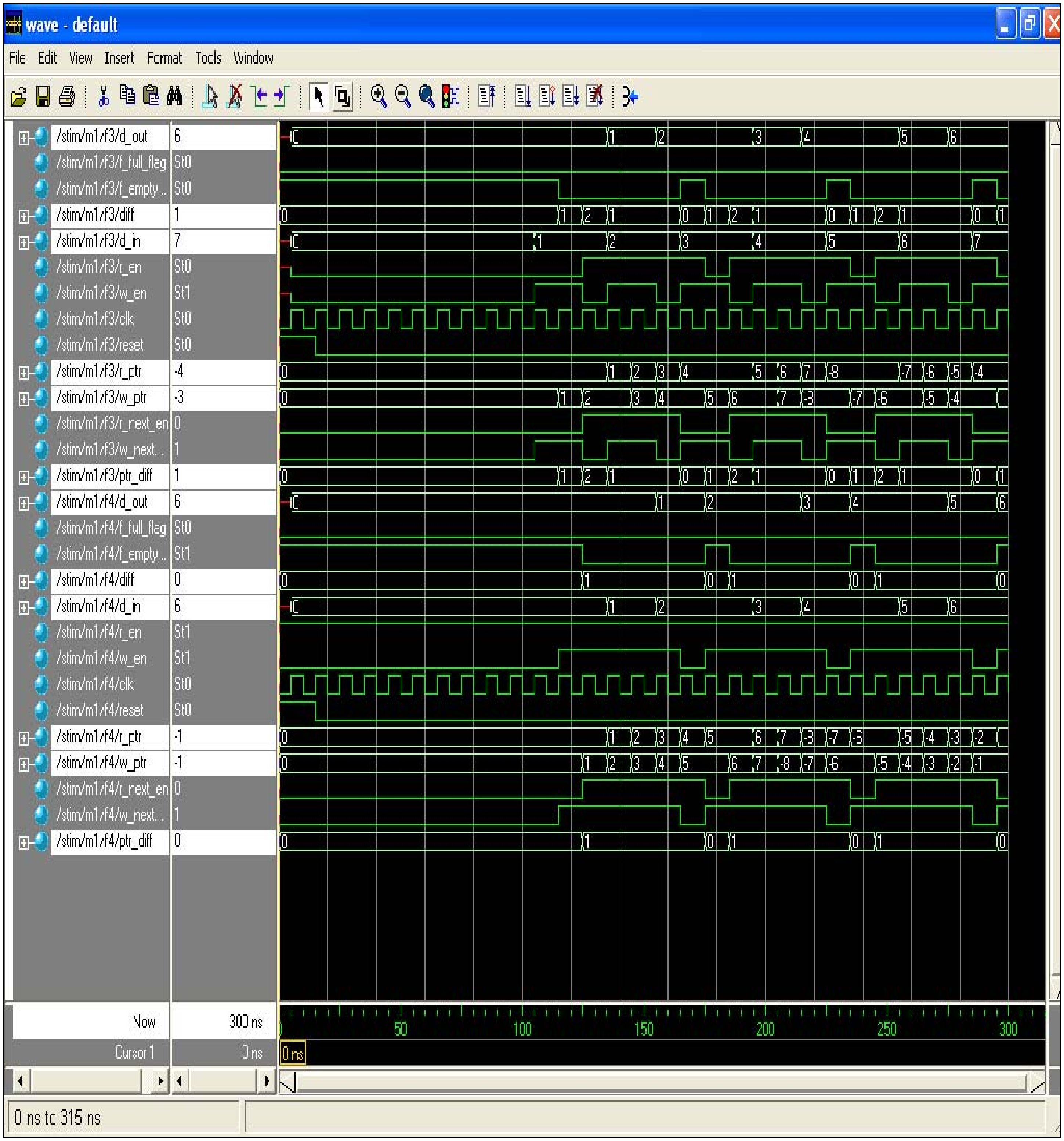
0 ns to 315 ns

FIFO (1_2) BEHAVIOR



Signal	Value
/stim/m1/f1/d_out	1
/stim/m1/f1/f_full_flag	St0
/stim/m1/f1/f_empty...	St0
/stim/m1/f1/diff	1
/stim/m1/f1/d_in	2
/stim/m1/f1/r_en	St0
/stim/m1/f1/w_en	St0
/stim/m1/f1/clk	St1
/stim/m1/f1/reset	St0
/stim/m1/f1/r_ptr	1
/stim/m1/f1/w_ptr	2
/stim/m1/f1/r_next_en	0
/stim/m1/f1/w_next...	0
/stim/m1/f1/ptr_diff	1
/stim/m1/f2/d_out	0
/stim/m1/f2/f_full_flag	St0
/stim/m1/f2/f_empty...	St1
/stim/m1/f2/diff	0
/stim/m1/f2/d_in	1
/stim/m1/f2/r_en	St0
/stim/m1/f2/w_en	St0
/stim/m1/f2/clk	St1
/stim/m1/f2/reset	St0
/stim/m1/f2/r_ptr	0
/stim/m1/f2/w_ptr	0
/stim/m1/f2/r_next_en	0
/stim/m1/f2/w_next...	0
/stim/m1/f2/ptr_diff	0

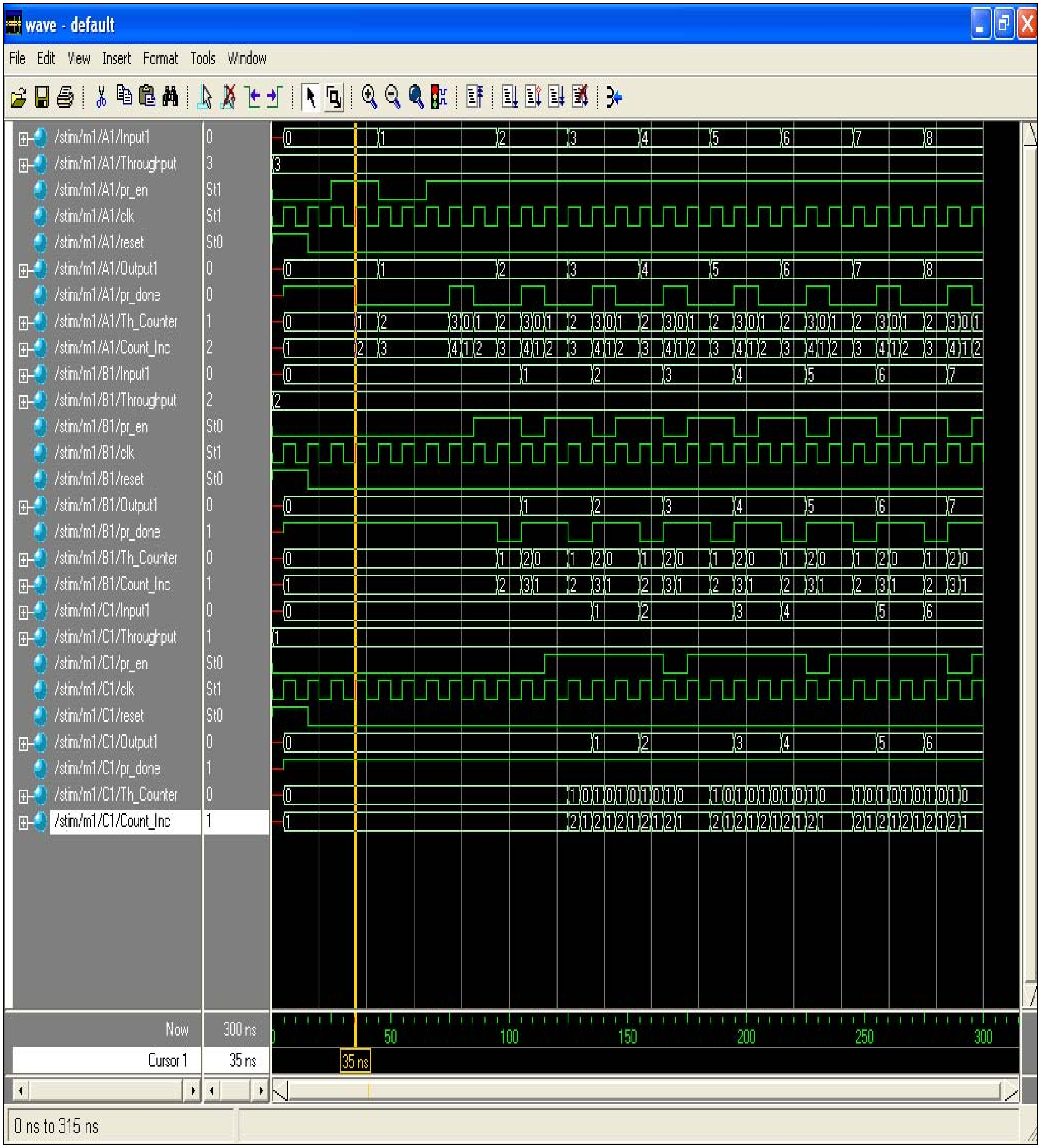
FIFO (3_4) BEHAVIOR



/stim/m1/f3/d_out	6
/stim/m1/f3/r_full_flag	St0
/stim/m1/f3/r_empty...	St0
/stim/m1/f3/diff	1
/stim/m1/f3/d_in	7
/stim/m1/f3/r_en	St0
/stim/m1/f3/w_en	St1
/stim/m1/f3/clk	St0
/stim/m1/f3/reset	St0
/stim/m1/f3/r_ptr	-4
/stim/m1/f3/w_ptr	-3
/stim/m1/f3/r_next_en	0
/stim/m1/f3/w_next...	1
/stim/m1/f3/ptr_diff	1
/stim/m1/f4/d_out	6
/stim/m1/f4/r_full_flag	St0
/stim/m1/f4/r_empty...	St1
/stim/m1/f4/diff	0
/stim/m1/f4/d_in	6
/stim/m1/f4/r_en	St1
/stim/m1/f4/w_en	St1
/stim/m1/f4/clk	St0
/stim/m1/f4/reset	St0
/stim/m1/f4/r_ptr	-1
/stim/m1/f4/w_ptr	-1
/stim/m1/f4/r_next_en	0
/stim/m1/f4/w_next...	1
/stim/m1/f4/ptr_diff	0
Now	300 ns
Cursor 1	0 ns

0 ns to 315 ns

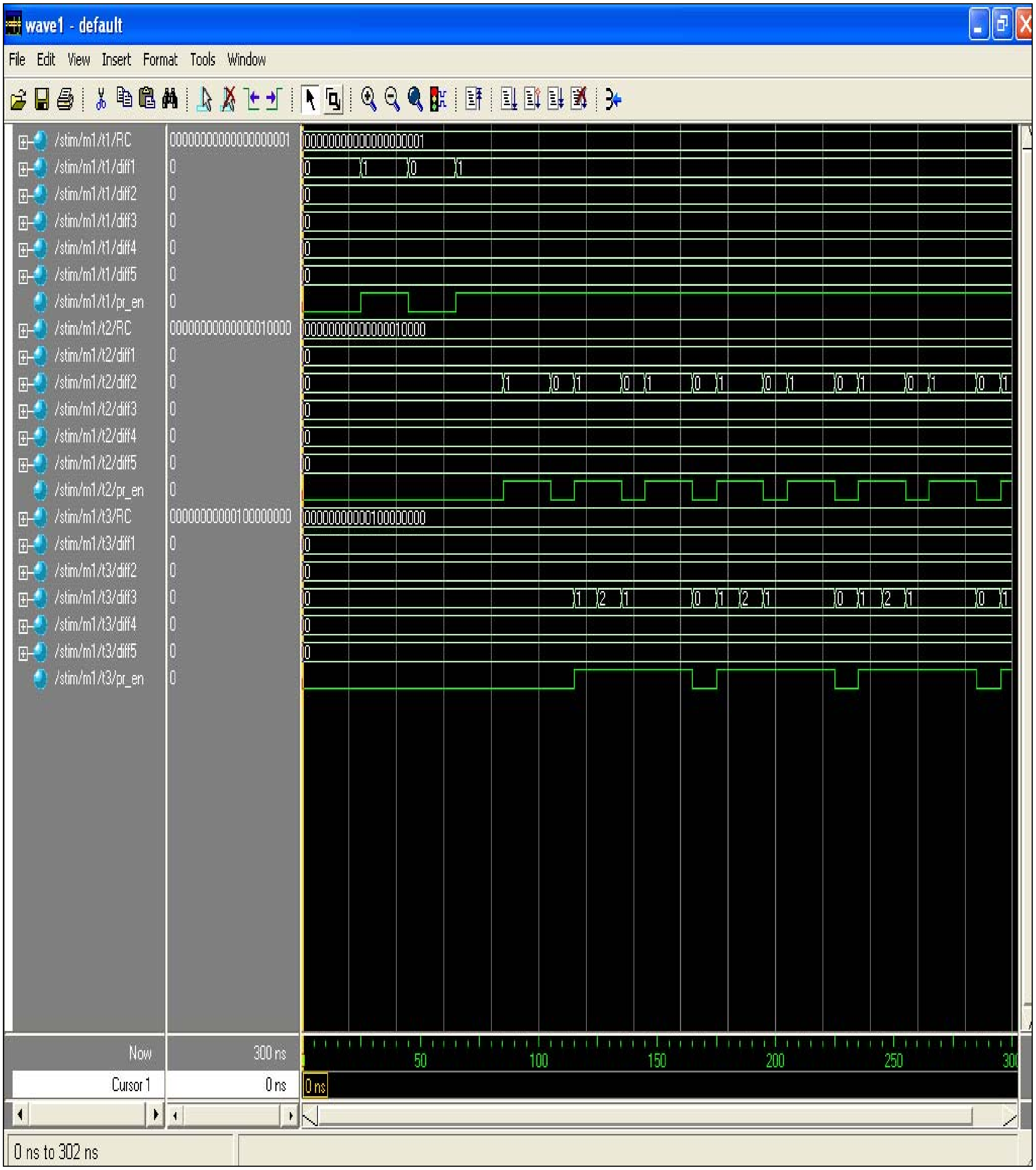
PROCESSES (A_B_C) BEHAVIOR



/stim/m1/A1/Input1	0
/stim/m1/A1/Throughput	3
/stim/m1/A1/pr_en	St1
/stim/m1/A1/clk	St1
/stim/m1/A1/reset	St0
/stim/m1/A1/Output1	0
/stim/m1/A1/pr_done	0
/stim/m1/A1/Th_Counter	1
/stim/m1/A1/Count_Inc	2
/stim/m1/B1/Input1	0
/stim/m1/B1/Throughput	2
/stim/m1/B1/pr_en	St0
/stim/m1/B1/clk	St1
/stim/m1/B1/reset	St0
/stim/m1/B1/Output1	0
/stim/m1/B1/pr_done	1
/stim/m1/B1/Th_Counter	0
/stim/m1/B1/Count_Inc	1
/stim/m1/C1/Input1	0
/stim/m1/C1/Throughput	1
/stim/m1/C1/pr_en	St0
/stim/m1/C1/clk	St1
/stim/m1/C1/reset	St0
/stim/m1/C1/Output1	0
/stim/m1/C1/pr_done	1
/stim/m1/C1/Th_Counter	0
/stim/m1/C1/Count_Inc	1
Now	300 ns
Cursor 1	35 ns

0 ns to 315 ns

TEST_RC_RP FOR PROCESS_A, B, C



```
/stim/m1/t1/RC      00000000000000000001
/stim/m1/t1/diff1   0
/stim/m1/t1/diff2   0
/stim/m1/t1/diff3   0
/stim/m1/t1/diff4   0
/stim/m1/t1/diff5   0
/stim/m1/t1/pr_en   0
/stim/m1/t2/RC      00000000000000010000
/stim/m1/t2/diff1   0
/stim/m1/t2/diff2   0
/stim/m1/t2/diff3   0
/stim/m1/t2/diff4   0
/stim/m1/t2/diff5   0
/stim/m1/t2/pr_en   0
/stim/m1/t3/RC      00000000000100000000
/stim/m1/t3/diff1   0
/stim/m1/t3/diff2   0
/stim/m1/t3/diff3   0
/stim/m1/t3/diff4   0
/stim/m1/t3/diff5   0
/stim/m1/t3/pr_en   0
```

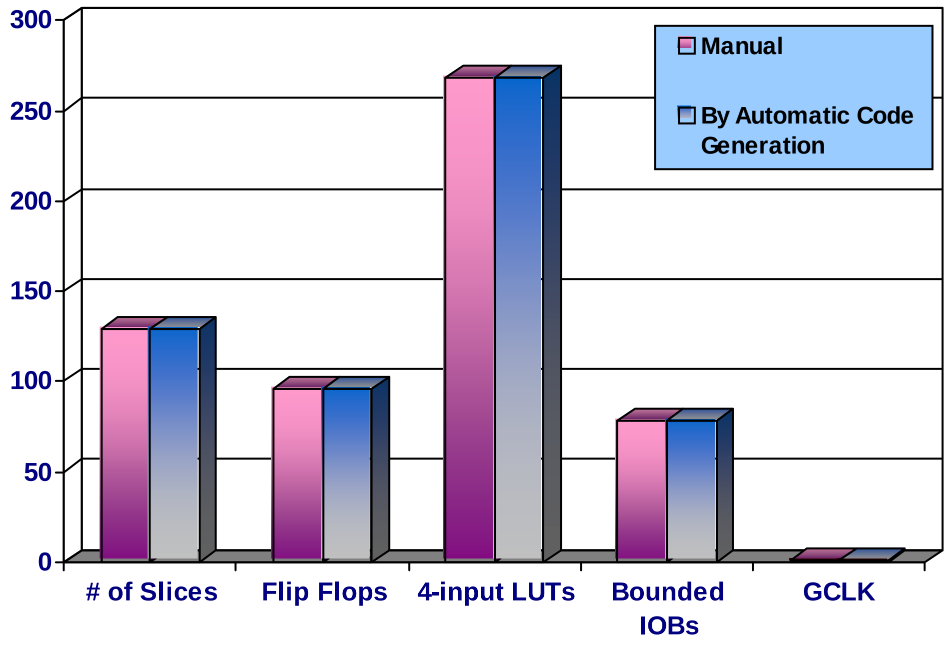
SYNTHESIS RESULTS

RTL CODE SYNTHESIS ON FPGA (SPARTAN 3E)

DEVICE UTILIZATION SUMMARY

Device Utilization Summary (estimated values)			[-]
Logic Utilization	***Used***	***Available***	***Utilization***
Number of Slices	*129*	*768*	*16%*
Number of Slice Flip Flops	*96*	*1536*	*6%*
Number of 4 input LUTs	*268*	*1536*	*17%*
Number of bonded IOBs	*78*	*100*	*78%*
Number of GCLKs	*1*	*8*	*12%*

COMPARISON OF MANUAL AND COMPILER GENERATED RTL CODES ON FPGA (SPARTAN 3E)



EXAMPLE 2

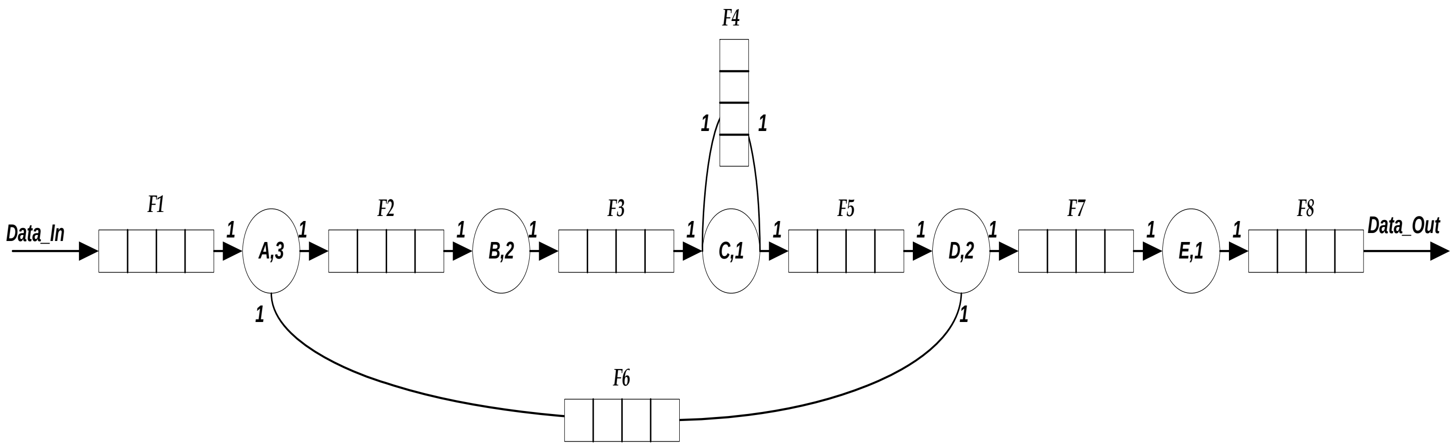


Fig. 2: Example 2

Description:

This DFG shows that there are five processing elements A, B, C, D and F. These processing elements are taking 3, 2, 1, 2 and 1 clock cycles for their execution. Also, these processing nodes will execute only as long as it acquires sufficient number of data tokens on their input buffers. There are eight FIFO buffers for temporarily holding data values. Also, nodes are listed with their rate of consumption and production parameters. Rate of consumption is defining the number of data tokens sufficient for node to process and rate of production parameter defines the number of data values that are produced by the respective processing element.

Configuration File Specifications

Nodes=5;

FIFO_Buffers=8;

Token_Size=8;

Topology_matrix =   [1 -1 0 0 0 0 0;0 1 -1 0 0 0 0;0 0 1 -1 0 0 0;

0 0 0 1 -1 0 0;0 -1 0 0 1 0 0;0 0 0 0 1 -1 0;

0 0 0 0 0 1 -1];

Self_Loops=[0 0 1 0 0];

Algo_Delays=[0 0 0 0 0 0 0 0];

Data_In=1:20;

Throughput_node1=3;

Throughput_node2=2;

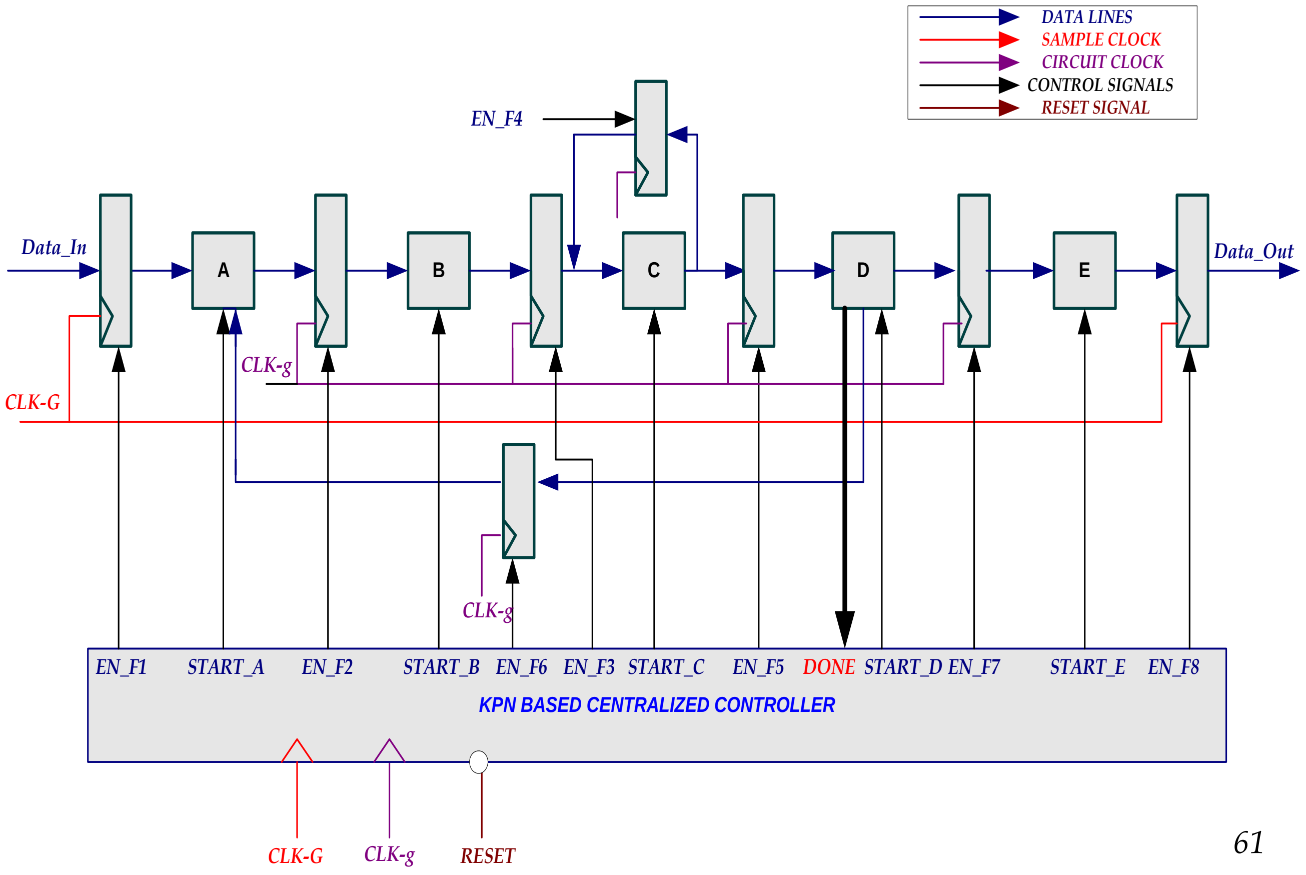
Throughput_node3=1;

Throughput_node4=2;

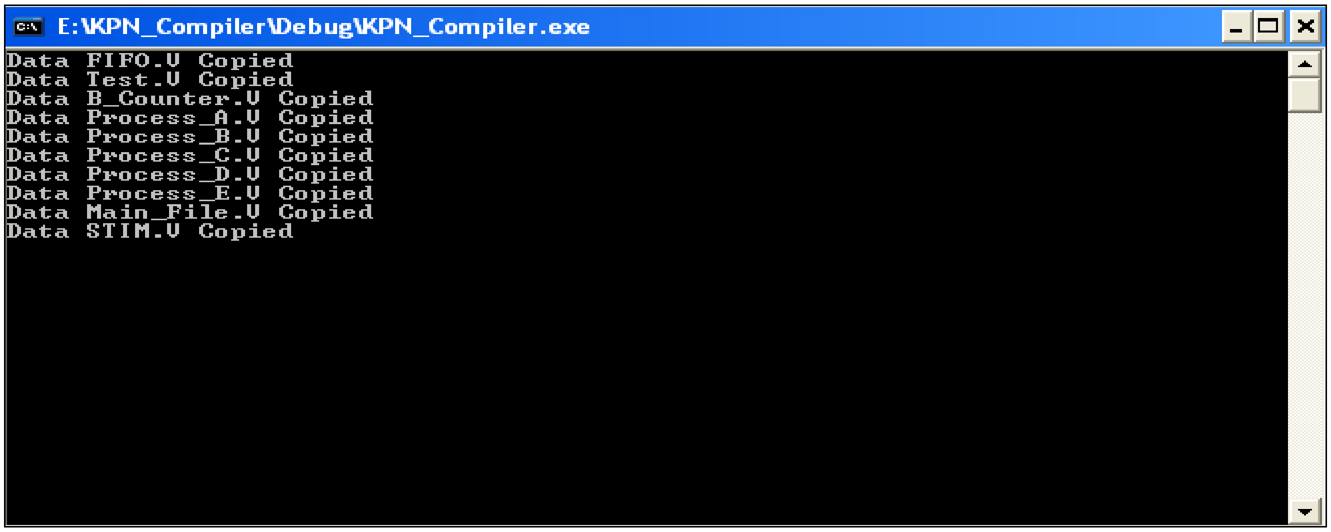
Throughput_node5=1;

Based on these specifications in MATLAB, C-based configuration file is generated that will be passed to C-based designed compiler. Then, this compiler will automatically generate synthesized HDL code of this DSP design.

<u>KPN Based Centralized Controller</u>

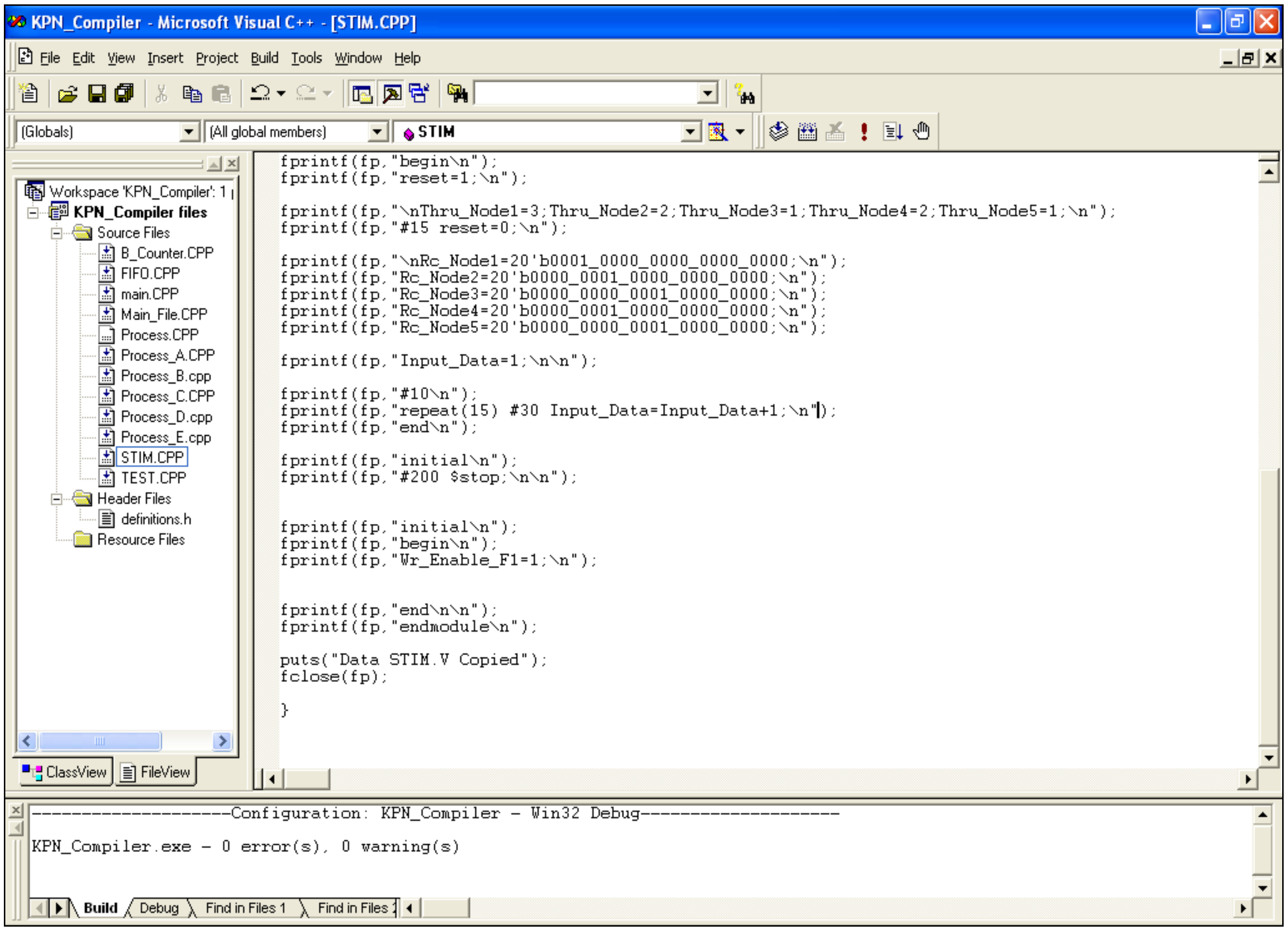


COMPILER OUTPUT



```
Data FIFO.V Copied
Data Test.V Copied
Data B_Counter.V Copied
Data Process_A.V Copied
Data Process_B.V Copied
Data Process_C.V Copied
Data Process_D.V Copied
Data Process_E.V Copied
Data Main_File.V Copied
Data STIM.V Copied
```

CODE GENERATION TOOL SNAPSHOT



```
fprintf(fp,"begin\n");
fprintf(fp,"reset=1;\n");

fprintf(fp,"\nThru_Node1=3;Thru_Node2=2;Thru_Node3=1;Thru_Node4=2;Thru_Node5=1;\n");
fprintf(fp,"#15 reset=0;\n");

fprintf(fp,"\nRc_Node1=20'b0001_0000_0000_0000_0000;\n");
fprintf(fp,"Rc_Node2=20'b0000_0001_0000_0000_0000;\n");
fprintf(fp,"Rc_Node3=20'b0000_0000_0001_0000_0000;\n");
fprintf(fp,"Rc_Node4=20'b0000_0001_0000_0000_0000;\n");
fprintf(fp,"Rc_Node5=20'b0000_0000_0001_0000_0000;\n");

fprintf(fp,"Input_Data=1;\n\n");

fprintf(fp,"#10\n");
fprintf(fp,"repeat(15) #30 Input_Data=Input_Data+1;\n");
fprintf(fp,"end\n");

fprintf(fp,"initial\n");
fprintf(fp,"#200 $stop;\n\n");


fprintf(fp,"initial\n");
fprintf(fp,"begin\n");
fprintf(fp,"Wr_Enable_F1=1;\n");


fprintf(fp,"end\n\n");
fprintf(fp,"endmodule\n");

puts("Data STIM.V Copied");
fclose(fp);

}
```

```
--------------------Configuration: KPN_Compiler - Win32 Debug--------------------

KPN_Compiler.exe - 0 error(s), 0 warning(s)
```

Verilog Source Code of given DSP Design generated by C-Compiler:

1. **_KPN_CONTROLLER.V_**

```
module main_File(wr_en1,rc_n1,rc_n2,rc_n3,rc_n4,rc_n5,data_in,
         data_out,thru_n1,thru_n2,thru_n3,thru_n4,
         thru_n5,clk,reset);


//Here i assumed the token size to be 08 bits
//Input Signals
input clk,reset;


//Write enable signal for fifo_1
input wr_en1;


//Throughputs for each node
input [3:0]thru_n1,thru_n2,thru_n3,thru_n4,thru_n5;


//Input Stream data will bestored at here
input [7:0]data_in;


//Rate of consumption at each link
input [23:0]rc_n1,rc_n2,rc_n3,rc_n4,rc_n5;


//Output Data
```

```
output [7:0]data_out;

reg [7:0]data_out;


//Temporary wires and Registers holding Outputs from FIFOs and processes

//and holding control signals as well

wire [7:0]d_out1,d_out2,d_out3,d_out4,

        d_out5,d_out6,d_out7,d_out8;


wire [7:0]Output1,Output2,Output3,Output4,

        Output5,Output6,Output7;


wire rd_en1,rd_en2,rd_en3,rd_en4,wr_en2,wr_en3,wr_en4;


reg read1,read2,read3,read4,read5,read6,read7,read8,

   write2,write3,write4,write5,write6,write7,write8;


//Flags showing FIFO Status i.e (Fifo Full and Fifo Empty)

wire f_full_flag1,f_full_flag2,f_full_flag3,f_full_flag4,

    f_full_flag5,f_full_flag6,f_full_flag7,f_full_flag8,

    f_empty_flag1,f_empty_flag2,f_empty_flag3,f_empty_flag4,

    f_empty_flag5,f_empty_flag6,f_empty_flag7,f_empty_flag8;


//Enable the process and Done Signals showing process has completed its execution

//and ready to write data at ints output buffer
```

```
wire pr_en1,pr_en2,pr_en3,pr_en4,pr_en5,
    pr_done1,pr_done2,pr_done3,pr_done4,pr_done5;

//Pointer that stores the Difference between w_ptr and r_ptr
wire [3:0]diff_ptr1,diff_ptr2,diff_ptr3,diff_ptr4,
        diff_ptr5,diff_ptr6,diff_ptr7,diff_ptr8;

//Call FIFO Instances
fifo f1(diff_ptr1,d_out1,f_full_flag1,f_empty_flag1,data_in,rd_en1,wr_en1,clk,reset);
fifo f2(diff_ptr2,d_out2,f_full_flag2,f_empty_flag2,Output1,rd_en2,wr_en2,clk,reset);
fifo f3(diff_ptr3,d_out3,f_full_flag3,f_empty_flag3,Output2,rd_en3,wr_en3,clk,reset);
fifo f4(diff_ptr4,d_out4,f_full_flag4,f_empty_flag4,Output3,rd_en4,wr_en4,clk,reset);
fifo f5(diff_ptr5,d_out5,f_full_flag5,f_empty_flag5,Output4,rd_en5,wr_en5,clk,reset);
fifo f6(diff_ptr6,d_out6,f_full_flag6,f_empty_flag6,Output5,rd_en6,wr_en6,clk,reset);
fifo f7(diff_ptr7,d_out7,f_full_flag7,f_empty_flag7,Output6,rd_en7,wr_en7,clk,reset);
fifo f8(diff_ptr8,d_out8,f_full_flag8,f_empty_flag8,Output7,rd_en8,wr_en8,clk,reset);

//Process Instantiations
ProcessA A1(clk,reset,pr_en1,d_out1,d_out6,Output1,thru_n1,pr_done1);
ProcessB B1(clk,reset,pr_en2,d_out2,Output2,thru_n2,pr_done2);
ProcessC C1(clk,reset,pr_en3,d_out3,d_out4,Output3,Output4,thru_n3,pr_done3);
ProcessD D1(clk,reset,pr_en4,d_out5,Output5,Output6,thru_n4,pr_done4);
ProcessE E1(clk,reset,pr_en5,d_outd_out7,Output7,thru_n5,pr_done5);
```

```
//Test Module checking sufficient tokens have acquired in each fifo to start execution
test t1(rc_n1,diff_ptr1,4'b0,4'b0,4'b0,diff_ptr6,4'b0,pr_en1);
test t2(rc_n2,4'b0,diff_ptr2,4'b0,4'b0,4'b0,4'b0,pr_en2);
test t3(rc_n3,4'b0,4'b0,diff_ptr3,diff_ptr4,4'b0,4'b0,pr_en3);
test t4(rc_n4,4'b0,4'b0,4'b0,diff_ptr5,4'b0,4'b0,pr_en4);
test t5(rc_n5,4'b0,4'b0,4'b0,4'b0,diff_ptr7,4'b0,pr_en5);

//Simple assignments
//assign data_out=d_out4;
assign rd_en1=read1;
assign rd_en2=read2;
assign rd_en3=read3;
assign rd_en4=read4;
assign rd_en5=read5;
assign rd_en6=read6;
assign rd_en7=read7;
assign rd_en8=1;

assign wr_en2=write2;
assign wr_en3=write3;
assign wr_en4=write4;
assign wr_en5=write5;
assign wr_en6=write6;
assign wr_en7=write7;
assign wr_en8=write8;


//Activate final read signal
always @(diff_ptr8)
if(diff_ptr8>=1)
        read8=1;
else
        read8=0;


//When Pr_en1 is active high or low
always @(posedge clk)
if(pr_en1)
        begin if(pr_done1)
                begin read1=1;read6=1;end
        else
                begin read1=0;read6=0;end
        end
else
        begin read1=0;read6=0;end


//When pr_en2 is active high or low
always @(posedge clk)
if(pr_en2)
        begin if(pr_done2)
```

```
                begin read2=1;end
        else
                begin read2=0;end
        end
else
        read2=0;

//When pr_en3 is active high or low
always @(posedge clk)
if(pr_en3)
        begin if(pr_done3)
                begin read3=1;read4=1;end
        else
                begin read4=0;read3=0;end
        end
else
        begin read3=0;read4=0;end

//When pr_en4 is active high or low
always @(posedge clk)
if(pr_en4)
      begin if(pr_done4)
                begin read5=1;end
        else
```

```
                    begin read5=0;end
            end
else
            read5=0;

//When pr_en5 is active high or low
always @(posedge clk)
if(pr_en5)
            begin if(pr_done5)
                    begin read7=1;end
            else
                    begin read7=0;end
            end
else
            read7=0;



///////////////Enabling Write Pointer of FIFOs after Process's execution
//When pr_done1 is active high or low
always @(pr_done1)
if(pr_done1)
            if(pr_en1)
            write2=1;
            else
```

```
        write2=0;
else
        write2=0;

//When pr_done2 is active high or low
always @(pr_done2)
if(pr_done2)
        if(pr_en2)
        write3=1;
        else
        write3=0;
else
        write3=0;

//When pr_done3 is active high or low
always @(pr_done3)
if(pr_done3)
        if(pr_en3)
        begin write4=1;write5=1;end
        else
        begin write4=0;write5=0;end
else
        begin write4=0;write5=0;end
```

```
//When pr_done4 is active high or low
always @(pr_done4)
if(pr_done4)
        if(pr_en4)
        begin write6=1;write7=1;end
        else
        begin write6=0;write7=0;end
else
        begin write6=0;write7=0;end

//When pr_done5 is active high or low
always @(pr_done5)
if(pr_done5)
        if(pr_en5)
        write8=1;
        else
        write8=0;
else
        write8=0;

/////Writing in Output Buffer at each posedge of sample clock
always @(posedge clk)
begin
        if(reset)
```

```
          begin
                    data_out<=0;
          end
          else
          begin
                    data_out<=d_out8;
          end
end
endmodule
```

2. FIFO.V

```
//================================================
//fifo.v; verilog code for asynchronous FIFO
//This module describes FIFO
//===============================================

module fifo(diff,d_out,f_full_flag,f_empty_flag,d_in,r_en,w_en,clk,reset);
parameter width=8; //FIFO width
parameter f_depth=16; //FIFO depth
parameter f_ptr_width=4; //because depth =16;

output [width-1:0] d_out; reg [width-1:0] d_out; //outputs
output f_full_flag,f_empty_flag;
output [3:0]diff;
```

```
input [width-1:0] d_in;

input r_en,w_en,clk;

input reset;


//internal registers,wires

wire [f_ptr_width-1:0] r_ptr,w_ptr;

reg r_next_en,w_next_en;

reg [f_ptr_width-1:0] ptr_diff;

reg [width-1:0] f_memory[f_depth-1:0];


assign diff=ptr_diff;

assign f_full_flag=(ptr_diff==(f_depth-1)); //assign FIFO status

assign f_empty_flag=(ptr_diff==0);


//instantiate address counters


b_counter r_b_counter(.c_out(r_ptr),.c_reset(reset),.c_clk(clk),.en(r_next_en));

b_counter w_b_counter(.c_out(w_ptr),.c_reset(reset),.c_clk(clk),.en(w_next_en));



//-------------------------------------------------------

always @(posedge clk) //write to memory

begin
```

```
if(reset)
        d_out<=0; //f_memory[r_ptr];
if(w_en)
begin
                if(!f_full_flag)
                f_memory[w_ptr]<=d_in;
            end
if(r_en)
begin
                if(!f_empty_flag)
                d_out<=f_memory[r_ptr];
            end


end
//-----------------------------------------------------
always @(*) //ptr_diff changes as clock changes
begin
        if(w_ptr>r_ptr)
                ptr_diff<=w_ptr-r_ptr;
        else if(w_ptr<r_ptr)
                ptr_diff<=((f_depth-r_ptr)+w_ptr);
        else ptr_diff<=0;
end
//-----------------------------------------------------
```

```
always @(*) //after empty flag activated fifo read counter should not increment;
begin
        if(r_en && (!f_empty_flag))
                r_next_en=1;
        else
                r_next_en=0;
end
//-------------------------------------------------------
always @(*) //after full flag activated fifo write counter should not increment;
begin
        if(w_en && (!f_full_flag))
                w_next_en=1;
        else
                w_next_en=0;
end
endmodule
```

3. <u>B_COUNTER.V</u>

```
//=================================================
//b_counter.v; 4 bit asynchronous binary up counter
//=================================================
module b_counter(c_out,c_reset,c_clk,en);

parameter c_width=4; //counter width
```

```
output [c_width-1:0] c_out; reg [c_width-1:0] c_out;
input c_reset,c_clk,en;

always @(posedge c_clk or posedge c_reset)
        if (c_reset)
                c_out <= 0;
        else if(en)
                c_out <= c_out + 1;
endmodule

//==============================================
```

4. PROCESS_A.V

```
module ProcessA(clk,reset,pr_en,Input1,Input2,Output1,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1,Input2;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;
```

```
//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1+Input2;
//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
            if(reset)
            begin
                        pr_done<=1;
                        Th_Counter<=0;
            end
            else
            begin
                if(pr_en)
                        Th_Counter=Count_Inc;
                else
                        Th_Counter=Th_Counter;//Do Nothing
            end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
```

```
            pr_done=1;
            #5 Th_Counter=0;
end
else
            if(pr_en)
            pr_done=0;
            else pr_done=1;
endmodule
```

5. *PROCESS_B.V*

```
module ProcessB(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;

//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
```

```
assign Output1=Input1;
//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
```

```
else
        if(pr_en)
        pr_done=0;
        else pr_done=1;
endmodule
```

6. ***PROCESS_C.V***

```
module ProcessC(clk,reset,pr_en,Input1,Input2,Output1,Output2,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1,Input2;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1,Output2;
output pr_done;
reg pr_done;

//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1;
assign Output2=Input2;
//Simply Waste Clock Cycles for process internal algorithm execution
```

```
always @(posedge clk)
begin
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
else
        if(pr_en)
```

```
          pr_done=0;

          else pr_done=1;

endmodule
```

7. <u>PROCESS_D.V</u>

```
module ProcessD(clk,reset,pr_en,Input1,Output1,Output2,Throughput,pr_done);

parameter width=8; //FIFO width


input [width-1:0]Input1;

input [3:0]Throughput;

input pr_en,clk,reset;


output [width-1:0]Output1,Output2;

output pr_done;

reg pr_done;


//Temporary Registers and wires

reg [3:0]Th_Counter;

wire [3:0]Count_Inc=Th_Counter+1;

assign Output1=Input1;

assign Output2=Input1;

//Simply Waste Clock Cycles for process internal algorithm execution

always @(posedge clk)

begin
```

```
        if(reset)
        begin
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
else
        if(pr_en)
        pr_done=0;
        else pr_done=1;
```

```
endmodule
```

8. PROCESS_E.V

```
module ProcessE(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);
parameter width=8; //FIFO width

input [width-1:0]Input1;
input [3:0]Throughput;
input pr_en,clk,reset;

output [width-1:0]Output1;
output pr_done;
reg pr_done;

//Temporary Registers and wires
reg [3:0]Th_Counter;
wire [3:0]Count_Inc=Th_Counter+1;
assign Output1=Input1;
//Simply Waste Clock Cycles for process internal algorithm execution
always @(posedge clk)
begin
        if(reset)
        begin
```

```
                pr_done<=1;
                Th_Counter<=0;
        end
        else
        begin
           if(pr_en)
                Th_Counter=Count_Inc;
           else
                Th_Counter=Th_Counter;//Do Nothing
        end
end

always @(Th_Counter)
if(Th_Counter == Throughput)
begin
        pr_done=1;
        #5 Th_Counter=0;
end
else
        if(pr_en)
        pr_done=0;
        else pr_done=1;
endmodule
```

9. TEST_RC_RP.V

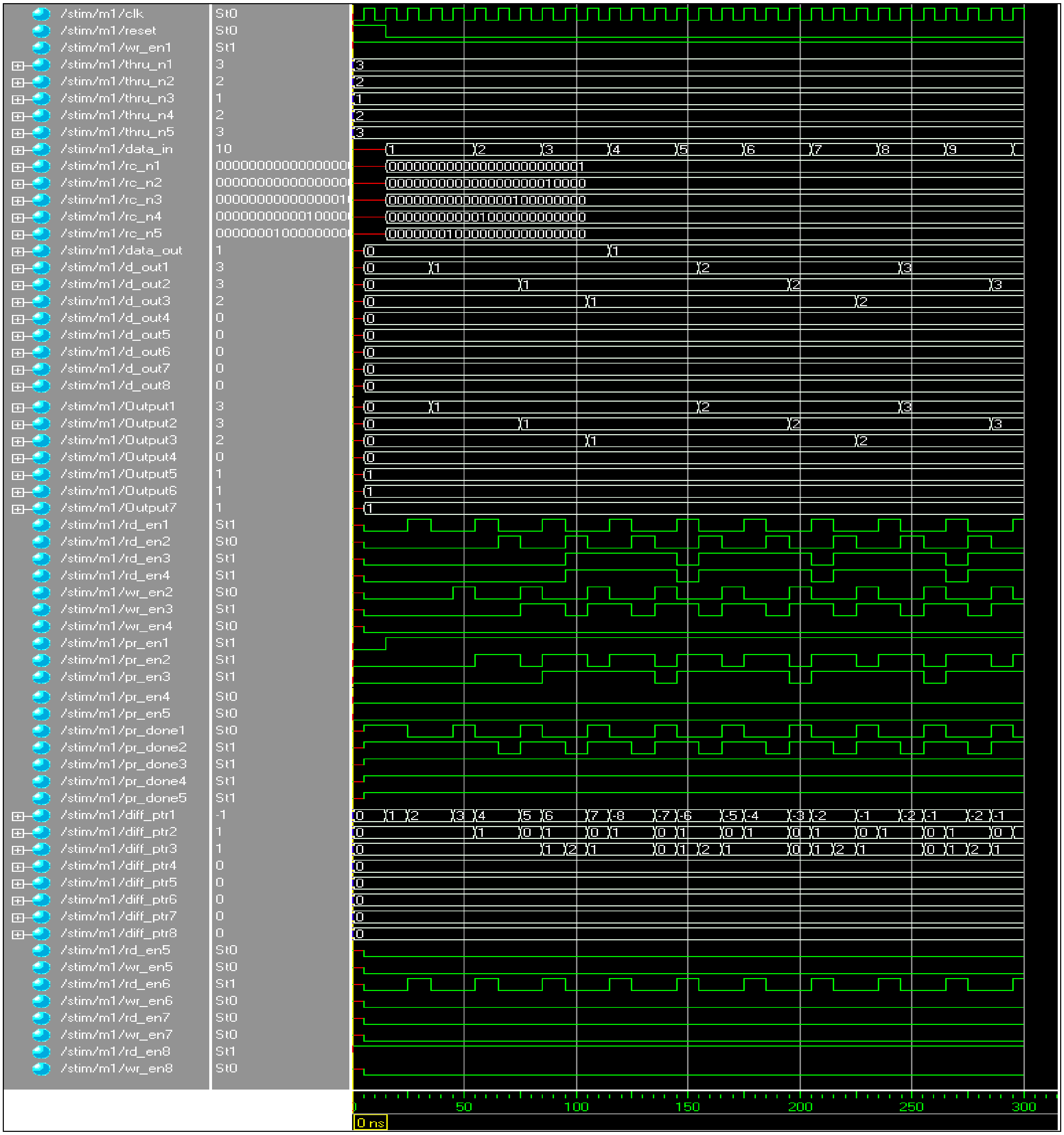
```
module test(RC,diff1,diff2,diff3,diff4,diff5,diff6,pr_en);

input [23:0]RC;

input [3:0]diff1,diff2,diff3,diff4,diff5,diff6;

output pr_en;

reg pr_en;

always @(diff1 or diff2 or diff3 or diff4 or diff5 or diff6 or RC)

if(diff1 >= RC[3:0] && diff2 >= RC[7:4] && diff3 >= RC[11:8] &&
diff4>=RC[15:12] &&

  diff5>=RC[19:16] && diff6 >= RC[23:20])

        begin

        pr_en=1;

        end

else

        begin

        pr_en=0;

        end

        endmodule
```

10. STIMULUS.V

```
module stim;

reg clk,reset;

reg wr_en1;
reg [3:0]thru_n1,thru_n2,thru_n3,thru_n4,thru_n5;

reg [7:0]data_in;

reg [23:0]rc_n1,rc_n2,rc_n3,rc_n4,rc_n5;

wire [7:0]data_out;

main_File m1(wr_en1,rc_n1,rc_n2,rc_n3,rc_n4,rc_n5,data_in,data_out,

              thru_n1,thru_n2,thru_n3,thru_n4,thru_n5,clk,reset);


initial

begin

clk=0;

forever

#5 clk=~clk;

end

initial

begin

reset=1;

thru_n1=3;thru_n2=2;thru_n3=1;thru_n4=2;thru_n5=3;

#15 reset=0;

rc_n1=24'b0000_0000_0000_0000_0000_0001;

rc_n2=24'b0000_0000_0000_0000_0001_0000;

rc_n3=24'b0000_0000_0000_0001_0000_0000;

rc_n4=24'b0000_0000_0001_0000_0000_0000;

rc_n5=24'b0000_0001_0000_0000_0000_0000;

data_in=1;
```

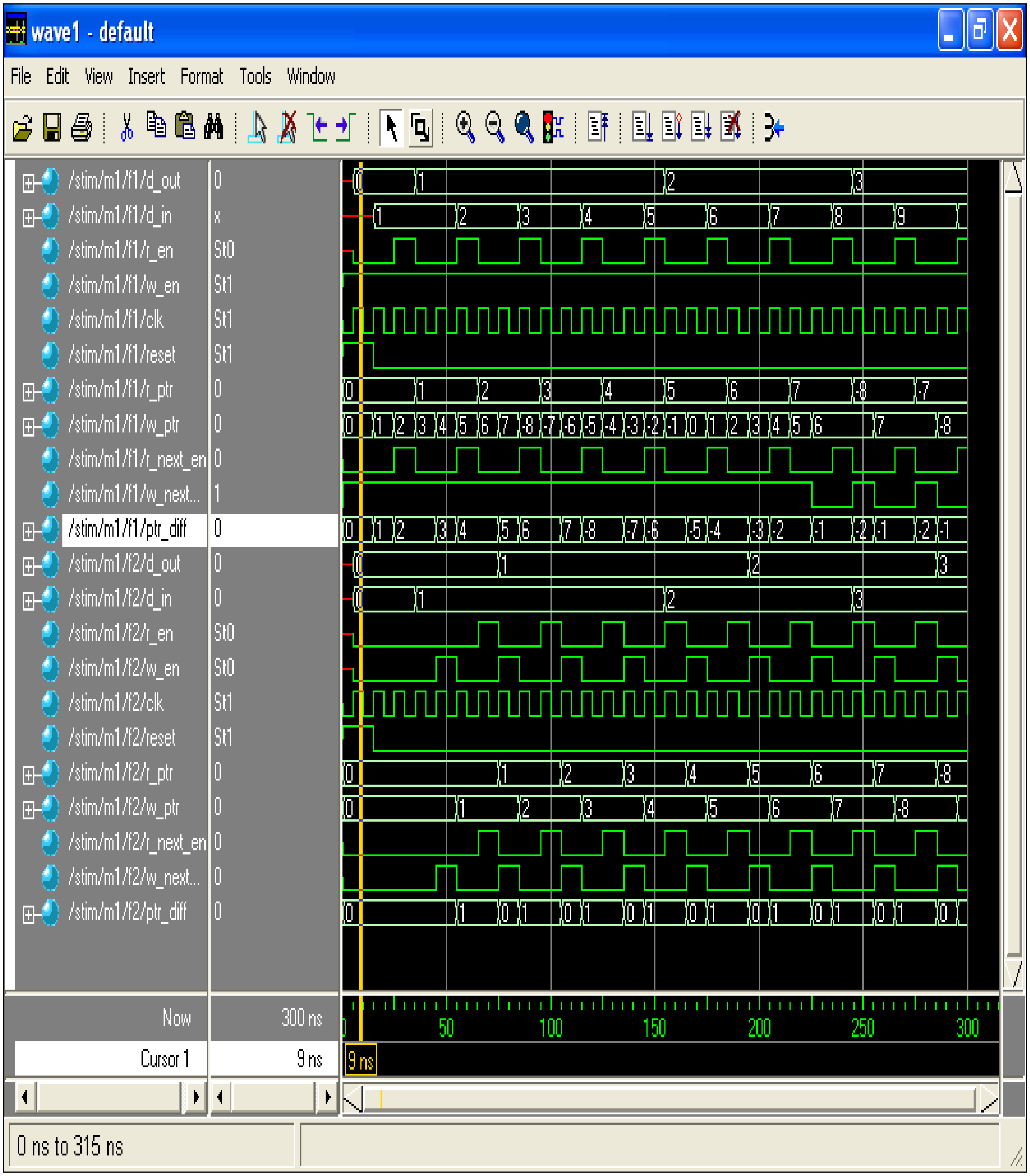
```
#10

repeat(15)

#30 data_in=data_in+1;

end

initial

#200 $stop;

initial

begin

wr_en1=1;

end

endmodule
```

SIMULATION RESULTS



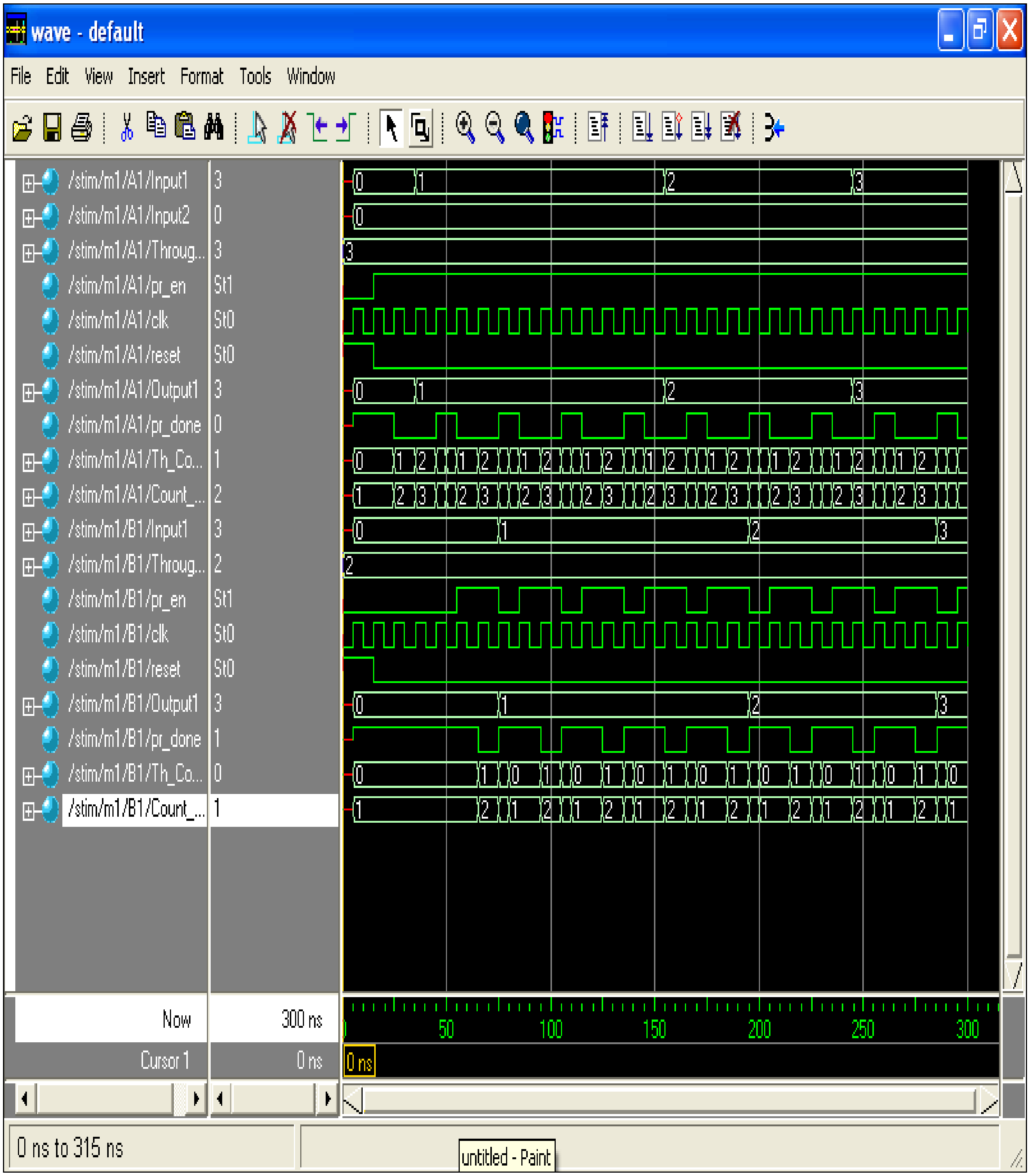
/stim/m1/clk	St0
/stim/m1/reset	St0
/stim/m1/wr_en1	St1
/stim/m1/thru_n1	3
/stim/m1/thru_n2	2
/stim/m1/thru_n3	1
/stim/m1/thru_n4	2
/stim/m1/thru_n5	3
/stim/m1/data_in	10
/stim/m1/rc_n1	0000000000000000
/stim/m1/rc_n2	0000000000000000
/stim/m1/rc_n3	0000000000000001
/stim/m1/rc_n4	0000000000010000
/stim/m1/rc_n5	0000000100000000
/stim/m1/data_out	1
/stim/m1/d_out1	3
/stim/m1/d_out2	3
/stim/m1/d_out3	2
/stim/m1/d_out4	0
/stim/m1/d_out5	0
/stim/m1/d_out6	0
/stim/m1/d_out7	0
/stim/m1/d_out8	0
/stim/m1/Output1	3
/stim/m1/Output2	3
/stim/m1/Output3	2
/stim/m1/Output4	0
/stim/m1/Output5	1
/stim/m1/Output6	1
/stim/m1/Output7	1
/stim/m1/rd_en1	St1
/stim/m1/rd_en2	St0
/stim/m1/rd_en3	St1
/stim/m1/rd_en4	St1
/stim/m1/wr_en2	St0
/stim/m1/wr_en3	St1
/stim/m1/wr_en4	St0
/stim/m1/pr_en1	St1
/stim/m1/pr_en2	St1
/stim/m1/pr_en3	St1
/stim/m1/pr_en4	St0
/stim/m1/pr_en5	St0
/stim/m1/pr_done1	St0
/stim/m1/pr_done2	St1
/stim/m1/pr_done3	St1
/stim/m1/pr_done4	St1
/stim/m1/pr_done5	St1
/stim/m1/diff_ptr1	-1
/stim/m1/diff_ptr2	1
/stim/m1/diff_ptr3	1
/stim/m1/diff_ptr4	0
/stim/m1/diff_ptr5	0
/stim/m1/diff_ptr6	0
/stim/m1/diff_ptr7	0
/stim/m1/diff_ptr8	0
/stim/m1/rd_en5	St0
/stim/m1/wr_en5	St0
/stim/m1/rd_en6	St1
/stim/m1/wr_en6	St0
/stim/m1/rd_en7	St0
/stim/m1/wr_en7	St0
/stim/m1/rd_en8	St1
/stim/m1/wr_en8	St0

FIFO(1_2) BEHAVIOR_



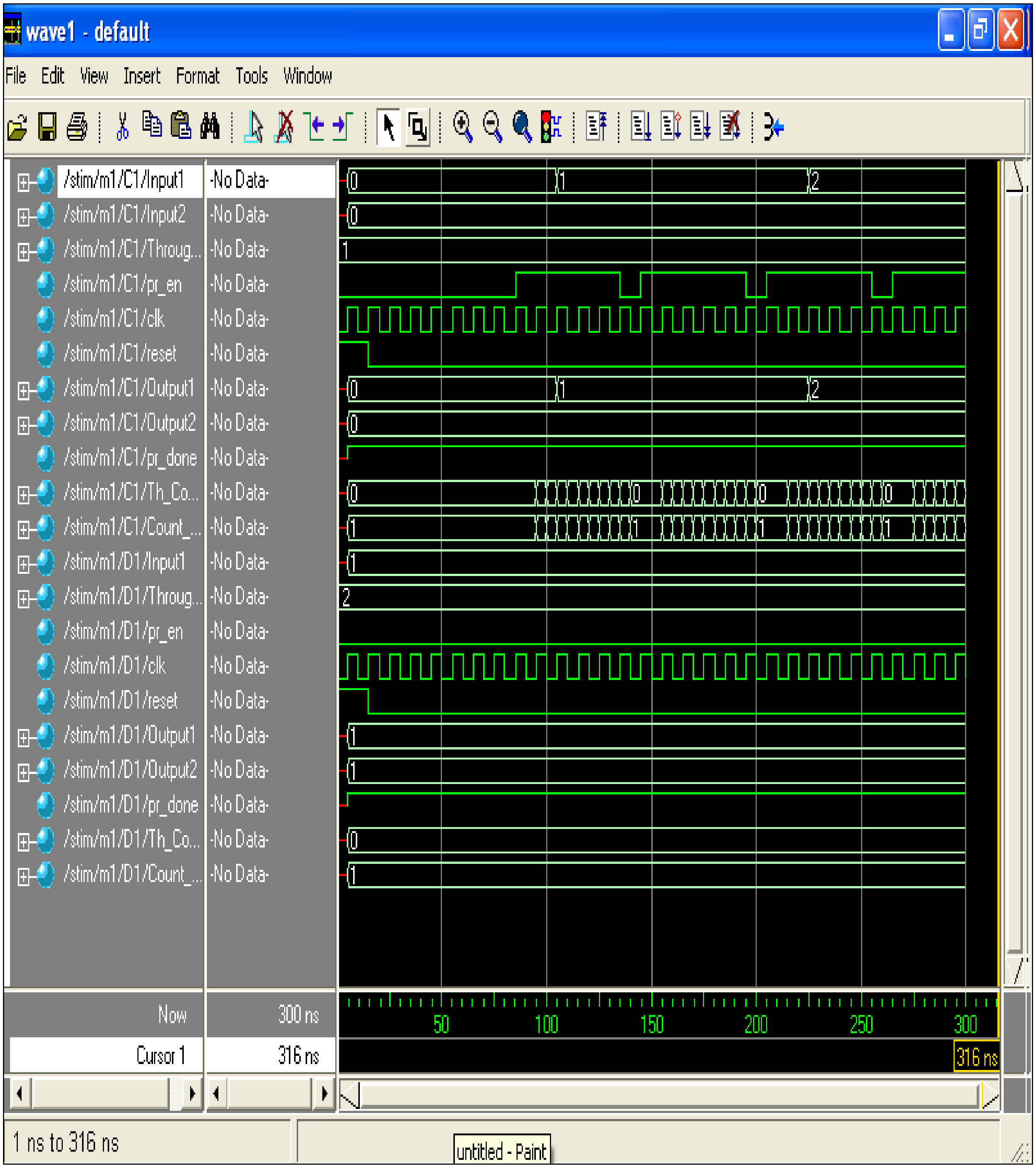
```
/stim/m1/f1/d_out        0
/stim/m1/f1/d_in         x
/stim/m1/f1/r_en         St0
/stim/m1/f1/w_en         St1
/stim/m1/f1/clk          St1
/stim/m1/f1/reset        St1
/stim/m1/f1/r_ptr        0
/stim/m1/f1/w_ptr        0
/stim/m1/f1/r_next_en    0
/stim/m1/f1/w_next...    1
/stim/m1/f1/ptr_diff     0
/stim/m1/f2/d_out        0
/stim/m1/f2/d_in         0
/stim/m1/f2/r_en         St0
/stim/m1/f2/w_en         St0
/stim/m1/f2/clk          St1
/stim/m1/f2/reset        St1
/stim/m1/f2/r_ptr        0
/stim/m1/f2/w_ptr        0
/stim/m1/f2/r_next_en    0
/stim/m1/f2/w_next...    0
/stim/m1/f2/ptr_diff     0

Now        300 ns
Cursor 1   9 ns

0 ns to 315 ns
```

PROCESS (A_B) BEHAVIOR



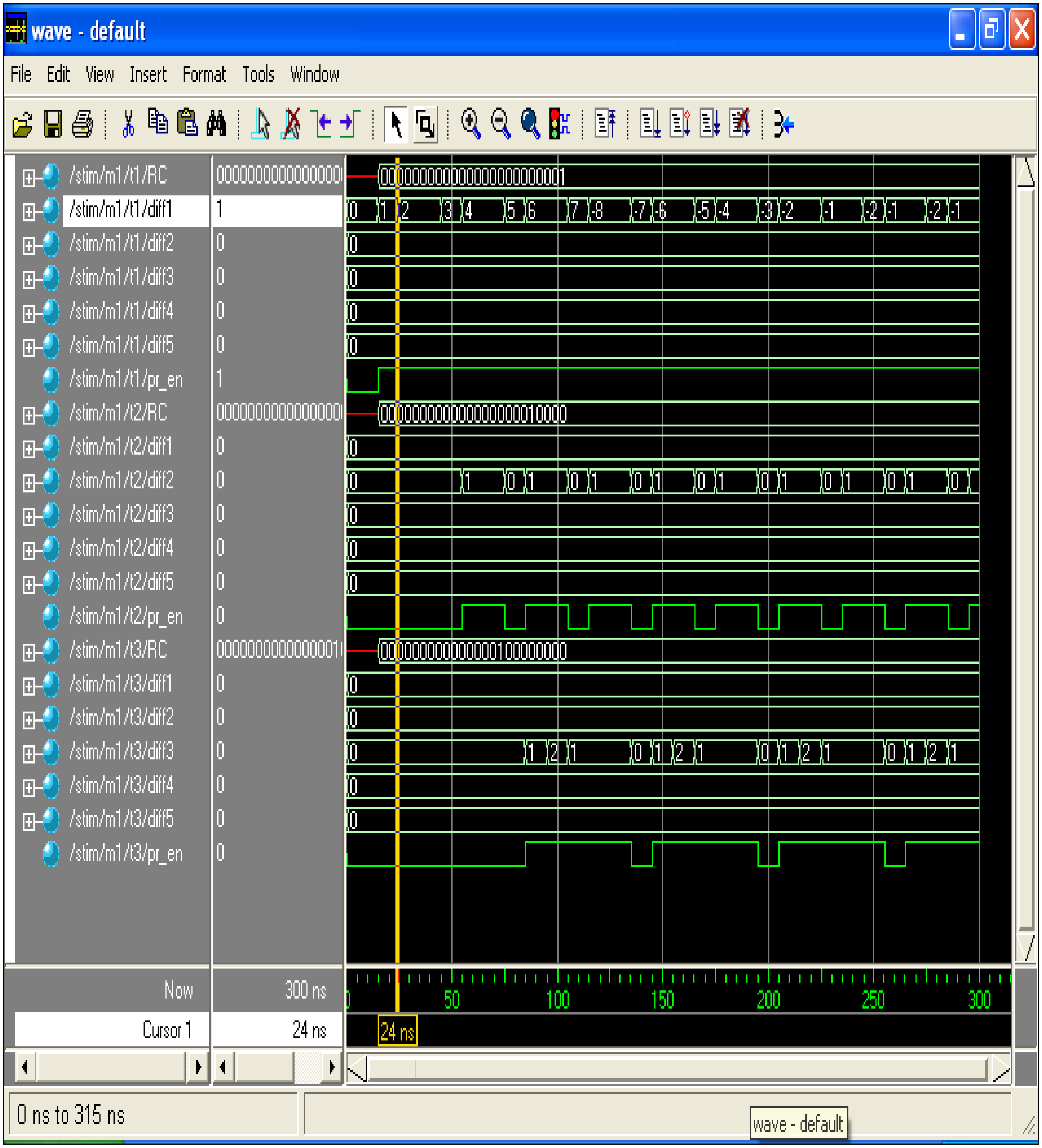
```
/stim/m1/A1/Input1     3
/stim/m1/A1/Input2     0
/stim/m1/A1/Throug...  3
/stim/m1/A1/pr_en      St1
/stim/m1/A1/clk        St0
/stim/m1/A1/reset      St0
/stim/m1/A1/Output1    3
/stim/m1/A1/pr_done    0
/stim/m1/A1/Th_Co...   1
/stim/m1/A1/Count_...  2
/stim/m1/B1/Input1     3
/stim/m1/B1/Throug...  2
/stim/m1/B1/pr_en      St1
/stim/m1/B1/clk        St0
/stim/m1/B1/reset      St0
/stim/m1/B1/Output1    3
/stim/m1/B1/pr_done    1
/stim/m1/B1/Th_Co...   0
/stim/m1/B1/Count_...  1

Now       300 ns
Cursor 1  0 ns    0 ns

0 ns to 315 ns
```

PROCESS (C_D) BEHAVIOR



Signal	Value
/stim/m1/C1/Input1	-No Data-
/stim/m1/C1/Input2	-No Data-
/stim/m1/C1/Throug...	-No Data-
/stim/m1/C1/pr_en	-No Data-
/stim/m1/C1/clk	-No Data-
/stim/m1/C1/reset	-No Data-
/stim/m1/C1/Output1	-No Data-
/stim/m1/C1/Output2	-No Data-
/stim/m1/C1/pr_done	-No Data-
/stim/m1/C1/Th_Co...	-No Data-
/stim/m1/C1/Count_...	-No Data-
/stim/m1/D1/Input1	-No Data-
/stim/m1/D1/Throug...	-No Data-
/stim/m1/D1/pr_en	-No Data-
/stim/m1/D1/clk	-No Data-
/stim/m1/D1/reset	-No Data-
/stim/m1/D1/Output1	-No Data-
/stim/m1/D1/Output2	-No Data-
/stim/m1/D1/pr_done	-No Data-
/stim/m1/D1/Th_Co...	-No Data-
/stim/m1/D1/Count_...	-No Data-

TEST (A_B_C) BEHAVIOR



wave - default

File Edit View Insert Format Tools Window

```
/stim/m1/t1/RC      0000000000000000    00000000000000000000001
/stim/m1/t1/diff1   1                   0 1 2 3 4 5 6 7 8 -7 -6 -5 -4 -3 -2 -1 -2 -1 -2 -1
/stim/m1/t1/diff2   0                   0
/stim/m1/t1/diff3   0                   0
/stim/m1/t1/diff4   0                   0
/stim/m1/t1/diff5   0                   0
/stim/m1/t1/pr_en   1
/stim/m1/t2/RC      0000000000000000    0000000000000000010000
/stim/m1/t2/diff1   0                   0
/stim/m1/t2/diff2   0                   0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
/stim/m1/t2/diff3   0                   0
/stim/m1/t2/diff4   0                   0
/stim/m1/t2/diff5   0                   0
/stim/m1/t2/pr_en   0
/stim/m1/t3/RC      0000000000000001    000000000000000100000000
/stim/m1/t3/diff1   0                   0
/stim/m1/t3/diff2   0                   0
/stim/m1/t3/diff3   0                   0 1 2 1 0 1 2 1 0 1 2 1 0 1 2 1
/stim/m1/t3/diff4   0                   0
/stim/m1/t3/diff5   0                   0
/stim/m1/t3/pr_en   0

Now       300 ns    0  50  100  150  200  250  300
Cursor 1  24 ns     24 ns

0 ns to 315 ns                          wave - default
```

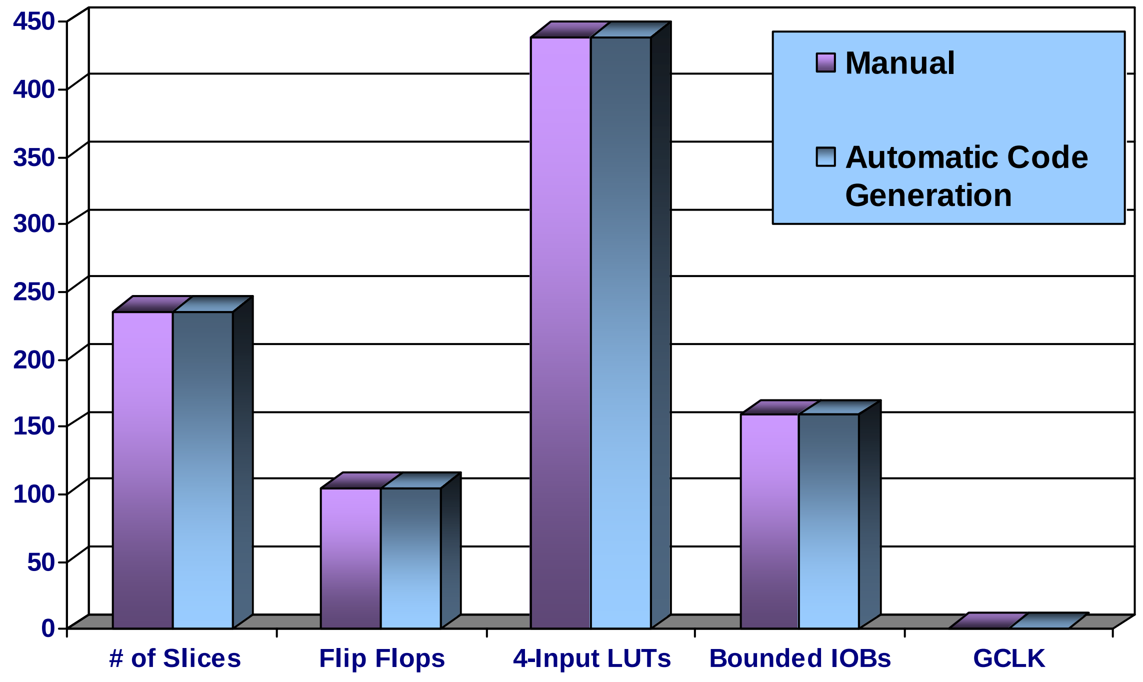
SYNTHESIS RESULTS

RTL CODE SYNTHESIS ON FPGA VIRTEX-4)

DEVICE UTILIZATION SUMMARY

Device Utilization Summary (estimated values)			*[-]*
Logic Utilization	***Used***	***Available***	***Utilization***
Number of Slices	***235***	***6144***	***3%***
Number of Slice Flip Flops	***105***	***12288***	***0%***
Number of 4 input LUTs	***439***	***12288***	***3%***
Number of bonded IOBs	***151***	***240***	***62%***
Number of GCLKs	***1***	***32***	***3%***

COMPARISON OF MANUAL AND COMPILER GENERATED RTL CODES ON FPGA (SPARTAN 3E)



CHAPTER 5

CONCLUSION AND FUTURE WORK

In this scheme, I have developed a structure of KPN that can easily be mapped on any reconfigurable platforms. For that, I ask the specification from the user of specific format, and then the compiler reads it and gives an automatic HDL based controller for hardware mapping. Future work will be the automatic generation of this specification file. By simply viewing the streaming application, an automatic design file will be generated that will be then passed to this generalized compiler which is giving the hardware implementation of KPN framework.

Conclusively, I have proposed and implemented a framework of KPN taking the input specifications of streaming applications resulting into automatic synthesized RTL code generation. This is essential because the actual critical streaming applications is constituent of thousands or millions of such independent components or processing elements and managing/controlling their processing in a big challenge. Also, Execution time of such application requires more than a week and when a matter of designing a manual controller for such application comes, it becomes a huge overhead. Lastly, if the design goes fail then all your effort will go down. For this consideration, we have come to this point that designers just need to put their application specifications/demands and an automatic synthesized RTL optimized controller will be generated without any manual considerations and overheads fulfilling their current design demands and if required, then it can easily be upgraded according to restructured design.

APPENDIX A

C-CODE OF GIVEN DSP DESIGN:

1. DEFINITIONS.H

```
void FIFO();

void TEST();

void B_COUNTER();

void Process_A();

void Process_B();

void Process_C();

void Process_D();

void Process_E();

void KPN();

void STIM();
```

2. MAIN_C.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>

#include"definitions.h"
//#include"FIFO.h"

void main()
{
        FIFO();
        TEST();
        B_COUNTER();
        Process_A();
        Process_B();
        Process_C();
        Process_D();
        Process_E();
        KPN();
        STIM();
        getche();
}
```

3. KPN_CONTROLLER.C

```
#include<conio.h>

#include<process.h>

#include<stdio.h>


void KPN()

{

        FILE *fp;

        fp=fopen("Main_File.V","w");

        if(fp==NULL)

        {

                puts("Cannot Open Target File");

                exit(0);

        }

/*--------------------------------------------------------------------|

|----------------------WRITING TO FILE Main_KPN.V---------------------------|

|--------------------------------------------------------------------*/
```

fprintf(fp,"module Main_File(wr_en1,rc_n1,rc_n2,rc_n3,rc_n4,rc_n5,data_in,data_out,thru_n1,thru_n2,thru_n3,thrU_n4,thru_n5,clk,reset);\n");

//Input Signals

fprintf(fp,"input clk,reset;\n");

//Write enable signal for fifo_1

fprintf(fp,"input wr_en1;\n");

//Throughputs for each node

fprintf(fp,"input [3:0]thru_n1,thru_n2,thru_n3,thru_n4,thru_n5;\n");

//Input Stream data will bestored at here

fprintf(fp,"input [7:0]data_in;\n");

//Rate of consumption at each link

fprintf(fp,"input [19:0]rc_n1,rc_n2,rc_n3,rc_n4,rc_n5;\n\n");

//Output Data

fprintf(fp,"output [7:0]data_out;\n");

fprintf(fp,"reg [7:0]data_out;\n\n");

//Temporary wires and Registers holding Outputs from FIFOs and processes

//and holding control signals as well

fprintf(fp,"wire [7:0]d_out1,d_out2,d_out3,d_out4,d_out5,d_out6,d_out7,d_out8;\n");

fprintf(fp,"wire [7:0]Output1,Output2,Output3,Output4,Output5,Output6,Output7;\n");

fprintf(fp,"wire rd_en1,rd_en2,rd_en3,rd_en4,rd)en5,rd_en5,rd_en6,rd_en7,rd_en8,wr_en2,wr_en3,wr_en4,wr_en5,wr_en6,wr_en7,wr_en8;\n\n");

fprintf(fp,"reg read1,read2,read3,read4,read5,read6,read7,write2,write3,write4,write5,write6,write7,write8;\n\n");

//Flags showing FIFO Status i.e (Fifo Full and Fifo Empty)

fprintf(fp,"wire f_full_flag1,f_full_flag2,f_full_flag3,f_full_flag4;\n");

fprintf(fp,"wire f_empty_flag1,f_empty_flag2,f_empty_flag3,f_empty_flag4;\n\n");

//Enable the process and Done Signals showing process has completed its execution

//and ready to write data at ints output buffer

fprintf(fp,"wire pr_en1,pr_en2,pr_en3,pr_done1,pr_done2,pr_done3;\n\n");

//Pointer that stores the Difference between w_ptr and r_ptr

fprintf(fp,"wire [3:0]diff_ptr1,diff_ptr2,diff_ptr3,diff_ptr4;\n\n");

//Call FIFO Instances

fprintf(fp,"FIFO f1(diff_ptr1,d_out1,f_full_flag1,f_empty_flag1,data_in,rd_en1,wr_en1,clk,reset);\n");

fprintf(fp,"FIFO f2(diff_ptr2,d_out2,f_full_flag2,f_empty_flag2,Output1,rd_en2,wr_en2,clk,reset);\n");

fprintf(fp,"FIFO f3(diff_ptr3,d_out3,f_full_flag3,f_empty_flag3,Output2,rd_en3,wr_en3,clk,reset);\n");

fprintf(fp,"FIFO f4(diff_ptr4,d_out4,f_full_flag4,f_empty_flag4,Output3,rd_en4,wr_en4,clk,reset);\n\n");

fprintf(fp,"FIFO f5(diff_ptr5,d_out5,f_full_flag5,f_empty_flag5,Output4,rd_en5,wr_en5,clk,reset);\n");

fprintf(fp,"FIFO f6(diff_ptr6,d_out6,f_full_flag6,f_empty_flag6,Output5,rd_en6,wr_en6,clk,reset);\n");

fprintf(fp,"FIFO f7(diff_ptr7,d_out7,f_full_flag7,f_empty_flag7,Output6,rd_en7,wr_en7,clk,reset);\n");

fprintf(fp,"FIFO f8(diff_ptr8,d_out8,f_full_flag8,f_empty_flag8,Output7,rd_en8,wr_en8,clk,reset);\n\n");

//Process Instantiations

fprintf(fp,"ProcessA A1(clk,reset,pr_en1,d_out1,Output1,Output6,thru_n1,pr_done1);\n");

fprintf(fp,"ProcessB B1(clk,reset,pr_en2,d_out2,Output2,thru_n2,pr_done2);\n");

fprintf(fp,"ProcessC C1(clk,reset,pr_en3,d_out3,Output3,Output4,thru_n3,pr_done3);\n");

fprintf(fp,"ProcessD D1(clk,reset,pr_en4,d_out4,Output5,thru_n4,pr_done4);\n");

fprintf(fp,"ProcessE E1(clk,reset,pr_en5,d_out5,Output7,thru_n5,pr_done5);\n\n");

//Test Module checking sufficient tokens have acquired in each fifo to start execution

fprintf(fp,"TEST t1(rc_n1,diff_ptr1,4'b0,4'b0,4'b0,4'b0,pr_en1);\n");

fprintf(fp,"TEST t2(rc_n2,4'b0,diff_ptr2,4'b0,4'b0,4'b0,pr_en2);\n");

fprintf(fp,"TEST t3(rc_n3,diff_ptr1,4'b0,4'b0,4'b0,4'b0,pr_en3);\n");

```
fprintf(fp,"TEST t4(rc_n4,4'b0,diff_ptr2,4'b0,4'b0,4'b0,pr_en4);\n");
fprintf(fp,"TEST t5(rc_n5,4'b0,4'b0,diff_ptr3,4'b0,4'b0,pr_en5);\n\n");



//Simple assignments
//assign data_out=d_out4;
fprintf(fp,"assign rd_en1=read1;\n");
fprintf(fp,"assign rd_en2=read2;\n");
fprintf(fp,"assign rd_en3=read3;\n");
fprintf(fp,"assign rd_en4=read4;\n");
fprintf(fp,"assign rd_en5=read5;\n");
fprintf(fp,"assign rd_en6=read6;\n");
fprintf(fp,"assign rd_en7=read7;\n");
fprintf(fp,"assign rd_en4=1;\n\n");
fprintf(fp,"assign wr_en2=write2;\n");
fprintf(fp,"assign wr_en3=write3;\n");
fprintf(fp,"assign wr_en4=write4;\n");
fprintf(fp,"assign wr_en5=write5;\n");
fprintf(fp,"assign wr_en6=write6;\n");
fprintf(fp,"assign wr_en7=write7;\n");
fprintf(fp,"assign wr_en8=write8;\n\n");



//When Pr_en1 is active high or low
```

```
fprintf(fp,"always @(posedge clk)\n");

fprintf(fp,"if(pr_en1)\n");

fprintf(fp,"\tif(pr_done1)\n\t\tread1=1;\n\telse\n\t\tread1=0;\n\n");

fprintf(fp,"else\n");

fprintf(fp,"\tread1=0;\n\n");


//When pr_en2 is active high or low

fprintf(fp,"always @(posedge clk)\n");

fprintf(fp,"if(pr_en2)\n");

fprintf(fp,"\tif(pr_done2)\n\t\tread2=1;\n\telse\n\t\tread2=0;\n\n\n");

fprintf(fp,"else\n");

fprintf(fp,"\tread2=0;\n\n");


//When pr_en3 is active high or low

fprintf(fp,"always @(posedge clk)\n");

fprintf(fp,"if(pr_en3)\n");

fprintf(fp,"\tif(pr_done3)\n\t\tread3=1;\n\telse\n\t\tread3=0;\n\n");

fprintf(fp,"else\n");

fprintf(fp,"\tread3=0; \n\n");


//When pr_en4 is active high or low

fprintf(fp,"always @(posedge clk)\n");

fprintf(fp,"if(pr_en4)\n");

fprintf(fp,"\tif(pr_done4)\n\t\tread4=1;\n\telse\n\t\tread4=0;\n\n");
fprintf(fp,"else\n");

fprintf(fp,"\tread4=0; \n\n");



//When pr_en3 is active high or low

fprintf(fp,"always @(posedge clk)\n");

fprintf(fp,"if(pr_en5)\n");

fprintf(fp,"\tif(pr_done5)\n\t\tread3=1;\n\telse\n\t\tread5=0;\n\n");

fprintf(fp,"else\n");

fprintf(fp,"\tread5=0; \n\n");




//When pr_done1 is active high or low

fprintf(fp,"always @(pr_done1)\n");

fprintf(fp,"if(pr_done1)\n");

fprintf(fp,"\tif(pr_en1)\n\t\twrite2=1;\n\telse\n\t\twrite2=0;\n");

fprintf(fp,"else\n");

fprintf(fp,"\twrite2=0;\n\n");


//When pr_done2 is active high or low

fprintf(fp,"always @(pr_done2)        \n");

fprintf(fp,"if(pr_done2)\n");

fprintf(fp,"\tif(pr_en2)\n\t\twrite3=1;\n\telse\n\t\twrite3=0;\n");
fprintf(fp,"else\n");
fprintf(fp,"\twrite3=0;\n\n");

//When pr_done3 is active high or low
fprintf(fp,"always @(pr_done3)\n");
fprintf(fp,"if(pr_done3)\n");
fprintf(fp,"\tif(pr_en3)\n\t\twrite4=1;\n\telse\n\t\twrite4=0;\n");
fprintf(fp,"else\n");
fprintf(fp,"\twrite4=0; \n\n");

//When pr_done4 is active high or low
fprintf(fp,"always @(pr_done4)\n");
fprintf(fp,"if(pr_done4)\n");
fprintf(fp,"\tif(pr_en4)\n\t\twrite5=1;\n\telse\n\t\twrite5=0;\n");
fprintf(fp,"else\n");
fprintf(fp,"\twrite5=0; \n\n");

//When pr_done3 is active high or low
fprintf(fp,"always @(pr_done5)\n");
fprintf(fp,"if(pr_done5)\n");
fprintf(fp,"\tif(pr_en5)\n\t\twrite6=1;\n\telse\n\t\twrite6=0;\n");
fprintf(fp,"else\n");
fprintf(fp,"\twrite6=0; \n\n");
fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tif(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tdata_out<=0;\n");
fprintf(fp,"\tend\n");
fprintf(fp,"\telse\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tdata_out<=d_out4;\n");
fprintf(fp,"\tend\n\n");
fprintf(fp,"end\n\n");
fprintf(fp,"endmodule\n");

puts("Data Main_File.V Copied");
fclose(fp);
}
```

4. FIFO.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>

void FIFO()
```

```
{
        FILE *fp;
        fp=fopen("FIFO.V","w");
        if(fp==NULL)
        {
                puts("Cannot Open Target File");
                exit(0);
        }

/*-------------------------------------------------------------------|
|---------------------WRITING TO FILE FIFO.V--------------------------|
|-------------------------------------------------------------------*/
int f_width=8;//Token size=8 bits
int f_depth=16;//Fifo Depth=16 locations of 8 bits wide
int f_ptr_width=4;//ptr size to address 16 locations
int flag;
int ptr_diff=0;
int rd_ptr,wr_ptr;

        fprintf(fp,"module
FIFO(ptr_diff,output_data,f_full_flag,f_empty_flag,input_data,rd_enable,wr_enable,cloc
k,reset);\n");
   fprintf(fp,"parameter f_depth=16; //FIFO depth\n\n");
        fprintf(fp,"input rd_enable,wr_enable,clock,reset;\n");
        fprintf(fp,"input [%d:0]input_data;\n",(f_width-1));
        //fprintf(fp,"input [7:0]Data_In;\n\n");


        fprintf(fp,"output [%d:0]ptr_diff;\n",(f_ptr_width-1));
        fprintf(fp,"output [%d:0]output_data;\n",(f_width-1));
        fprintf(fp,"output f_full_flag,f_empty_flag;\n\n");
```

//Outputs need to be declared as reg for behavioral modeling

```
        fprintf(fp,"//Outputs need to be declared as reg for behavioral modeling\n");
        fprintf(fp,"reg [%d:0]output_data;\n\n",(f_width-1));
```

//Internal wires, registers and register file declarations

```
        fprintf(fp,"//Internal wires, registers and register file declarations\n");
        fprintf(fp,"reg [%d:0]diff;\n",(f_ptr_width-1));
        fprintf(fp,"wire [%d:0]rd_ptr,wr_ptr;\n",(f_ptr_width-1));
        fprintf(fp,"reg rd_next_en,wr_next_en;\n");
        fprintf(fp,"reg [%d:0]f_memory[0:%d];\n\n",(f_width-1),(f_depth-1));
```

//Simple assignments

```
        fprintf(fp,"assign ptr_diff=diff;\n");
        if(ptr_diff==(f_depth-1)) //IF_ELSE for checking FIFO is FULL
                flag=1;
        else
                flag=0;
        fprintf(fp,"assign f_full_flag=(ptr_diff==(f_depth-1));\n");
```

```
            if(ptr_diff==0)//IF_ELSE checking the empty status of FIFO Buffer
                    flag=1;
            else
                    flag=0;
            fprintf(fp,"assign f_empty_flag=(ptr_diff==0);\n\n");


//instantiate address counters for increments and decrements
            fprintf(fp,"b_counter
rd_b_counter(.c_out(rd_ptr),.c_reset(reset),.c_clk(clock),.en(rd_next_en));\n");
            fprintf(fp,"b_counter
wr_b_counter(.c_out(wr_ptr),.c_reset(reset),.c_clk(clock),.en(wr_next_en));\n\n\n");


/*----------Always block starts at here -----------
|-------------------------------------------------*/
            fprintf(fp,"always @(posedge clock)\n");
            fprintf(fp,"begin\n\n");


            fprintf(fp,"if(reset)\n");
            fprintf(fp,"\toutput_data=%d;\n\n",0); //Output must be reset to zero when reset
signal is asserted


            //if write signal is asserted then first you need to check whether fifo is full or not
            //if not, then input data is written into fifo buffer
            fprintf(fp,"if(wr_enable) begin\n");
            fprintf(fp,"\tif(!f_full_flag)\n");
            fprintf(fp,"\t\tf_memory[wr_ptr]<=input_data;\n");
```

fprintf(fp,"\tend\n\n");

//if read signal is asserted then first you need to check whether fifo is empty or not

//if not, then fifo buffer is being read

fprintf(fp,"if(rd_enable) begin\n");

fprintf(fp,"\tif(!f_empty_flag)\n");

fprintf(fp,"\t\toutput_data<=f_memory[rd_ptr];\n");

fprintf(fp,"\tend\n\n");

fprintf(fp,"end\n\n");

//--

fprintf(fp,"always @(*)\n\n"); //ptr_diff changes as clock changes

fprintf(fp,"begin \n\n");

fprintf(fp,"if(wr_ptr>rd_ptr)\n");

fprintf(fp,"\tdiff<=wr_ptr-rd_ptr;\n\n");

fprintf(fp,"else if(wr_ptr<rd_ptr)\n");

fprintf(fp,"\tdiff<=%d;\n\n",((f_depth-rd_ptr)+wr_ptr));

fprintf(fp,"else diff<=0;\n\n");

fprintf(fp,"end\n\n\n");

//--

```
            fprintf(fp,"always @(*)\n\n"); //after empty flag activated fifo read counter should
not increment;

            fprintf(fp,"begin \n\n");

            fprintf(fp,"if(rd_enable && (!f_empty_flag))\n");

            fprintf(fp,"\trd_next_en=1;\n\n");

            fprintf(fp,"else \n");

            fprintf(fp,"\trd_next_en=0;\n\n");

            fprintf(fp,"end\n\n\n");


//-------------------------------------------------------

            fprintf(fp,"always @(*)\n\n"); //after full flag activated fifo write counter should
not increment;

            fprintf(fp,"begin \n\n");

            fprintf(fp,"if(wr_enable && (!f_full_flag))\n");

            fprintf(fp,"\twr_next_en=1;\n\n");

            fprintf(fp,"else \n");

            fprintf(fp,"\twr_next_en=0;\n\n");

            fprintf(fp,"end\n\n");


//-------------------------------------------------------

            fprintf(fp,"endmodule");


puts("Data FIFO.V Copied");

fclose(fp);

}
```

5. B_COUNTER.C

```
#include<conio.h>

#include<process.h>

#include<stdio.h>


void B_COUNTER()

{

int c_width=4;


        FILE *fp;

        fp=fopen("b_counter.V","w");


        if(fp==NULL)

        {

        puts("Cannot Open Source File");

                exit(0);

        }


        /*-----------------------------------------------------------------------|

        |----------------------WRITING TO FILE b_counter.V----------------------|

        |-----------------------------------------------------------------------*/

        fprintf(fp,"module b_counter(c_out,c_reset,c_clk,en);\n\n");

        fprintf(fp,"parameter c_width=%d;\n\n",c_width);
```

```
            //Input Specifications

            fprintf(fp,"input c_reset,c_clk,en;\n\n");


            //Output specifications and also it needs to be declared as reg for behavioral
modeling

            fprintf(fp,"output [%d:0]c_out;\n",(c_width-1));

            fprintf(fp,"reg [%d:0]c_out;\n\n",(c_width-1));


            /*----------Always block starts at here -----------

            |-----------------------------------------------*/

            fprintf(fp,"always @(posedge c_clk or posedge c_reset)\n");

            fprintf(fp,"if(c_reset)\n");

            fprintf(fp,"\tc_out<=0;\n");

            fprintf(fp,"else if(en)\n");

            fprintf(fp,"\tc_out<=c_out+1;\n");

            fprintf(fp,"else\n");

            fprintf(fp,"\tc_out<=c_out;\n\n");


            fprintf(fp,"endmodule");


puts("Data B_Counter.V Copied");

fclose(fp);

}
```

6. PROCESS_A.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>

void Process_A(){

        FILE *fp;
        fp=fopen("ProcessA.V","w");

        if(fp==NULL)
        {
        puts("Cannot Open Source File");
                exit(0);
        }

        /*------------------------------------------------------------------
        |----In Actual sense, this block contains the actual processing element
        that needs to be executed. But we are just sketshing the hardware, so at
        here, we will just waste throughput number of clock cycles-------------*/



        /*-------------------------------------------------------------------|
        |---------------------WRITING TO FILE Process_A.V----------------------|
|---------------------------------------------------------------------*/
```

//Input Specifications

fprintf(fp,"module ProcessA(clk,reset,pr_en,Input1,Output1,Output2,Throughput,pr_done);\n");

fprintf(fp,"parameter width=8; //FIFO width\n\n");

fprintf(fp,"input [width-1:0]Input1;\n");

fprintf(fp,"input [3:0]Throughput;\n");

fprintf(fp,"input pr_en,clk,reset;\n\n");

//Output Specifications

fprintf(fp,"output [width-1:0]Output1;\n");

fprintf(fp,"output pr_done;\n");

fprintf(fp,"reg pr_done;\n\n");

//Temporary Registers and wires

fprintf(fp,"reg [3:0]Th_Counter;\n");

fprintf(fp,"wire [3:0]Count_Inc;\n");

fprintf(fp,"assign Count_Inc=Th_Counter+1;\n\n");

fprintf(fp,"assign Output1=Input1;\n\n");

fprintf(fp,"assign Output2=Input1;\n\n");

//Simply Waste Clock Cycles for process internal algorithm execution

```
/*-------Always block starts at here ---------
|--------------------------------------------*/

fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");

fprintf(fp,"if(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tpr_done<=0;\n");
fprintf(fp,"\t\tTh_Counter<=0;\n");
fprintf(fp,"\tend\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tif(pr_en)\n");
fprintf(fp,"\t\tTh_Counter=Count_Inc;\n\n");
fprintf(fp,"\t\telse\n");
fprintf(fp,"\t\tTh_Counter=Th_Counter;//Do Nothing\n\n");
fprintf(fp,"\tend\n\n");

fprintf(fp,"end\n\n");


        /*-------Always block For Counting starts at here ---------
```

```
          |-----------------------------------------------------------*/

fprintf(fp,"always @(Th_Counter)\n");
fprintf(fp,"if(Th_Counter == Throughput)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tpr_done=1;\n");
fprintf(fp,"\t#5 Th_Counter=0;\n");
fprintf(fp,"end\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tif(pr_en)\n\t\tpr_done=0;\n\telse\n\t\tpr_done=1;\n\n");

fprintf(fp,"endmodule");

puts("Data Process_A.V Copied");
fclose(fp);

}
```

7. PROCESS_B.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>
```

```
void Process_B(){

        FILE *fp;
        fp=fopen("ProcessB.V","w");

        if(fp==NULL)
        {
        puts("Cannot Open Source File");
                exit(0);
        }

        /*--------------------------------------------------------------------
        |----In Actual sense, this block contains the actual processing element
        that needs to be executed. But we are just sketshing the hardware, so at
        here, we will just waste throughput number of clock cycles-------------*/



        /*--------------------------------------------------------------------|
        |---------------------WRITING TO FILE Process_B.V-----------------------|
        |--------------------------------------------------------------------*/

//Input Specifications
fprintf(fp,"module ProcessB(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);\n");
fprintf(fp,"parameter width=8; //FIFO width\n\n");
fprintf(fp,"input [width-1:0]Input1;\n");
fprintf(fp,"input [3:0]Throughput;\n");
fprintf(fp,"input pr_en,clk,reset;\n\n");

//Output Specifications
fprintf(fp,"output [width-1:0]Output1;\n");
fprintf(fp,"output pr_done;\n");
fprintf(fp,"reg pr_done;\n\n");

//Temporary Registers and wires
fprintf(fp,"reg [3:0]Th_Counter;\n");
fprintf(fp,"wire [3:0]Count_Inc;\n");
fprintf(fp,"assign Count_Inc=Th_Counter+1;\n\n");

fprintf(fp,"assign Output1=Input1;\n\n");

//Simply Waste Clock Cycles for process internal algorithm execution

/*-------Always block starts at here ---------
|--------------------------------------------*/

fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");
fprintf(fp,"if(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tpr_done<=0;\n");
fprintf(fp,"\t\tTh_Counter<=0;\n");
fprintf(fp,"\tend\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tif(pr_en)\n");
fprintf(fp,"\t\tTh_Counter=Count_Inc;\n\n");
fprintf(fp,"\t\telse\n");
fprintf(fp,"\t\tTh_Counter=Th_Counter;//Do Nothing\n\n");
fprintf(fp,"\tend\n\n");

fprintf(fp,"end\n\n");


        /*-------Always block For Counting starts at here ---------
        |--------------------------------------------------------*/

fprintf(fp,"always @(Th_Counter)\n");
fprintf(fp,"if(Th_Counter == Throughput)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tpr_done=1;\n");
fprintf(fp,"\t#5 Th_Counter=0;\n");
fprintf(fp,"end\n\n");


fprintf(fp,"else\n");
fprintf(fp,"\tif(pr_en)\n\t\tpr_done=0;\n\telse\n\t\tpr_done=1;\n\n");


fprintf(fp,"endmodule");


puts("Data Process_B.V Copied");
fclose(fp);


}
```

8. PROCESS_C.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>


void Process_C(){


        FILE *fp;
        fp=fopen("ProcessC.V","w");


        if(fp==NULL)
        {
        puts("Cannot Open Source File");
                exit(0);
        }

        /*--------------------------------------------------------------------
        |----In Actual sense, this block contains the actual processing element
        that needs to be executed. But we are just sketshing the hardware, so at
        here, we will just waste throughput number of clock cycles-------------*/



        /*--------------------------------------------------------------------|
        |----------------------WRITING TO FILE Process_C.V------------------------|
        |--------------------------------------------------------------------*/

//Input Specifications
fprintf(fp,"module
ProcessC(clk,reset,pr_en,Input1,Output1,Output2,Throughput,pr_done);\n");
fprintf(fp,"parameter width=8; //FIFO width\n\n");
fprintf(fp,"input [width-1:0]Input1;\n");
fprintf(fp,"input [3:0]Throughput;\n");
fprintf(fp,"input pr_en,clk,reset;\n\n");

//Output Specifications
fprintf(fp,"output [width-1:0]Output1;\n");
fprintf(fp,"output pr_done;\n");
fprintf(fp,"reg pr_done;\n\n");


//Temporary Registers and wires
fprintf(fp,"reg [3:0]Th_Counter;\n");
fprintf(fp,"wire [3:0]Count_Inc;\n");
fprintf(fp,"assign Count_Inc=Th_Counter+1;\n\n");


fprintf(fp,"assign Output1=Input1;\n\n");
fprintf(fp,"assign Output2=Input1;\n\n");


//Simply Waste Clock Cycles for process internal algorithm execution


/*-------Always block starts at here ---------
|-------------------------------------------*/


fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");


fprintf(fp,"if(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tpr_done<=0;\n");
fprintf(fp,"\t\tTh_Counter<=0;\n");
fprintf(fp,"\tend\n\n");
fprintf(fp,"else\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tif(pr_en)\n");
fprintf(fp,"\t\tTh_Counter=Count_Inc;\n\n");
fprintf(fp,"\t\telse\n");
fprintf(fp,"\t\tTh_Counter=Th_Counter;//Do Nothing\n\n");
fprintf(fp,"\tend\n\n");

fprintf(fp,"end\n\n");

        /*-------Always block For Counting starts at here ---------
        |----------------------------------------------------------*/

fprintf(fp,"always @(Th_Counter)\n");
fprintf(fp,"if(Th_Counter == Throughput)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tpr_done=1;\n");
fprintf(fp,"\t#5 Th_Counter=0;\n");
fprintf(fp,"end\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tif(pr_en)\n\t\tpr_done=0;\n\telse\n\t\tpr_done=1;\n\n");
```

```
 fprintf(fp,"endmodule");


 puts("Data Process_C.V Copied");
 fclose(fp);
 }
```

9. PROCESS_D.C

```
#include<conio.h>
#include<process.h>
#include<stdio.h>

void Process_D(){


        FILE *fp;
        fp=fopen("ProcessD.V","w");

        if(fp==NULL)
        {
        puts("Cannot Open Source File");
                exit(0);
        }

        /*-------------------------------------------------------------------
        |----In Actual sense, this block contains the actual processing element
```

that needs to be executed. But we are just sketshing the hardware, so at here, we will just waste throughput number of clock cycles-------------*/

```
/*-------------------------------------------------------------------|
|---------------------WRITING TO FILE Process_D.V-----------------------|
|--------------------------------------------------------------------*/
```

//Input Specifications

```
fprintf(fp,"module ProcessD(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);\n");
fprintf(fp,"parameter width=8; //FIFO width\n\n");
fprintf(fp,"input [width-1:0]Input1;\n");
fprintf(fp,"input [3:0]Throughput;\n");
fprintf(fp,"input pr_en,clk,reset;\n\n");
```

//Output Specifications

```
fprintf(fp,"output [width-1:0]Output1;\n");
fprintf(fp,"output pr_done;\n");
fprintf(fp,"reg pr_done;\n\n");
```

//Temporary Registers and wires

```
fprintf(fp,"reg [3:0]Th_Counter;\n");
fprintf(fp,"wire [3:0]Count_Inc;\n");
fprintf(fp,"assign Count_Inc=Th_Counter+1;\n\n");
fprintf(fp,"assign Output1=Input1;\n\n");


//Simply Waste Clock Cycles for process internal algorithm execution


/*-------Always block starts at here ---------
|--------------------------------------------*/


fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");


fprintf(fp,"if(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tpr_done<=0;\n");
fprintf(fp,"\t\tTh_Counter<=0;\n");
fprintf(fp,"\tend\n\n");


fprintf(fp,"else\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tif(pr_en)\n");
fprintf(fp,"\t\tTh_Counter=Count_Inc;\n\n");
fprintf(fp,"\t\telse\n");
fprintf(fp,"\t\tTh_Counter=Th_Counter;//Do Nothing\n\n");
fprintf(fp,"\tend\n\n");
fprintf(fp,"end\n\n");

        /*-------Always block For Counting starts at here ---------
        |---------------------------------------------------------*/

fprintf(fp,"always @(Th_Counter)\n");
fprintf(fp,"if(Th_Counter == Throughput)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tpr_done=1;\n");
fprintf(fp,"\t#5 Th_Counter=0;\n");
fprintf(fp,"end\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tif(pr_en)\n\t\tpr_done=0;\n\telse\n\t\tpr_done=1;\n\n");

fprintf(fp,"endmodule");
puts("Data Process_D.V Copied");
fclose(fp);

}
```

10. PROCESS_E.C

```
#include<conio.h>
#include<process.h>
```

```
#include<stdio.h>


void Process_E(){
        FILE *fp;
        fp=fopen("ProcessE.V","w");

        if(fp==NULL)
        {
        puts("Cannot Open Source File");
                exit(0);
        }

        /*-------------------------------------------------------------------
        |----In Actual sense, this block contains the actual processing element
        that needs to be executed. But we are just sketshing the hardware, so at
        here, we will just waste throughput number of clock cycles-------------*/



        /*-------------------------------------------------------------------|
        |---------------------WRITING TO FILE Process_E.V-----------------------|
        |-------------------------------------------------------------------*/

//Input Specifications
fprintf(fp,"module ProcessE(clk,reset,pr_en,Input1,Output1,Throughput,pr_done);\n");
fprintf(fp,"parameter width=8; //FIFO width\n\n");
fprintf(fp,"input [width-1:0]Input1;\n");
fprintf(fp,"input [3:0]Throughput;\n");
fprintf(fp,"input pr_en,clk,reset;\n\n");

//Output Specifications
fprintf(fp,"output [width-1:0]Output1;\n");
fprintf(fp,"output pr_done;\n");
fprintf(fp,"reg pr_done;\n\n");

//Temporary Registers and wires
fprintf(fp,"reg [3:0]Th_Counter;\n");
fprintf(fp,"wire [3:0]Count_Inc;\n");
fprintf(fp,"assign Count_Inc=Th_Counter+1;\n\n");

fprintf(fp,"assign Output1=Input1;\n\n");

//Simply Waste Clock Cycles for process internal algorithm execution

/*-------Always block starts at here ---------
|-------------------------------------------*/

fprintf(fp,"always @(posedge clk)\n");
fprintf(fp,"begin\n");
fprintf(fp,"if(reset)\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tpr_done<=0;\n");
fprintf(fp,"\t\tTh_Counter<=0;\n");
fprintf(fp,"\tend\n\n");


fprintf(fp,"else\n");
fprintf(fp,"\tbegin\n");
fprintf(fp,"\t\tif(pr_en)\n");
fprintf(fp,"\t\tTh_Counter=Count_Inc;\n\n");
fprintf(fp,"\t\telse\n");
fprintf(fp,"\t\tTh_Counter=Th_Counter;//Do Nothing\n\n");
fprintf(fp,"\tend\n\n");


fprintf(fp,"end\n\n");


        /*-------Always block For Counting starts at here ---------
        |--------------------------------------------------------*/


fprintf(fp,"always @(Th_Counter)\n");
fprintf(fp,"if(Th_Counter == Throughput)\n");
fprintf(fp,"begin\n");
fprintf(fp,"\tpr_done=1;\n");
fprintf(fp,"\t#5 Th_Counter=0;\n");
fprintf(fp,"end\n\n");

fprintf(fp,"else\n");
fprintf(fp,"\tif(pr_en)\n\t\tpr_done=0;\n\telse\n\t\tpr_done=1;\n\n");

fprintf(fp,"endmodule");

puts("Data Process_E.V Copied");
fclose(fp);
}
```

11. TEST_RC_RP.C

```
#include<stdio.h>
#include<process.h>
#include<conio.h>

void TEST()
{

        FILE *fp;
        fp=fopen("TEST.V","w");

        if(fp==NULL)
        {
                puts("Cannot Open Target File");
                exit(0);
        }

/*-------------------------------------------------------------------|
|---------------------WRITING TO FILE TEST.V--------------------------|
|-------------------------------------------------------------------*/
int f_width=8;//Token size=8 bits
int f_depth=16;//Fifo Depth=16 locations of 8 bits wide
int f_ptr_width=4;//ptr size to address 16 locations
int flag;
int ptr_diff=0;
int rd_ptr,wr_ptr;

        fprintf(fp,"module TEST(RC,diff1,diff2,diff3,diff4,diff5,pr_en);\n");
        fprintf(fp,"input [19:0]RC;\n");
        fprintf(fp,"input [3:0]diff1,diff2,diff3,diff4,diff5;\n\n");

        fprintf(fp,"output pr_en;\n");
        fprintf(fp,"reg pr_en;\n\n");

        fprintf(fp,"always @(diff1 or diff2 or diff3 or diff5 or diff5 or RC)\n\n");
        fprintf(fp,"if(diff1>=RC[19:16] && diff2>=RC[15:12] &&
diff3>=RC[11:8]\n");
```

```
        fprintf(fp,"&& diff4>=RC[7:4] && diff5>=RC[3:0])\n");


        fprintf(fp,"\tbegin\n");

        fprintf(fp,"\tpr_en=1;\n");

        fprintf(fp,"\tend\n");


        fprintf(fp,"else\n");

        fprintf(fp,"\tbegin\n");

        fprintf(fp,"\tpr_en=0;\n");

        fprintf(fp,"\tend\n\n");

        fprintf(fp,"endmodule\n");

puts("Data Test.V Copied");


fclose(fp);

}
```

12. STIMULUS.C

```
#include<conio.h>

#include<process.h>

#include<stdio.h>


void STIM()

{

        FILE *fp;

        fp=fopen("stim.V","w");
        if(fp==NULL)
        {
                puts("Cannot Open Target File");
                exit(0);
        }
/*----------------------------------------------------------------------|
|----------------------WRITING TO FILE STIMULUS.V---------------------------|
|-----------------------------------------------------------------------*/
fprintf(fp,"module stim;\n\n");
fprintf(fp,"reg clock,reset;\n");
fprintf(fp,"reg Wr_Enable_F1;\n");
fprintf(fp,"reg
[3:0]Thru_Node1,Thru_Node2,Thru_Node3,Thru_Node4,Thru_Node5;\n");
fprintf(fp,"reg [7:0]Input_Data;\n");
fprintf(fp,"reg [19:0]Rc_Node1,Rc_Node2,Rc_Node3,Rc_Node4,Rc_Node5;\n\n");
fprintf(fp,"wire [7:0]Output_Data;\n\n");
fprintf(fp,"Main_File
mm(Wr_Enable_F1,Rc_Node1,Rc_Node2,Rc_Node3,Rc_node4,Rc_node5,Input_Data,O
utput_Data,\n\t");
fprintf(fp,"   Thru_Node1,Thru_Node2,Thru_Node3,Thru_n3,Thru_n5,clock,reset);\n\n");
fprintf(fp,"initial\n");
fprintf(fp,"begin\n");
fprintf(fp,"clock=0;\n");
fprintf(fp,"forever\n");
fprintf(fp,"#5 clock=~clock;\n");
fprintf(fp,"end\n\n");

fprintf(fp,"initial\n");

fprintf(fp,"begin\n");

fprintf(fp,"reset=1;\n");


fprintf(fp,"\nThru_Node1=3;Thru_Node2=2;Thru_Node3=1;Thru_Node4=2;Thru_Node
5=1;\n");

fprintf(fp,"#15 reset=0;\n");

fprintf(fp,"\nRc_Node1=20'b0001_0000_0000_0000_0000;\n");

fprintf(fp,"Rc_Node2=20'b0000_0001_0000_0000_0000;\n");

fprintf(fp,"Rc_Node3=20'b0000_0000_0001_0000_0000;\n");

fprintf(fp,"Rc_Node4=20'b0000_0001_0000_0000_0000;\n");

fprintf(fp,"Rc_Node5=20'b0000_0000_0001_0000_0000;\n");

fprintf(fp,"Input_Data=1;\n\n");

fprintf(fp,"#10\n");

fprintf(fp,"repeat(15) #30 Input_Data=Input_Data+1;\n");

fprintf(fp,"end\n");

fprintf(fp,"initial\n");

fprintf(fp,"#200 $stop;\n\n");


fprintf(fp,"initial\n");

fprintf(fp,"begin\n");

fprintf(fp,"Wr_Enable_F1=1;\n");


fprintf(fp,"end\n\n");
fprintf(fp,"endmodule\n");


puts("Data STIM.V Copied");

fclose(fp);


}
```

REFERENCES

[1] Gilles Kahn, “The semantics of a simple language for parallel programming”. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471-475. IFIP, North-Holland, August 1974.

[2] Edward A. Lee and Thomas M. Parks, “Dataflow process networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[3] Eric Cheung, Harry Hsieh, and Feris Baralin, “Automatic Buffer Sizing for Rate-Constrained KPNApplications on Multiprocessor System-on- Chip,” Proceedings of IEEE, pages 37-44, 2007.

[4] Marc Geilen and Twan Basten, “Requirements on the execution of kahn process networks,” In *Programming Languages and Systems*, *12th European Symposium on Programming, ESOP 2003*, pages 319-334, Warsaw, Poland, April 2003. Lecture Notes in Computer Science vol. 2618.

[5] Twan Basten and Jan Hoogerbrugge, “Efficient execution of process networks”. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proc. Communicating Process Architectures*, pages 1-14, Bristol, UK, September 2001. IOS Press

[6] Thomas M. Parks, “*Bounded Scheduling of Process Networks*,” PhD Thesis, EECS Department, University of California, Berkeley, CA, December 1995.

[7] Bharath N., S.K. Nandy, and Nagaraju Bussa, “Artificial Deadlock Detection in Process Networks for Eclipse”, Proceedings of 16th International Conference on Application-Specific Systems, Architectures and Processors, IEEE Computer Society, 1063-6862/05, 2005

[8] Ceponis J., Kazanavicius E., Mikuckas A., “Design and Analysis of DSP systems using Kahn process Networks,” DSP Lab, Kaunas University of technology, ISSN 1392-2114Ultragarsas, Nr .4(45), 2002.

[9] Zvironas A., Kazanavicius E. Partitioning of DSP tasks to Kahn network. KTU. Kaunas. Ultragarsas. ISSN1392-2114, 2002. Nr. 2(43).

[10] Javed DULLOO, Philippe MARQUET, “Design of a Real-Time Scheduler for Kahn Process Networks on Multiprocessor systems,” Rapport LIFL # 2003-06, september 2003.

[11] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere, “System Design using Kahn Process Netwroks: The Compaan/Laura Approach,” Presented at DATE’04, Paris 16-20 Feb 2004.

[12] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, “Yapi: application modeling for signal processing systems,” in *DAC ’00: Proceedings of the 37th conference on Design automation*. New York, NY, USA: ACM Press, 2000, pp. 402–405.

[13] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, “System level design with spade: an m-jpeg case study,” in *ICCAD ’01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 31–38.

[14] Dr. Shoab A. Khan, Book: “Digital Design for Signal Processing Systems” to be published.