Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization

Gargi Roy

Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization

Thesis submitted to the Indian Institute of Technology, Kharagpur
for the award of the degree

of

Master of Science (by research)

by

Gargi Roy

Under the supervision of
Prof. Chittaranjan Mandal

School of Information Technology
Indian Institute of Technology, Kharagpur
West Bengal 721302, India

October, 2014

Dedicated to my parents, for their endless support and patience

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **"Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization,"** submitted by **Gargi Roy** to the Indian Institute of Technology, Kharagpur, for the award of the degree of Master In Science (by research) has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

Prof. Dipankar Sarkar (Member of the DAC)	Prof. K. S. Rao (Member of the DAC)
Prof. Debashis Samanta (Member of the DAC)	Prof. Chittaranjan Mandal (Supervisor)
(External Examiner)	(Chairman)

Date:

Indian Institute of Technology Kharagpur
Certificate by the Supervisor

Date: 10-10-2014

This is to certify that the thesis entitled,
“Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization ”
submitted by GARGI ROY (11IT71P02) to the Indian Institute of Technology Kharagpur, is a record of bonafide research work carried under my supervision and is worthy of consideration for the award of the Master of Science of the Institute.

Signature of the Supervisor(s):

Prof. CHITTARANJAN MANDAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
AND
SCHOOL OF INFORMATION TECHNOLOGY

Indian Institute of Technology Kharagpur
Declaration by the Student

Date: 10-10-2014

Title of the Thesis:

Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization

I certify that

a. the work contained in the thesis is original and has been done by me under the guidance of my Supervisor;

b. the work has not been submitted to any other Institute for any degree or diploma;

c. I have followed the guidelines provided by the Institute in preparing the thesis;

d. I have conformed to ethical norms and guidelines while writing the thesis and;

e. whenever I have used materials (data, models, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis, giving their details in the references, and taking permission from the copyright owners of the sources, whenever necessary.

Signature of the Student: ____________________________________

Name of the Student: GARGI ROY (11IT71P02)

Acknowledgement

I would like to convey my sincere gratitude to Prof. Chittaranjan Mandal for being my supervisor and guiding me throughout my research work. Without his help and encouragement this work could not be accomplished. I am really thankful to him for all I have learned from him. I am also thankful Prof. Dipankar Sarkar and other professors of the Computer Science Department and School of Information Technology Department, IIT Kharagpur, for their valuable and motivating teaching during my entire course work. I would like to convey my acknowledgement to the technical staffs of both the departments for their spontaneous help and also thanks to Dr. B. Hemalatha for being a good source of inspiration.

I am grateful to be a part of the stimulating environment provided by the fellow research scholars at IIT Kharagpur. Specially, I am thankful to Devleena for her help and motivating technical and nontechnical discussions. Technical discussions with Tamal and Kunal were also very helpful. I am also grateful to Srobona-di for her help regarding using the CUDD package. It was a great fun to have energizing and supportive friends and seniors like Subhadip-da, Ranita, Tanwi, Partha-da, Antara-di, Sudakshina-di, Aritra-da, Chandan-da, Tuhin and all other lab mates. I would also like to convey my thanks to Sujeet for sharing his laboratory assignments accomplished using our tool.

Last but not the least, I am deeply thankful to my mom, dad and masi. Without their love, understanding, support and patience this thesis would not get a real shape. I also want to thank Roshni for her valuable friendship. Finally, I would like to convey deep gratitude to Dr. Debashis Ghosh and Shibendu Lahiri for their inspiring support.

Gargi Roy

Abstract

This work presents some techniques and algorithms to support teaching of logic design and computer organization through developing a web based virtual laboratory (COLDVL) and a formal verification method of bit-level equivalence checking for automatic evaluation of student designs. At the heart of the virtual laboratory is the COLDVL tool equipped with a circuit drawing and experimentation interface as a front end, a logic simulator as the back end with features to provide real laboratory like learning experience and a set of pre-designed guided experiments with the facility to add new experiments.

In the front end of the tool, a repertoire of components and design functionalities for building circuits are made available. Features to aid learning include the Huffman structure identification in a circuit, detection of possible race around condition prior to simulation, automatic generation of controller unit from a given control state chart to be used in a circuit in association with a data path and a case based analysis for determining indeterminate signal values of some nets in gate level memory elements after regular round of simulation. Creation and reuse of user-defined encapsulated hierarchical modules, structural verilog netlist generation and saving user circuits with unique identification to check plagiarism are also supported. While circuit simulation is generally carried out in an event driven manner, a more efficient simulation technique has been devised for circuits conforming to the Huffman model. Laboratory experiments using COLDVL are conveniently conducted on a regular desktop or laptop computer.

Automated checking of student assignments through application of formal verification is a novel feature of this work. A new bit-level equivalence checking method has been developed for this purpose to compare the designs submitted by students against a reference design provided by the instructor. A submitted design may differ from the reference design in non-trivial ways but may still be perfectly acceptable. The aim of the equivalence checker is to determine conformance to the reference design despite the differences. Bit-level arithmetic, logical and shifting operations, conditional branching, etc. are handled along with computer arithmetic algorithms (eg. multiplication and division) that build the result through stages involving shifting.

The developed virtual laboratory has been successfully used in undergraduate and postgraduate laboratory courses in IIT Kharagpur.

Keywords: Virtual Laboratory, logic simulation, logic design, computer organization, equivalence checking, finite state machines with data path, bit-level equivalence checking

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Logic Design and Computer Organization are typically a core course among the electrical and computing science disciplines. This subject requires strong laboratory support for a wide variety of experiments necessary for developing essential concepts for this subject. An appropriate laboratory would be well equipped with a variety of IC chips, experimental setups (having LED indicators, DIP switches, clock and signal generators, regulated power supplier, digital oscilloscopes etc.). Access to equipments in such a laboratory is typically regulated. The regulated access to the hardware laboratory may get influenced by several factors such as, chips being expensive, chips and equipment depreciating with usage, limited availability of timetable slots for the use of the laboratory and inadequate availability of laboratory instructors throughout the year. Also, small colleges in developing countries may not even have adequately equipped laboratories. These circumstances throw up challenges for satisfactory conduction of a laboratory to support logic Design and Computer organization. A virtual laboratory environment dependent on limited Internet access and a personal computer to conduct relevant experiments in a way that is similar to a conventional laboratory but without the associated encumbrances is an inherently attractive solution.

Apart from the conduction of experiments, their evaluation is another important aspect, to track student learning. Manual evaluation, which is currently, the main mechanism is slow, manpower intensive and limited in availability. Simulation based evaluations have limited ability to uncover various problems. Small and simple circuits may be easily simulated for evaluations, but larger circuits are harder to test. Generation of the test plan, is in itself, a challenge. Because of these limitations of simulation based evaluations, an automatic evaluation technique would be desirable.

We now introduce some terms that are related to the work reported in this thesis. Thereafter, a survey of related literature is presented, leading to motivation and objective of this work. This chapter is then closed with a description of the contributions of the work done and an organization of the rest of the thesis.

1.1 Logic Simulation

As the term simulation implies to mimic a real word scenario, it is extremely important to understand the functional behaviors of the entities which are being simulated. For logic simulation, which is used to simulate the operational behaviors of digital circuits, it is necessary to correctly schedule simulation in addition with modeling the behaviors of gates, nets, inputs and outputs. The logic simulation technique can use different logic values for the signals in order to represent *true*, *false*, unknown states, error conditions etc. Following are some of the widely used simulation techniques.

Event Driven Simulation: In event driven simulation [13], events are defined in terms of change in signal value in a net. Whenever a change in net value is detected the simulation of gates are scheduled and this scheduling is done dynamically that it can not be predicted before hand while parsing the circuit. This simulation is being used in simulators for its elegance in selective trace approach which simulates only the active components which are affected by the change in the net value. Simulation of synchronous, asynchronous circuits and timing analysis are also being handled in this simulation. However, in spite of its elegance of the approach, the approach has a major drawback. As, not all the events produced by the evaluation of active components are necessary to produce useful output this approach generates some unnecessary events thus degrading the performance of the simulator.

Topologically Ordered Levelized Simulation: In literature it is also known as levelization simulation. In levelized simulation [60], all the components are given levels to ensure that whenever a component is simulated all of its inputs are available. This approach is very efficient as it does not generate unnecessary events and does not employ event management. However, this approach has a severe restriction because of its levelization process, it works correctly only for acyclic circuits i.e. combinational circuits. Whenever a circuit contains loop, in case of sequential circuits with memory elements, this simulation fails.

1.2 Formal Verification

In formal verification, a relationship is established between the specification and the implementation. The specifications, implementation and the relationship between them, all these three entities are represented in a formal way so that formal analysis can be carried out. In the context of formal verification of hardware designs, a hardware is modeled formally, often as a state transition system, its behaviors are specified in terms of properties and finally, it is automatically checked whether the model of the hardware satisfies all the specified properties or not. Formal verification ensures that for a set of given specifications, its corresponding implementation satisfies the given specifications. The implementation is the actual hardware design for the given specifications i.e. intended behaviors which the design should satisfy. The notion of satisfaction is defined in terms of holding a formal relationship between the description of the desired specifications and its implementation.

To ensure that the satisfaction relationship between the specification and the implementation holds good, various notions are used in the formal verification domain. Following are few types of formal verification technique.

Theorem Proving: The relationship between a specification and an implementation is regarded as a theorem in logic. Using a proof calculus, the theorem is to be proved. The axioms and assumptions which the proof procedure uses are obtained from the implementation. For most of the cases the logic which is used for the theorem proving goes beyond propositional logic and due to their undecidability, a completely automated theorem prover can not be used. More details can be found in [30], [37].

Model Checking: In model checking [30], [9], the specification is defined in the terms of logic formula. Validation of the logic formula is then determined with respect to a semantic model obtained from the implementation. Although this is a widely used formal verification technique, the automated circuit evaluation is more of establishing behavioral equivalence.

Equivalence Checking: The equivalence of a specification and an implementation is checked. This is a restricted form of property checking between the models representing the specifications and its corresponding implementation. Further details can be obtained from [9].

The model used to represent the expected behavioral specifications and its implementation has a significant rol in the formal verification method. The present work developed for automatic design evaluation, uses FSMD (finite state machines with data paths) as the model to represent the specifications and the implementation. The FSMD is a universal specification model [21] which can represent all hardware designs. In FSMD, all the data storage and data transformations/status detection circuits resides in the data path and the sequential behavioral aspects of the circuit are taken care of in the control path of the circuit. Starting from the initial state, the control path invokes signals which set up paths for the register transfer operations in the data path as specified in the behavioral specification. The results of these operations are available to the control path through certain status outputs of the data path. depending upon the states of these lines, the control path determines the next state. The entire data path state space is partitioned by some data predicates captured by the status output lines.

1.3 Literature survey

Typically a virtual laboratory consists of a set of pre-designed experiments along with a simulation platform to conduct the experiments. There are many virtual laboratories available for different subjects, however, the available literature for the virtual laboratory in the domain of computer organization along with digital logic is limited. A virtual laboratory is presented in [49] for digital logic course which is an animated environment with textual tutorial links, demonstration movies and interactive modules for practice. The virtual lab can be run as a stand alone application or through a

Web browser having a simulator incorporated in it for conduction of designed lab sessions. As the simulation platform is a key component of a virtual laboratory, a brief literature on the simulation platform has been presented in the next few paragraphs.

Although there are many simulators are available these days, not all are appropriate for educational purpose. While designing a course, the teaching instructors have a set of pedagogic goals which must be achieved through the course. This section includes a brief literature survey on logic simulators suitable for educational purpose on logic design and computer organization. This section also includes a brief background on formal verification techniques for automated evaluation of student designs.

1.3.1 Logic Simulators

Literature includes a vast set of simulation tools in the domain of logic design and computer organization. [40] and [24] nicely presents a survey on the simulators in the above mentioned domains including a list of simulators. The simulators can be classified into two broad groups. One group of simulators allow users to build their own circuit, simulate and reuse their circuits. For example, SLEEP [44] is a general purpose discrete event simulator where new components can be created using graphical or textual editor. JHDL [2] is a "structurally based Hardware Description Language (HDL) implemented with JAVA" that allows users to define circuit using high level language. It also supports debug and test simulation of circuits using their set of FPGA CAD tools. HASE [1] allows users to create hierarchical modules, configure and simulate them. Where as, another group of simulators give only the facility of simulating already built systems. Users can perform testing on those built-in systems with different test configurations but they can not build their own system and simulate them or perform testing with different input configurations. Most of the available simulators belong to this group.

Logisim [16] has simple interface that facilitates the design and experimentation process. The tool interface uses colored wires for different logic value that are used in the simulator. Three different colors are used to indicate three different logic values. Horizontal and vertical wires are drawn and automatically connected to other components and to other wires. Design components includes basic gates, tristate gates, flip-flops, input-output components etc and do not contain other complex components. Although the tool facilitates the hierarchical design through the use subcircuits, the subcircuits can not be saved as user designed modules which can be saved and reused in other designs. Logisim does not include bus to support bus based design, circuit timings and moreover it does not deal with simulation efficiency. Due to the design limitations, it is not suitable for bigger circuits with 16-bit data or more. In Logisim, a circuit is simulated whenever a wire is connected or any change in input propagates the value through the circuit instantly. This simulation approach has been tested and found to be failed in case of building master-slave JK flipflop with NAND gates in a specific order which has been discussed in the section 4.1 Although the case is rare but it can be manifested during experimentation.

JLS [42] is another educational simulation tool suitable for digital logic designs along with potability feature. The tool includes a graphical editor for creating and editing logic circuits and then simulator simulates the circuits over a period of time. JLS contains basic gates, tristate gates, decoder, multiplexer, adder, registers, SRAM, ROM and component connecting mechanism along with designs with subcircuits. Truth table editor is used to specify an arbitrary number of single-bit inputs and outputs. The truth table supports logic value *true*, *false* and *don't care*. The simulator then simulates the effect of the equivalent logic instead of generating actual gates and wiring to realize the circuit. This tool also provides a state machine editor where user can specify Moore type machines and can see the behavior of that machine. Event driven simulation is used in order to simulate circuits. Although JLS has facility to generate signals from specified state chart, it can not be used as a controller component along with a data path to build complex circuit designs. For evaluation purpose, the tool provides a feature to simulate student circuits in batch mode where simulation runs in back end without the GUI appearing and a final text-based output is generated. However, the evaluation technique is based on simulation (for limited combinations of inputs) which does not guarantee the design correctness with the given design specifications.

The SLEEP [44] is another simulation tool having a graphical interface and a list of components using which users can build and simulate digital circuits. Parameterized components are also provided where user can instantiate a component and configure its several parameters such as size, capacity or other component specific parameters. For example, gates can be instantiated with user given input pin numbers. The tool also supports the hierarchical design by allowing users to create their own component and reuse them later. There are facilities to build the components either using the graphical interface or a given textual editor. The textual editor allows user to build components in a behavioral level by specifying the component behaviors through Java programming language. On the other hand, in the graphical editor components can be created on a hierarchical structural level. From the simulation technique aspect, SLEEP supports multiple simulation execution algorithms including single thread, multi-thread and distributed. The simulation is carried out interactively either event-by-event or multiple-events. However, it does not support wired AND operation as it uses three levels of values, in order to support bus based design.

Apart from the simulators discussed above, there are some simulators which focus on some specific topics. These tools provide parameterized components for testing the behavior of the component or set of components with different configurations given by the users. Users can change the parameters of the components and simulate the corresponding effect. However, users can not build their own hierarchical designs with these kinds of tools. Following paragraphs describes some of the tools of this category which can aid learning for specific topics from the curricula of logic design and computer organization.

SpimVista [41] is a program-driven simulator which allows the students to visualize the working interactions between two levels of cache memories on a graphical interface. The simulator allows users to specify different parameters such as cache size, replacement algorithm, write-hit pol- icy, write-miss policy for the hierarchical cache memories and simulates the cache behaviors accordingly. It shows performance statistics for each cache level along with the cache control information and block content for a given configuration. These features for testing cache memories with several policies and cache sizes help students to analyze the best set of cache parameters for a given workload.

[59] introduces a virtual CPU with user defined control unit which can be hard-wired or micro-coded. Students are expected to define the control unit's circuits as Boolean formulas or define the memory array and then test the CPU by loading RAM with the machine code and data. It is mandatory that student designs must support a machine language specification. After defining the controller and loading the memory students can initiate the execution of their loaded program through simulating the virtual CPU and can check how the microcode instructions interpret the machine program.

[31] presents a visualization tool for Booth's multiplication algorithm along with an online text-book. Students can visualize the operational flow of the algorithm on different inputs. The inputs can be in terms of value or register size. The input values can be either user defined or randomly generated. Once the operations of the algorithm is started on the inputs, it can be paused and move to previous or next steps for better understanding. This tool runs in an interactive manner with randomly popping up questions for self-assessment. The tool also includes performance issues of the Booth's multiplication algorithm and justification for its correctness.

However, no simulator has been found in the literature to provide all the features supported in our simulator which we believe to be important to aid learning. The features include support for case analysis technique and simulation while reconstructing a circuit in an ordered manner, Huffman structure identification in a circuit, possible race around condition detection in a circuit before simulation, control signal generation tool from user-specified ASM chart and usage of the automatically generated controller in bigger circuits, bus based design with wired AND operation, structural verilog netlist generation and saving circuits with user identification. Therefore, we have designed and developed our own simulator supporting all the aforesaid features including the facility of creating and reusing encapsulated user module along with providing an efficient simulation platform.

1.3.2 Bit-Level Equivalence Checking

In [50], an equivalence checking method for RTL descriptions that implement arithmetic computations (such as ADD , MULT) over bit-vectors with finite widths is presented. In this method the bit-vector arithmetic is modeled as algebra over finite integer rings (where the bit-vector size (m) dictates the cardinality of the ring (Z_{2^m})) as a bit-vector having size m represents integer values from 0 to $2m - 1$ and the corresponding integer values can be reduced to modulo 2^m ($\%2^m$). Thus the equivalence checking problem is reduced to checking equivalence between two polynomial functions from $Z_{2^n}1 \times Z_{2^n}2 \times \cdots \times Z_{2^n}d \rightarrow Z_{2^m}$. They formulate the equivalence problem $f \equiv g$ into proving whether $f - g \equiv 0 \ \% \ 2^m$. This formulation corresponds to that of testing for membership in the ideal of all vanishing polynomials over the given finite ring. Such vanishing ideals have been analyzed and we have derived efficient algorithmic approaches to test whether or not a polynomial is a member of this vanishing ideal. However, this approach can not handle right-shift operations which occurs in circuit designs.

The equivalence checking method presented in [54] for arithmetic circuits at the arithmetic bit level includes a boolean mapping algorithm that extracts the network of half adders from a gate netlist of an addition circuit to get an arithmetic bit-level representation of the circuit. Equivalence checking of the two circuits is performed on their arithmetic bit-level representation using simple arithmetic operations. The extraction of half adder networks as the smallest arithmetic units from an arithmetic circuit is possible in many circuits because arithmetic operations such as addition, subtraction, multiplication, division are achieved through the addition operation. However, they do not identify word operations. They only deal with bit operations and among several bit operations they identify only addition operation performed between single single bits. The equivalence checking is performed on the arithmetic bit-level representation of the circuits using commutative and associative laws. Although their technique can also establish equivalence in multiplier circuits employing carry-save-addition scheme, Wallace tree, the major limitation of their approach is that it is restricted to the combinational circuits and thus it can not handle circuits with delay elements, shift operations and control information.

The equivalence checking methods presented in [33], [32] can account both the data path and control flow of a behavior. They model the behavior as FSMD (finite state machines with data paths) which is a universal specification model [21] capable of representing all hardware designs. In FSMD, all the data storage and data transformations/status detection circuits resides in the data path and the sequential behavioral aspects of the behavior are taken care of in the control path of the design. The proposed equivalence checking methods are mainly path cover based approaches where each behavior is decomposed into a finite set of finite paths and equivalence of behaviors are established by showing path level equivalence between the two behaviors. Changes in structures are handled by extending the path segments in an FSMD. The methods can also establish the equivalence between two behaviors with a change in control structures and capable of verifying different code motion techniques. They use a normalization technique, adopted from [46], to find equivalence between structurally dissimilar but equivalent arithmetic operations. The normalization process actually reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure. However, in case of handling code motions, as a path segment cannot be extended across a loop, the methods fail in the case of code motions across loops, when some code segment before a loop body is placed after the loop body, or vice-versa. A value propagation based equivalence checking method is proposed in [10], [11] can efficiently handles the code motion across loops. They also model the behaviors as FSMDs adopts the path cover based approach including the normalization technique to determine equivalence among different arithmetic operations and behaviors with different control structures. Code motions across loops is verified through propagating the variable values through all the subsequent path segments if mismatch in the values of some live variables is detected. Repeated propagation of values is performed until an equivalent path or a final path segment ending in the starting state is reached and if the propagated mismatched value is not resolved after all possible propagation then non-equivalence is detected between the behaviors. The variables whose values are propagated beyond a loop must be invariant to that loop for valid code motions across loops. The loop invariance of such values can be ascertained by comparing the propagated values that are obtained while entering the loop and after one traversal of the loop. However, the normalization technique used in the equivalence checking methods presented in [33], [32], [10], [11] is restricted to the arithmetic operations over the integer domain and thus the methods are also restricted to the integer arithmetic operations and can not handle the bit-level operations such as bit-level arithmetic operations, logical operations, shift operations and some other operations such as concatenation, extraction in a design. Moreover, the use of infinite precision of data, these methods fail to identify overflow/underflow of the arithmetic operations in the bit-level.

1.4 Motivation and objective of the present work

As computer hardware is the fundamental to any types of computing, it is necessary for the students to know the basic internal working principles of a computer. The important concepts regarding the hierarchical functional units of a computer and how the units communicate with each other along with their performance issues and their practical implications must be understood. Understanding the basic low level system detail also helps writing efficient computer programs considering limited scopes of hardware, memory issues etc. Digital logic is the fundamental theory on which hardware is built independent of the implementing technology. It reveals different layers of abstraction when complex and larger modules are obtained from smaller modules. It also gives theoretical support how to implement logic functions in hardware in several ways including the optimal implementation. Digital logic also introduces the basics for the issues such as timing, spacing, performance, consistency, stability, reusability, design trade-offs etc. which are more exposed to the students while teaching computer organization. Therefore, learning computer organization is deeply associated with the learning of digital logic which involves applying the theory through laboratory experimentation and then evaluation for design correctness. With the advancement of current simulation technology, laboratory experiments can be virtually accomplished on software tools without employing overheads and limitations in experimentation in real hardware laboratories. Usage of simulation tools for experimentation gives more ability to build and edit hierarchical designs, different behavioral analysis. While industrial grade simulation tools are expensive having steep learning curves and are generally used by experienced designers. On the other hand many of the free tools available provides limited functionality. Therefore we have developed a virtual laboratory equipped with a newly developed tool having a design editor at front end with a rich set of design functionalities and a high performance back end logic simulator with features to aid learning and a set of guided experiments. The motivation of developing a tool with graphical user interface over text-based tool was to provide a easy circuit debugging feature. Because in a text-based circuit specification environment finding missing and wrong connections, tracking value propagation through the circuit is quite hard for students new to the area of study. Apart from providing a generic platform for the experimentation the evaluation of the performed experiments are also very important. Manual evaluation can be accomplished for simple experiments, however, sometimes manual evaluation becomes impossible for complex circuits. Simulation can be one way to check design correctness but only for restricted set of inputs. Therefore, an automatic evaluation technique is required when design specifications are needed to be verified without simulation which is a big challenge. As a result, formal verification methods are required to validate a user design against a given set of design specifications. This work also describes an automatic design evaluation technique for checking design correctness to address the shortcomings so identified. Objectives of this work broadly outlined as follows along with the summary of the work in the next subsection.

1. Designing the web interface, the front end of the simulation platform and the flow of learning activities applying commonly accepted pedagogic principles and to analyze the effectiveness of their application by gathering and analyzing feedback from prospective end users

2. Designing and developing an efficient simulator which will provide the student with nearly the same experience of a conventional laboratory and aid his learning of the subject

3. Developing an automatic evaluation mechanism of user designed circuits

1.5 Contributions of the present work

Contributions of the current work are briefly outlined as follows.

1.5.1 Designing the virtual laboratory

- The design of *Computer Organization and Logic Design Virtual Laboratory* (COLDVL) is based on some pedagogic considerations which are assimilated through designing the web interface of the virtual laboratory, a set of pre-designed experiments based on a concept hierarchy (derived from various text books), a sequence of learning activities based on some commonly accepted pedagogic principles.

- COLDVL is being deployed in several colleges and institutes and satisfactory feedback from prospective end users are gathered and analyzed indicating the effective assimilation of the pedagogic considerations. The COLDVL is also been used in IIT Kharagpur, to conduct the undergraduate and postgraduate level laboratory course.

1.5.2 Development of front end features of COLDVL tool

- A design editor is developed containing repertoire of components (several building blocks and reference designs, input-output components, different types of adders and combinational components, sequential components) along with a rich set of design functionalities to enable students to design their own hierarchical circuit modules.

- For complex experiments, such as CPU design, the tool provides an interface for specifying the controller abstractly as an algorithmic state machine (ASM) and another interface to load binary program to a working memory to check the behavior of the designed CPU.
- The features which are developed to aid learning and teaching includes creating and reusing user-defined encapsulated modules, saving circuits with unique user design to check plagiarism, structural verilog netlist generation, reduction of design complexity of hardwired control unit through automatic control signal generation from user-given ASM chart, detecting the existence of possible race around conditions in the circuit prior to simulation to help students learning a safe circuit design, ease of error analysis using different colored wires in a circuit.

1.5.3 Development of back end features and techniques of COLDVL tool

- A five valued logic is employed in the simulator to support tri-state logic, wired AND operation for a bus based design. Simulation dynamically changes the visual aspect of the circuit and delivers the final result.
- The logic simulator of the COLDVL tool uses an efficient simulation algorithm which is a combined approach of general event driven simulation [13] and levelized simulation [60] to achieve better simulation performance for sequential circuits conforming to the Huffman model [29].
- In order to apply the efficient simulation algorithm, the conformance of a circuit with the Huffman model is checked by identifying the structural storage elements in a netlist to help partition the netlist into a combinational cloud and the sequential storage components in a gate level circuit.
- Use of structural master-slave flip flops is identified, as a way to warn for the possible presence of race around conditions (when non-master-slave storage elements are used).
- For simulation of pure combinational circuit, levelized simulation technique [60] is used and sequential circuits not conforming to the Huffman model are simulated with regular event driven approach.
- During simulation, some nets driven by flip-flops may assume the logic value unknown (indicating either 0 or 1) as the default initial value. Under some circumstances, the output of the overall flip-flop may assume a specific 0/1 logic value nevertheless. Standard event driven simulators are found not to account for this possibility even though these circuits should produce definite outputs (for the given inputs) on bread board. This results in a discrepancy between results observed in real circuits versus simulation output, creating a learning gap.

Our simulator does a more intelligent case based analysis and resolves the output to specific logic values that would be actually assumed.

- When resolution by case analysis is not possible, specific desirable outputs are obtained by simulating a sequence of netlists that would result if the flip-flop were to be reconstructed gate by gate. In such a situation, our tool provides a better learning experience for novice students.

1.5.4 Development of automatic design evaluation technique

For evaluation of student designs, a new path based bit-level equivalence checking method is developed having the following aspects.

- In the proposed bit-level equivalence checking method, the arithmetic, logical, bit-wise and shift operations are handled at bit level. It is to be noted that the arithmetic operations over integer domain are different from the arithmetic operations at bit level due to the possible occurrence of overflow or underflow. In addition, two dissimilar bit-level operations may actually be equivalent. The proposed equivalence checker also deals with that problem of identifying equivalent operations from structurally different operations.

- The equivalence checking task is performed between two designs. Two designs are equivalent if the corresponding variables (variables which present in both the designs) contain same value in all possible cases. In some designs, output(s) of an operation is assigned in more than one register, in such case, there will not be any one-to-one variable correspondence . Instead, the composition of those variables should be considered to check value with the corresponding single variable or the composed variable. The current method also handles this situation.

- In some complex designs such as multipliers and dividers, output(s) may be iteratively built bit by bit through different shift operations in a loop. The registers which hold the final results (after loop iterations) may initially hold different values. Moreover, when this situation occurs on composed variables then the equivalence checking task becomes more challenging. In some of the such cases, the proposed equivalence checking method tries to identify the data shifting characteristics in each path of a loop based on some design observations using some predefined rules.

- The equivalence checking is accomplished without unrolling the loops of the designs which may face problems when some operations after the loops uses the bits of some registers whose value can only be obtained after full iteration of the loop. Before the full loop iteration, all of the operand bits may not hold valid data and thus may lead to the failure of the equivalence checking. In such cases, a mechanism is developed to infer the data range after the completion of a loop.

- The equivalence checking method decomposes the designs into path segments and uses symbolic execution of the paths in order to check equivalence. However, in some cases, the equivalence checker does a special analysis to determine the specific value of a symbol in all possible cases through extending the paths across a loop and symbolically execute them to find equivalence.

- The proposed bit-level equivalence checking method also captures the similar operations performed within a design path but in different clock cycles.

1.6 Organization of the thesis

Rest of the thesis is organized as follows.

Chapter 2 (Design Issues of the Virtual Laboratory to support Logic Design and Computer Organisation) This chapter will contain the pedagogic Objectives of COLDVL, achieving of the objectives of COLDVL defined so far, web interface of the COLDVL and the validation of the achievement of the pedagogic objectives.

Chapter 3 (Front end of the COLDVL tool) This chapter will contain the description of the features of the font end of the tool and the brief description of the usability of the tool in post-graduation laboratory course at IIT Kharagpur.

Chapter 4 (Back end of the COLDVL tool) This chapter will contain the description of back end features, techniques developed along with algorithms and results obtained for some test benches run on the tool.

Chapter 5 (Checking Student Designs for Correctness) This chapter will contain challenges addressed, methodologies, algorithms along with illustrative example and finally the implementation and results.

Chapter 6 (Conclusion) A summary of work done and an overview of possible future extensions are given in this chapter.

Chapter 2

Design Issues of the Virtual Laboratory for Logic Design and Computer Organization

2.1 Introduction

This chapter is concerned with the design issues of the virtual laboratory. As learning through the online educational modules have significant dependency on the learners' self-motivation, therefore the design of the educational module should be done very carefully. The module should have a predefined pedagogic considerations and the design of the module must be done in a way that it assimilates the considerations. This chapter describes the pedagogic considerations, the design of the virtual laboratory to assimilate those considerations using some commonly known pedagogic principles. The web interface and the set of pre-designed guided experiments of the virtual laboratory are described. Satisfactory user feedback have also been gathered and analyzed.

2.2 Pedagogic considerations of COLDVL

As a virtual laboratory attempts to assist in learning through technology, it should be designed in a systematic manner so that it can fulfill its basic purpose of being a teaching system. Literature on e-learning shows that effective learning can be achieved by such teaching systems through considering several learning theories as learning theories help defining educational frameworks by considering several parameters which affect the learning process [56]. An important aspect of designing such teaching system is to determine the pedagogic considerations of that system [28]. Therefore, before designing and developing the COLDVL, some pedagogic considerations have been defined which are identified based on the studies on learning mechanisms and e-learning. The following paragraphs describe the set of pedagogic considerations of the COLDVL.

The *IEEE/ACM Computing Curricula 2001* report [57] presents a set of curriculum activities for undergraduate students of computer science. It says that all students of disciplines related to computing must understand the basic internal working principles of a computer. They should have a clear concept regarding the hierarchical functional units of a computer and how the units communicate with each other along with their performance issues and their practical implications. In addition with this, [55], [48] depicts that not only learning the major concepts but understanding the existing relationships among those concepts of a subject contributes to an effective learning leading to better overall understanding of the subject. These issues are collectively addressed as *ordering of learning concepts*.

Learning is a process of acquisition of knowledge which involves several cognitive resources of the brain. Literature includes a vast set of study on mechanisms of learning and several cognitive parameters affecting the learning process [55], [23], [22], [43], [48], [62], [25] etc. Research works show that learning is directly related with the attention paid by the learner to the topic [23], [22]. Existing knowledge of a learner has also been found to have significant impact on learning [55], [18]. Apart from issues such as attention and existing knowledge, application of acquired knowledge through practice has a prime role on the learning process itself [55], [47], [51]. Results of studies on these aspects have been incorporated and evaluated collectively as *cognitive issues for learning*.

As one of the basic goals of a teaching system is to enhance learning [28], three important concepts of learning from [14] have become another pedagogic consideration for COLDVL. The concepts considered are as follows:

- Learning is enhanced when learners are actively involved in the *learning activity*.
- Learning can be promoted by a structured and sequential *work flow* of the learning activities.
- Recording designs of learning for sharing and reusing in future helps to promote learning.

Apart from these issues, achievement of learning objectives in a virtual learning environment are greatly effected by personal motivation [28]. These studies are implemented in COLDVL to enhance learning along with promoting motivation in *learning enhancement through sequence of learning activities*.

2.3 Assimilation of pedagogic considerations into COLDVL

This section describes several design aspects of COLDVL for assimilations of the aforesaid pedagogic considerations using commonly accepted pedagogic principles.

2.3.1 Ordering of learning concepts

[55], [48] depict that understanding not only the major concepts but also the existing relationships among those concepts of a subject contribute to an effective learning. Therefore, the order in which

learners are introduced to the major concepts of a subject has a great importance. As a result, designing experiments for logic design and computer organization virtual laboratory requires a systematic approach. Because, each experiment focuses on some specific concepts and the whole set of experiments builds the understanding of the subject gradually. Learning evolves through each experiment, can follow either top down or bottom up approach. In any one of the approaches, some experiments have prerequisite of some other set of concepts for the clarity of understanding. For example, designing of a basic CPU requires the concepts of data processing, data storage, data transfer to be known in advance. As a result, the experiment for basic CPU design, in either top down or bottom up approach requires to cover the aforesaid concepts in prior, otherwise the design of experiments will fail to achieve step wise learning which is very necessary to develop a good understanding of the relationships among the concepts of a subject. For example, if a set of experiment is designed in order such that the Booth's multiplier is introduced before introducing the concepts of registers, counters, adder/subtracter and as Booth's multiplier multiplier requires registers, counter, adder/-subtracter components, therefore this design of experiments does not yield step wise learning.

To ensure the step wise learning of concepts, a concept hierarchy has been derived from several text books [38], [26], [52], [39] to guide the order of learning concepts, shown in Figure 2.1. This hierarchy depicts which concept must be known before which set of concepts. Each of the nodes of the hierarchy is a concept regarding digital logic and computer organization. The concepts which are bounded with an ellipse are the epochs which essentially means the significant and matured concepts of the subject. The concept hierarchy is directed and learning of any concept in the hierarchy must be followed by learning its parent concepts. The parent concepts which are connected with a dashed edge are not necessary to be known in prior, however, knowing this in advance is helpful. Some abbreviations have been used for the clarity of the diagram, such as, FF and FSM denotes flip flops and finite state machine respectively; HA, FA, RCA, CLA, WTA are for half adder, full adder, ripple-carry adder, carry-look-ahead adder and Wallace tree adder respectively. However, CSA denotes carry-save-addition scheme; combX, BoothX abbreviate combinational multiplier and Booth's multiplier respectively; Add/Sub and ALU is for adder/subtracter unit and arithmetic logic unit respectively. This concept hierarchy ensures the step wise learning of concepts, for example, it depicts that before doing Booth's multiplier a student must know the concepts of adder/subtracter unit, controller design, registers and counters and so for the other experiments. Designing experiments by following this concept hierarchy will also help reducing the creations of confusions associated with a random design of experiments. The set of experiments of COLDVL has also been designed on the basis of this concept hierarchy. The epochs bounded with blue lines are the concepts which is assumed to be known to the students, therefore, no experiments have been designed focusing on those concepts. The next paragraph describes the set of experiments of COLDVL developed using the concept hierarchy which also covers some important topics mentioned by the *IEEE/ACM Computing Curricula 2001* report [57]. Although, the *IEEE/ACM Computing Curricula 2001* report [57] contains many important topics, this virtual laboratory includes the guided experiments on the topics covered in the laboratory curriculum of Indian academics [3]. However, the one of the future work would be including more experiments on some other topics.

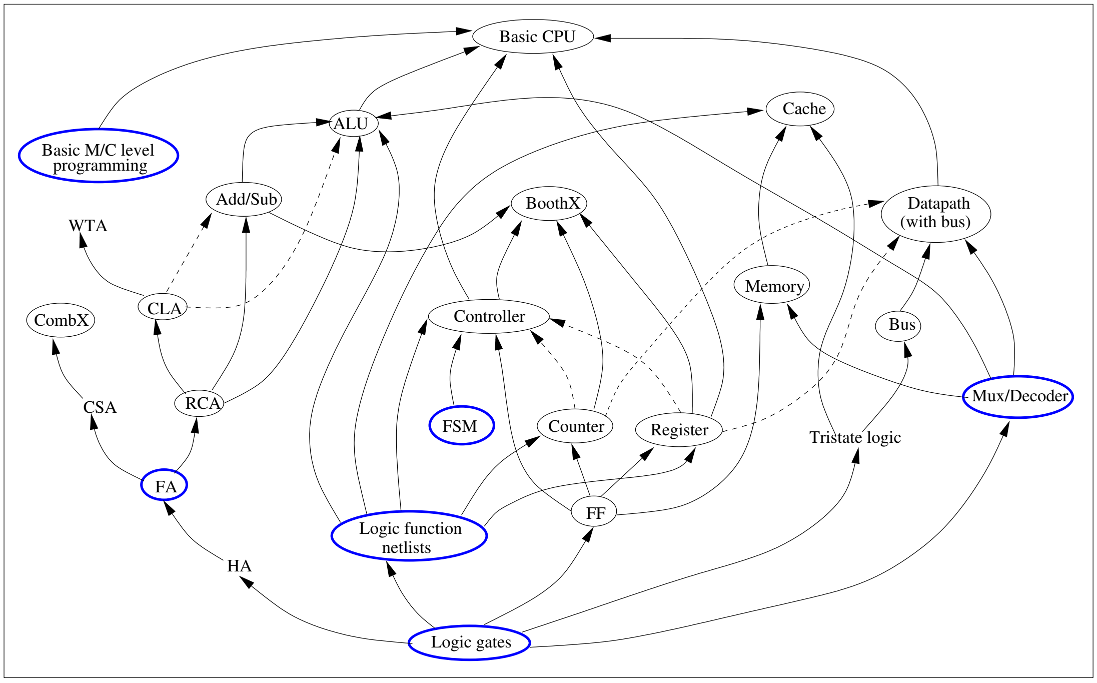


Figure 2.1: Concept hierarchy.

Set of experiments Currently COLDVL contains 12 experiments which covers the important topics of digital logic and computer organization mentioned in the *IEEE/ACM Computing Curricula* reports [57], [58] which includes logic system design, arithmetic unit design, memory design, data caching, control unit design, bus operation, CPU organization. The order of the experiments preserves the dependencies among the concepts depicted in the concept hierarchy shown in Figure 2.1. As the ultimate goal of logic design and computer organization course is to teach the functionalities of a CPU, the goal in the concept hierarchy is also to reach out the basic CPU concept from the bottom level concepts. In the concept hierarchy, the pathways from bottom level concepts to the CPU design concept consist of some moderate number of intermediate concepts and accordingly the intermediate concepts have been covered by the experiments. Therefore, the concepts from the bottom level to the CPU design build through the sequence of experiments. Likewise, not only the CPU, paths from the base to the other experiments, apart from the successors of the assumed concepts, contain intermediate experiment/s. As a result, the experiments gradually reveal the interdependence of an experiment at higher level. All the experiments are necessary as an experiment develops the base for another experiment at higher level except the *Wallace Tree Adder* experiment, because, no experiment has any dependence on *Wallace Tree Adder*. Although cache concept does not hold any dependence for any experiment, it is an important concept for operational performance issues in any standard CPU. *Booth's Multiplier* and *Combinational Multiplier* also do not have dependence edges for CPU in the concept hierarchy. However, these are important concepts for the data paths of a standard CPU. Moreover, *Booth's Multiplier* highlights the design aspects of a system with controller and data path. Therefore, these two experiments are categorized as necessary experiments.

One important aspect of computer organization course is to provide the understanding of the internal hierarchical nature of the subsystems of a computer system. The experiments of COLDVL evolves in such a way that gradually reveals the inter-dependence of the subsystems of a system which itself is hierarchical in nature up to a certain elementary level such as gates and flip-flops. The focus and objective of the experiments are multifold. Where, some of the experiments focuses on a single concept such as data processing, data storage, data caching, CPU organization, some other experiments illustrate the inter-functionality of these individual concepts. Some subsets of the set of experiments also address the performance issues of the alternate designs. The set of experiments includes.

1. *Ripple Carry Adder*, shows how the carry ripples through the adder. The concept hierarchy shown in Figure 2.1 depicts that the concept of carry ripple has dependence on the concept of full adder which in turn is dependent on half adder and finally knowledge of logic gates is necessary to know what a half adder is. This is the first experiment of COLDVL and it has been assumed that students have the basic knowledge of logic gates and full adder. This is a necessary experiment as a significant data processing concept situating at a higher level in the concept hierarchy goes through the ripple carry adder concept.

2. *Carry-look-ahead Adder*, focuses on how computing carries in parallel using carry generate and propagate functions greatly speeds up the overall computation over *Ripple Carry Adder* design technique. The concept of carry look ahead is built on top of the concept of carry rippling in the concept hierarchy. Therefore, this experiment is designed after the experiment of *Ripple Carry Adder* and it is necessary to do *Ripple Carry Adder* experiment in order to accomplish this experiment. This experiment is also categorized as necessary.

3. *Wallace Tree Adder*, helps understanding the concept of reducing gate delay by using tree of adders instead of using cascaded full adders. This experiment focuses on the design of an adder with less height. According to the concept hierarchy, this experiment requires the knowledge of carry look ahead adder, therefore, it is placed after *Carry-look-ahead Adder* experiment. It is not a necessary experiment.

4. *Synthesis of flip flops*, highlights the basic concepts of flip flops as elementary storage units, it also focuses on race around condition occurring in flip flops and the avoidance of race around condition through the master-slave design. This experiment also focuses on some deeper concepts such as inconsistent initialization problem in master-slave JK flip flop design which may cause the flip flop to malfunction. This topic is further elaborated in the subsequent sections. and need for asynchronous preset-clear to resolve inconsistent initialization in master-slave design. This is an independent experiment without having any dependency on combinational adders, although it requires the basic knowledge of logic gates which has been assumed here to be known to the students. However, COLDVL adopts a bottom-up approach for introducing the concepts, as a result, sequential circuits are introduced after developing some familiarity in combinational domain. This is a very important experiment and quite necessary one.

5. *Registers and Counters*, illustrates the sequential aspects of digital circuits such as registers, counters along with their data storage and counting capabilities respectively. Design of registers and counters uses the concepts of flip flops and logic function net lists. Flip flops are introduced in the previous experiment and logic function net lists are assumed to be known to the students. This is also a necessary experiment because many complex sequential circuits most frequently use registers and/or counters in their data paths.

6. *Combinational Multiplier*, shows how to design a multiplier from an array of AND gates, half adders and full adders by unrolling the multiplier loop using the *carry save addition* scheme. This experiment requires the knowledge of carry save addition scheme which in turn, requires the prior knowledge of full adder. The experiment introduces the carry save addition scheme in its theory and as mentioned before full adder is assumed to be known to the students. Although this is a combinational circuit, however, it is placed after two basic experiments on sequential circuit and before *Booth's Multiplier* experiment because it is a multiplier circuit and by performing two consecutive experiments on multiplier, students will essentially be introduced to different design aspects and their performance issues. This is a necessary experiment.

7. *Booth's Multiplier*, focuses on the design of a multiplier to multiply any combination of positive and negative numbers, which is a sequential circuit using Booth's multiplication algorithm where as the *Combinational Multiplier* is purely a combinational circuit. Booth's multiplier illustrates how the data processing unit including data storage units works according to the control signals generated by the control unit to produce the final result. Successful accomplishment of Booth's Multiplier depends on the prior knowledge of adder/subtracter, registers, counters and controller design. Therefore, it is placed after the experiments on adders, registers and counters. This experiment is important because in addition with focusing on the implementation of the multiplication algorithm, it also highlights the controller design and data path design aspects for a sequential circuit.

8. *Arithmetic Logic Unit*, shows the design issues of data processing units. The concept hierarchy depicts that a student must know the basic concepts of logic function net lists, multiplexer/decoder units which is assumed to be known in prior and the adder/subtracter units including ripple carry adder which have been covered in the previous experiments. It also shows that the prior knowledge of carry look ahead adder will be an extra benefit which is also made familiar to the students earlier. This is also categorized as necessary experiment as data processing unit is an important part for the CPU design.

9. *Memory Design*, focuses on the design of single bit and multiple bit random access memories. The concepts of flip flops and multiplexer/decoder must be known before designing the memories. flip flops are introduced in the earlier experiments where as as mentioned before multiplexer/decoder units are assumed to be known to the students. Memory design is necessary experiment.

10. *Associative Cache Design*, shows how data caching with parallel search technique improves performance. This experiment does not incorporate any replacement policy. It illustrates the drawbacks of this design from hardware point of view. Cache memory design is dependent on the concept of memory design which is covered earlier, logic functions net lists which is assumed to be known and tristate logic which is supported by the COLDVL simulator. This is also a necessary experiment.

11. *Direct Mapped Cache Design*, focuses on the another cache design which reduces the hardware complexity over *Associative cache Design* with introducing some other limitation. This experiment also does not incorporate any replacement policy. The concept hierarchy does not further elaborates the cache concept. Even if for the direct memory cache, the dependencies for cache concept hold. This experiment is included to focus on the performance issues between two different designs of cache memory. It is also a necessary experiment.

12. *CPU Design*, focuses on the basic functionality, organization and architecture of a single instruction CPU. This URISC (ultimate reduced instruction set computer) architecture can give a good understanding of the working principles of a computer in a lucid way. A clear understanding of the basic structure and functionality from this architecture will increase insight into the system bottlenecks and alternative designs. Because different cases reveal that changes in the system design have greater effect on the whole system performance rather than changing the individual components of that system [53]. To design a basic CPU a student must know the data transfer and data storage concepts for building data paths, data processing aspects, controller design issues and basic machine level programming. It is assumed that students are exposed to the basic machine level programming and the other concepts are already introduced in the earlier experiments. Although, the design of a basic CPU does not directly dependent on the knowledge of basic machine level programming, it is necessary for checking the functionality of the CPU as the whole purpose of the CPU is to execute programs. Naturally, this becomes a necessary and very important experiment. that increased system capability and fail-safe characteristics are achieved by changing the system design rather than changing the speed and reliability of individual components [53].

2.3.2 Addressing cognitive issues for learning

Attentional issues Literatures on cognitive science include a vast set of research work on attentional mechanism revealing its several aspects including its relation with learning. Attention can be defined as the mechanism of selecting perceptual inputs to be processed by our bounded cognitive resources of the brain among the enormous external stimuli to prevent overloading [43]. [23], [22] address how learning is achieved with the help of attentional procedure. However, lack of attention to the intended topic of learning degrades learning. [61] shows that if the topic to be learned is presented in a complex manner having focus on too many aspects at a time, learners will filter out some aspects due to the limitations of the cognitive resources of the brain. As a result they learn only those aspects which they choose to be attentive to. This selectiveness severely degrades the learning. Hence, a learning environment should be simplified and should have a very specific focus [62]. Theses issues are considered during the design of experiments of COLDVL and specifying the design objectives for each experiment. Each experiment focuses on a particular topic. For example, some of the experiments focus on the single concept such as data processing, data storage, data caching, CPU organization, where as, some other experiments illustrate only the inter-functionality of these individual concepts. Some subsets of the set of experiments address the performance issues of the alternate design. However, no experiment is designed to focus on too many significant aspects. The focus of an experiment is clearly mentioned in the *objectives* of the experiment. Moreover, every experiment contains a guideline to check the key behavior of the design. This guideline has been included in order to draw the attention of the students to the significant aspects of a particular topic.

Addressing misconceptions According to [12], [18], [55] prior knowledge of a learner has a significant impact on learning. If the topic to be learned matches with the existing knowledge of a learner then the learning is likely to be smooth and rapid. However, learners having unaddressed misconceptions tend to memorize the new facts which they may soon forget and less likely to apply them in relevant problem solving. [25] shows that directly addressing misconceptions significantly improves learning. Using this research outcome, COLDVL introduces test plan for every experiment to address some possible misconceptions. As, COLDVL is a virtual laboratory for teaching logic design and computer organization, students are most likely to have a parallel theoretical course on the subject. Therefore, during coursework, students may develop misconception or have inadequate knowledge. Hence, test plans are designed in such a way that they highlight the important features of the experiment topic which may, in turn, address the possible misconceptions about the topic.

Addressing learning mechanisms There are two learning mechanisms, which are acquisition of schema or cognitive construct and transformations of learned procedures from controlled to automatic processing [55], [55], [35], [47], [51]. Schema is the organization of information units in a way which depict how they will be dealt. These schemas provide basic unit of knowledge and these have a significant contribution in how new information will be dealt. After the schema acquisition, the acquired information can be processed either in a controlled way or in an automatic manner in any cognitive activity. Controlled manner require considerable amount of thought or conscious effort, where as, automatic processing occurs without conscious effort. A well learned aspect can be handled in automatic manner. When a concept is first learned which is essentially the schema acquisition and the learner tries to solve a problem using that newly learned concept, that problem solving task requires significant amount of cognitive efforts which can yield slow or erroneous performance. However, with time and practice, as the learner gets accustomed with the subject domain more and more, the skill for using the acquired schemas will get better, which essentially means that attention and other cognitive resources required for the process are reduced and the process becomes more automated. A teaching system for engineering students must focus on this automation process as students in engineering paradigm are expected to solve problems which require ability to apply knowledge systematically and creatively [27]. Moreover, this automation procedure is very important after the schema acquisition as it can affect the whole learning including the schemas. These issues are addressed while designing the web interface of COLDVL. Each experiment interface includes a brief theory along with circuit diagram and reading materials to help constructing the schemas. Moreover, the focus of the experiments of COLDVL evolves from easier to complex topic are designed in a way, using the concept hierarchy mentioned earlier, so that dependency among the concepts are preserved and building of a new schema gets supported by the existing schemas. The web interface of each experiment also includes test plans, guideline for checking behavior, assignments, rich set of quiz questions including multiple choice and subjective questions have been included so that the learning can be practiced from different angle being more accustomed with the subject domain leading to the aforesaid automation procedure. Assignments are designed in such a way that by performing them, a student not only practice their learning but also become more familiar with several design aspects of the experiment topic, which will in turn, improve their problem solving skills while uncovering their innovation. As the simulator provided by the COLDVL gives a generic simulation platform, students can also perform their own experiments apart from the designed experiments and assignments in COLDVL, this gives them enormous scope to nourish their learning at any time.

2.3.3 Learning enhancement through sequence of learning activities

The sequence of learning activities in COLDVL have been designed based on the concepts on learning enhancement presented in [14] which consists of a basic stage for all students and an advanced stage for more curious student. Learning evolves with the learning activities performed in the simulator in each stage. After a successful completion of a stage, students can access their learning through quizzes given for each stage in every experiment. The overall learning grows with the given set of experiments. However, learning is not confined with the given set of experiments as COLDVL simulator gives a generic platform for simulation. For the learners, COLDVL provides an enormous possibility to exercise their understanding, test their innovation and clear their theoretical doubts by practically simulating them. Following is the sequential description of the learning activities associated with every experiment through the stages of learning which in turn, helps students accumulating knowledge and concepts, in addition with uncovering their potential and creativity. The sequence of learning activities is shown in Figure 2.2 where the activities in blue rectangles are the major activities of the stages.

Sequenced learning activities of the basic stage

1. Learners are first recommended to go through the theory, objective and procedure and then perform the experiment on the working module provided in the simulator. As the working module is encapsulated and designed in the form of a chip, its internal circuit design is abstracted from the user, as a result, when a student will successfully test the input-output behavior of the module with being guided by the given instructions for checking key behavior and test plan, it will activate their inquiry arousal and thus attract the attention of the learner

to the relevance of the topic. One of the major goal of any teaching system is to bring positive changes in the motivation of the learner to promote learning. While designing COLDVL, the ARCS motivational model [34], [6] are considered, where learners' attention and topic relevance form the basis of the model.

2. The successful performance of the experimentation on the working module along with the enhanced understanding for the topic, will help learners to increase their self-confidence. Because "being successful in one learning situation can help to build confidence in subsequent endeavors" [8]. And self-confidence is an important factor to increase motivation in ARCS motivational model [34] which essentially signifies that learners will be driven towards the sequential activities of learning to enhance their understanding by performing the same experiment on his own designed modules, self-assessment using quizzes and going through further reading materials.

3. To perform the experiment on their designed modules, students need to build their own circuit with the available basic components in the simulator to perform the same experiment with additional circuit analysis guided by the test plan, checking the value propagation through out the circuit with different wire colors representing five different levels of values, error analysis by seeing the intermediate values of different components irrespective of whether the error is intentional or unintentional. These types of detailed analysis is not possible on the working module. Performing different kinds of analysis on the circuit, students gain deeper understanding on that topic. The success in this stage will again lead to the increase in self-confidence which subsequently drive the learner who is curious, to do the assignments which will lead to the advanced stage of the learning activity.

Sequenced learning activities of the advanced stage

1. More curious students are recommended to accomplish the assignments using their enhanced knowledge on that topic. To complete the assignments they will again go through the steps of building the circuit in the simulator and then testing and analyzing the circuit as stated in the previous stage. The assignments are designed in such a manner that students need to use their understanding of the topic along with their creativity in order to accomplish them.

2. Successful performance at this stage along with the self-assessment and further reading mature their concepts in the topic and encourage them, as stated in previous stages, to save and reuse their work to build larger circuits and systems.

3. After completing the self-assessments, students are recommended not to stick to the topic of the experiment but to consider the other experiments to build a larger system. For example, for experiment *ALU design*, at the advanced stage of learning activity, students can think of building a sophisticated ALU (arithmetic and logic unit) system which includes registers, accumulator, multipliers, adder/subtracter with large number of data bits. This will help students building their knowledge hierarchically.

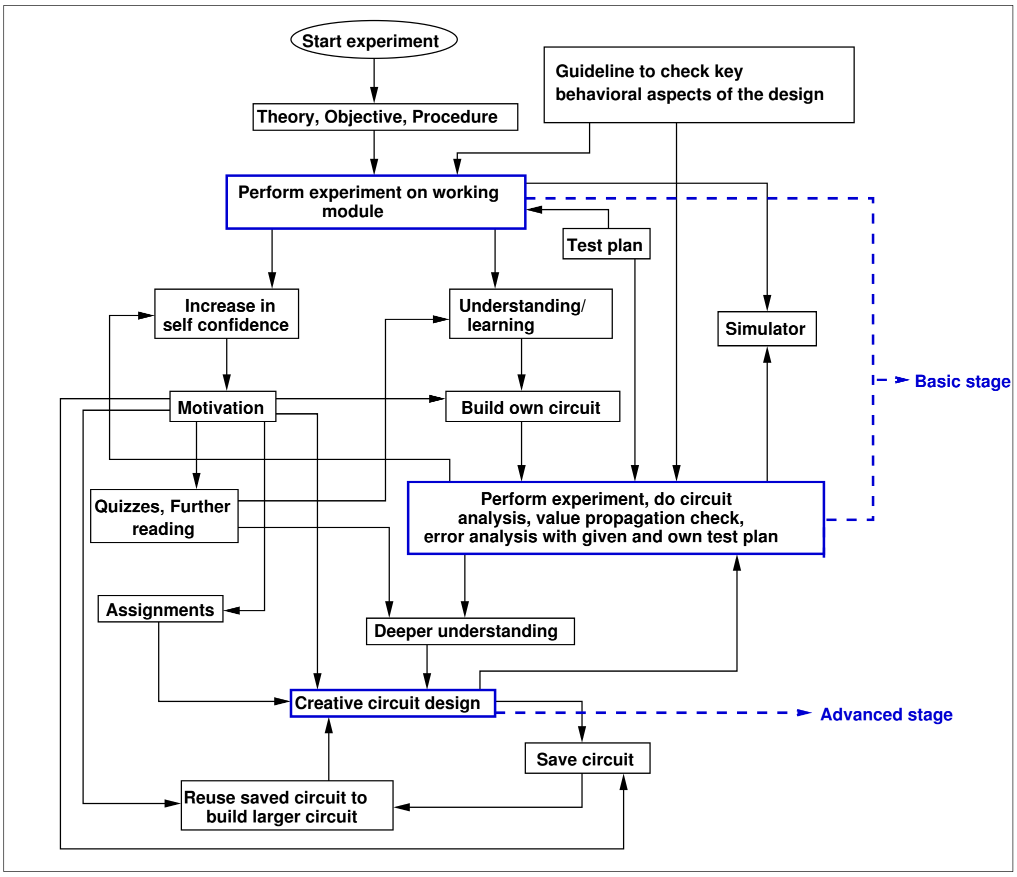

Start experiment
Guideline to check key behavioral aspects of the design
Theory, Objective, Procedure
Perform experiment on working module
Test plan
Increase in self confidence
Understanding/ learning
Simulator
Basic stage
Motivation
Build own circuit
Quizzes, Further reading
Perform experiment, do circuit analysis, value propagation check, error analysis with given and own test plan
Assignments
Deeper understanding
Creative circuit design
Advanced stage
Save circuit
Reuse saved circuit to build larger circuit

Figure 2.2: Sequence of learning activities.

At any stage of the learning activities, the students can collaborate with their remote peers, teachers through the Internet. Students can post their queries and can get answers from teachers or other peers through the social networking. Teachers' involvement to reply the queries through the collaborative learning are insisted to make the learning more effective.

Brief description of learning sequence with an experiment

This paragraph briefly describes the recommended learning activities for the experiment of *CPU design*. CPU designing is an important topic in the computer organization course. This experiment focuses on the basic functionality, organization and architecture of a single instruction CPU. This URISC (ultimate reduced instruction set computer) architecture can give a good understanding of the working principles of a basic computer. As recommended, the student will go through the basic stage of the experiment. He will first go through the theory, objective and procedure and then perform the experiment on the given working modules. The working modules are encapsulated so that student can not see the internal circuit detail of the modules. The CPU built up of the working modules will consists of the single instruction processor module with built in controller. The module have to be connected to the module provided as the working memory which has to be loaded with binary program. The loaded binary program will be executed by the CPU module. Students can

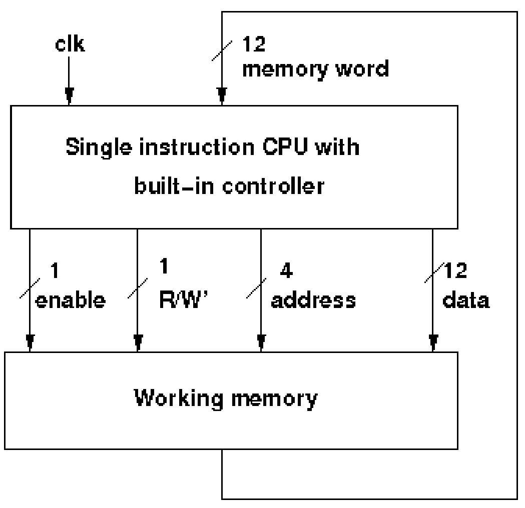


Figure 2.3: Schematic circuit diagram of the CPU experiment on the given working modules in the basic stage of learning sequence.

check the memory content after the program execution. The schematic circuit diagram of the CPU experiment in the basic stage of learning sequence is shown in the Figure 2.3. The two modules have to be connected as guided in the procedure section along with the pin configuration of the modules. Using the guideline, the students will accomplish the experimentation in the basic stage. In this stage, the student can only check the input-output behavior using the test plan given in the web interface of the experiment.

A successful experimentation will increase the motivation level and the student will design his own modules from the provided basic modules such as basic gates, input-output units, multiplexers, tristate buffers, adders/subtracter, comparators, registers etc. in the tool to build his own CPU circuit. A controller has to be designed in order to build the CPU. The circuit will also contain a working memory to load with an input binary program. figure 2.4 shows such a circuit designed by the students before simulation. After building the CPU circuit and loading the binary program, the student instantiates the simulation and checks the memory content which is shown in Figure 2.5. In this phase, the student gives several input binary program to see the behavior of the CPU. With the circuit analysis and checking the value propagation through the circuit, the execution of the program is observed which then provides a clear understanding of the working functionality of the CPU. State transitions of the controller of the CPU is seen dynamically during the program execution for better understanding which is shown in the Figure 2.6. In this stage students will accomplish the quizzes for self evaluation and go through the recommended study material for further knowledge. After the experimentations of the basic stage student will gain sufficient confidence, concept and motivation to perform the advanced stage.

In the advanced stage, the interested student will accomplish harder assignments given in the web interface. He will then design CPU with multiple instructions with increased design complexity. Figure 2.7 shows such a CPU built by the students with four instructions with a more complex controller state diagram and the memory content is also shown. Quizzes are provided for this stage

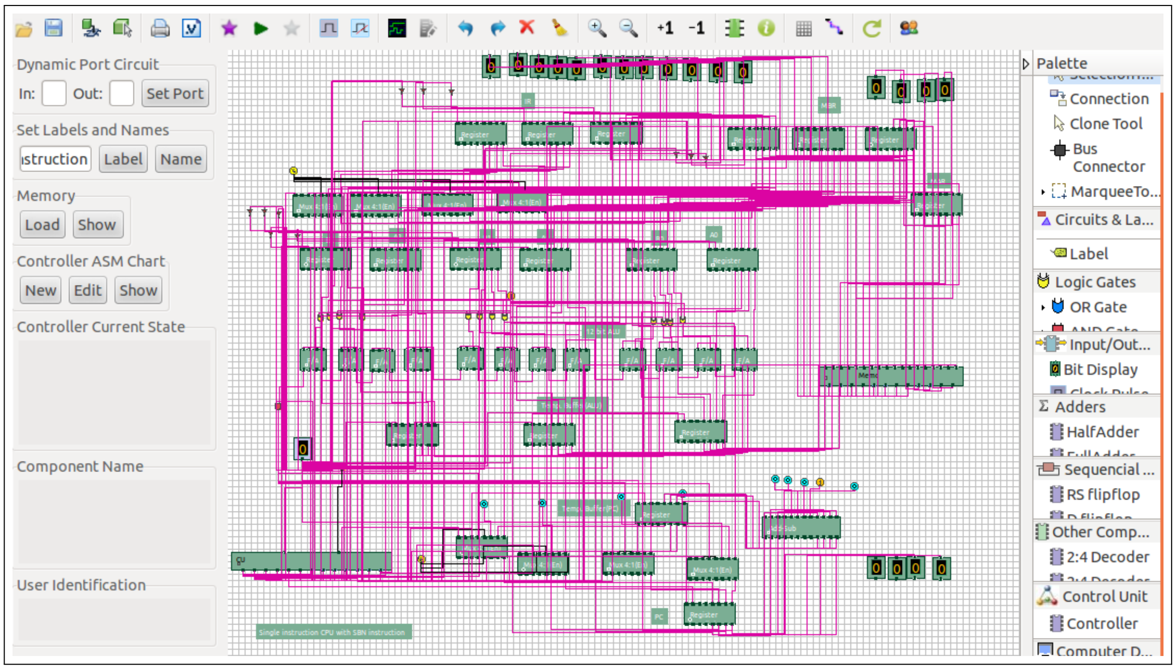


Figure 2.4: User designed single instruction CPU circuit before simulation in the basic stage of learning sequence.

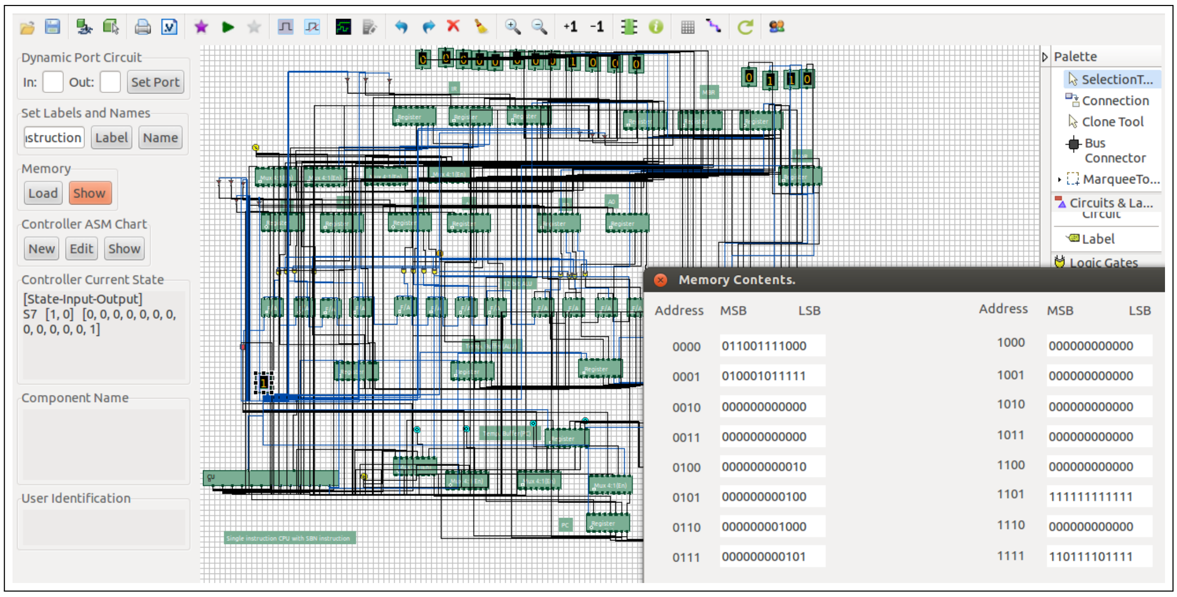
equipped with harder questions so that students can track their knowledge levels and subsequent progress by further going through the reading materials. In this stage, the student can apply his innovations and design the data path and controller of the CPU so that it can function efficiently.

2.4 Web interface of COLDVL

The way pedagogic considerations of COLDVL are assimilated which is described in the previous section, are reflected in the web interface of COLDVL. Figure 2.8 shows a snapshot of the web interface of COLDVL where as Figure 2.13 shows the organization of the interfaces of the COLDVL. It contains a set of guided experiments, a logic simulator to perform the experiments, a text manual for the simulator and some other components such as target users, courses aligned with the COLDVL, prerequisite software, frequently asked questions, overall objective, feedback etc.

Every experiment consists of the following tabbed sections.

- *Theory* section contains a brief description of the theoretical aspects about the topic which the experiment focuses upon.
- *Objectives* section specifies the clear objective of the experiment. It highlights the special characteristic of the theoretical aspect for which the experiment has been designed. It also gives a guideline to examine the behavior of given working module for the experiment as well as the module designed by the student for that experiment. Figure 2.9 shows the web interface of the objectives of the *Direct mapped cache* experiment in COLDVL. It includes



[State-Input-Output]
S7 [1, 0] [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

Address	MSB LSB	Address	MSB LSB
0000	011001111000	1000	000000000000
0001	010001011111	1001	000000000000
0010	000000000000	1010	000000000000
0011	000000000000	1011	000000000000
0100	000000000010	1100	000000000000
0101	000000000100	1101	111111111111
0110	000000001000	1110	000000000000
0111	000000000101	1111	110111101111

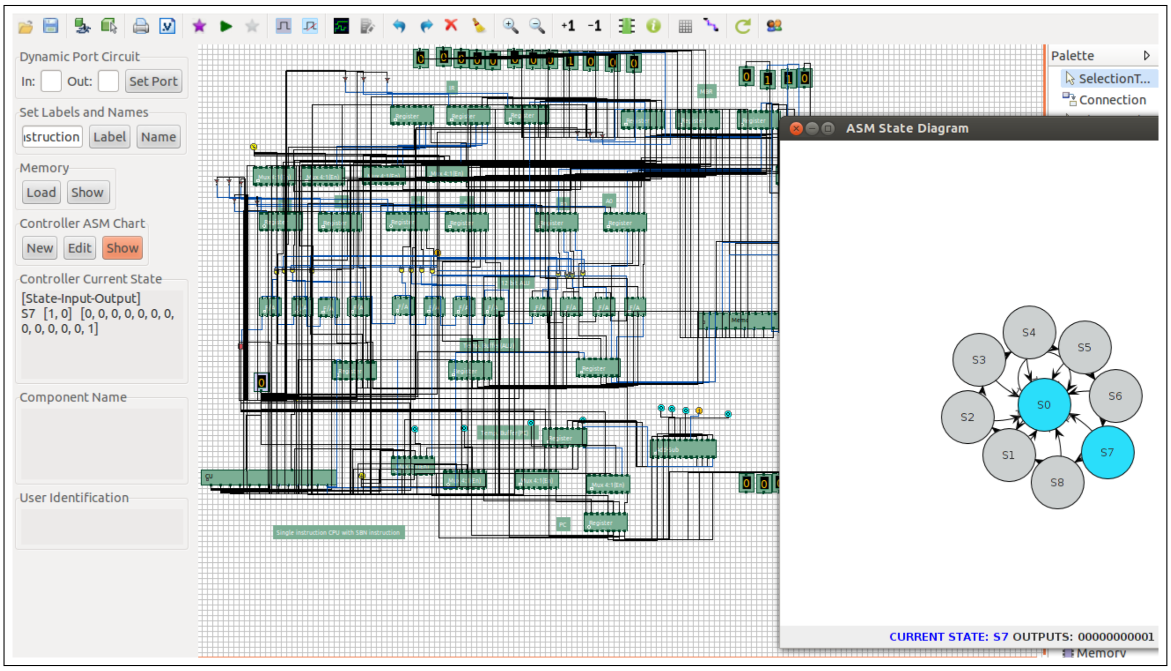
Figure 2.5: Memory content of the user designed single instruction CPU after simulation in the basic stage of the learning sequence.

the following parts (web interface of the test plan and assignments of the *Direct mapped cache* experiment in COLDVL in Figure 2.10).

- Test Plan includes several issues for design and analyzing different input-output behaviors of the system.
- Assignment Statements gives a set of assignments related to the experiment. Students are recommended to accomplish these assignments by building their own circuit, simulate and examine the desired behavior. These assignments have been designed in such a way that by performing them a student will be more familiar with the experiment topic and several design aspects, in totality, these will help students to have a good design experience and uncover their innovation along with enhanced understanding.

• *Procedure* section gives a detail step by step guidance to perform the experiment on the given working module in the simulator. Also it lists the required components to design circuit for that experiment. Web interface of the procedure of the *Direct mapped cache* experiment in COLDVL is shown in Figure 2.11.

• *Experiment* section contains the logic simulator which simulates the experiments. The simulator is available in two forms, it can either be launched or downloaded. If the simulator is launched, every time the user gets the updated version of the simulator automatically but if the simulator is downloaded and used then user does not get the updated version automatically. The user has to re-download the simulator to get the latest version. The simulator is capable of simulating circuits other than the designed experiments as it provides a generic simulation platform for both combinational circuits and synchronous sequential circuits. The



```
[State-Input-Output]
S7  [1, 0]  [0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1]
```

```
CURRENT STATE: S7 OUTPUTS: 00000000001
```

Figure 2.6: Controller state transitions of the user designed single instruction CPU in the basic stage of the learning sequence.

detail features of the simulator are described in the subsequent sections along with relevant figures.

- *Quizzes* section includes a wide set of questions for self assessment. Students can asses their knowledge level at any stage of their learning process. Web interface of the quizzes of the *Direct mapped cache* experiment in COLDVL is shown in Figure 2.12.

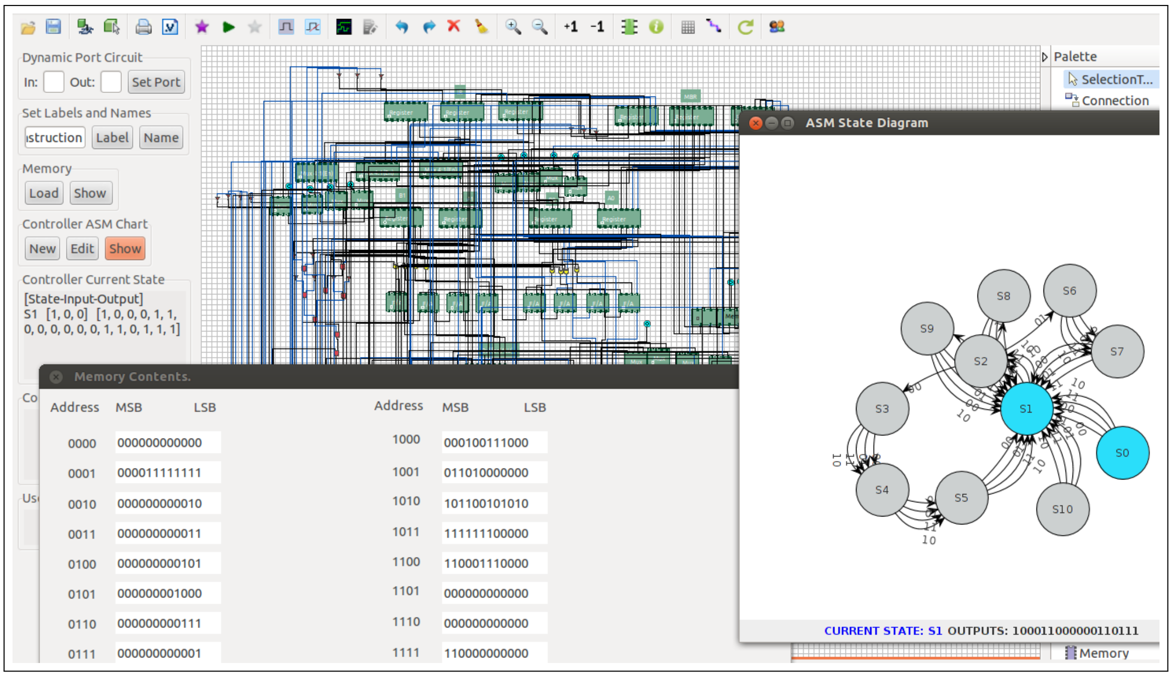
- *Further Reading* section gives a set of references such as web lectures, video lectures, books, URLs to increase their knowledge.

2.5 Gathering of user feedback

COLDVL has been deployed, used and evaluated by several students and faculty members of colleges all over India. It has also been used for academic curriculum in IIT Kharagpur to conduct laboratory courses of undergraduate and postgraduate students. This section briefly presents the designs of feedback questions, deployment of COLDVL for collecting user feedback and analysis of the gathered feedback.

2.5.1 Designing feedback questions

The feedback questions are designed in such a way that can focus many aspects of of the COLDVL and the assimilation of its associated pedagogic considerations. One of the questions, "*Do you think*



[State-Input-Output]
S1 [1, 0, 0] [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1]

Memory Contents.

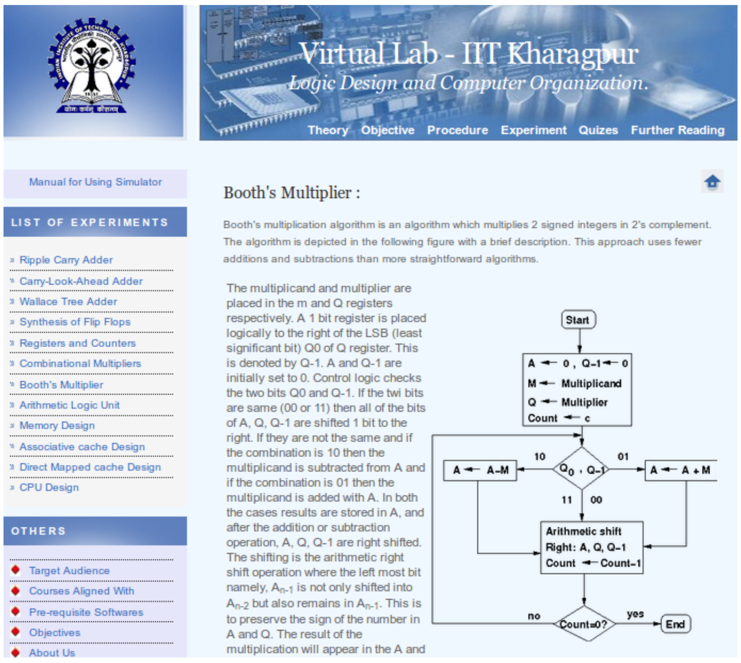
Address	MSB LSB	Address	MSB LSB
0000	000000000000	1000	000100111000
0001	000011111111	1001	011010000000
0010	000000000010	1010	101100101010
0011	000000000011	1011	111111100000
0100	000000000101	1100	110001110000
0101	000000001000	1101	000000000000
0110	000000000111	1110	000000000000
0111	000000000001	1111	110000000000

CURRENT STATE: S1 OUTPUTS: 100011000000110111

Figure 2.7: User designed regular CPU with four instructions in the advanced stage of the learning sequence.

that this kind of virtual lab with experiments and theory will really enhance student learning?" was designed to focus on the learning enhancement pedagogic consideration and "*Do you find this simulator and the associated experiments will motivate students for self-learning?*" was designed to analyze the implementation of the pedagogic principles for motivating learners. The usefulness of addressing cognitive issue such as attentional issues, addressing misconceptions, learning mechanisms were analyzed through the questions, "*Do you have a clear understanding of the experiment and related topics?*", "*Was the procedure and manual found to be helpful?*" and "*Do you think doing experiments through virtual lab gives scope for more innovative and creative work?*"

Other questions such as "*How much do you like the way of analyzing results and the value propagation in your circuit through different wire colors?*" was designed to know the impact of different wire colors for multiple logic values in the simulator for effective learning. Where as "*Rate the quality of graphics of the simulator*" question was set to analyze whether the simulator appears visually attracted to the users. However, the assimilation of the pedagogic considerations, *order of learning concepts*, does not likely to be validated in a one-day workshop. Therefore, it is planned to collect feedback from the students who have done a semester course lab with COLDVL. The following subsections describe the deployment of COLDVL for gathering feedback and the analysis of the feedback along with the summary of the collected feedback.



Booth's Multiplier :

Booth's multiplication algorithm is an algorithm which multiplies 2 signed integers in 2's complement. The algorithm is depicted in the following figure with a brief description. This approach uses fewer additions and subtractions than more straightforward algorithms.

The multiplicand and multiplier are placed in the m and Q registers respectively. A 1 bit register is placed logically to the right of the LSB (least significant bit) Q0 of Q register. This is denoted by Q-1. A and Q-1 are initially set to 0. Control logic checks the two bits Q0 and Q-1. If the twi bits are same (00 or 11) then all of the bits of A, Q, Q-1 are shifted 1 bit to the right. If they are not the same and if the combination is 10 then the multiplicand is subtracted from A and if the combination is 01 then the multiplicand is added with A. In both the cases results are stored in A, and after the addition or subtraction operation, A, Q, Q-1 are right shifted. The shifting is the arithmetic right shift operation where the left most bit namely, A_{n-1} is not only shifted into A_{n-2} but also remains in A_{n-1}. This is to preserve the sign of the number in A and Q. The result of the multiplication will appear in the A and

Figure 2.8: Web interface of COLDVL.

2.5.2 Deployment of COLDVL

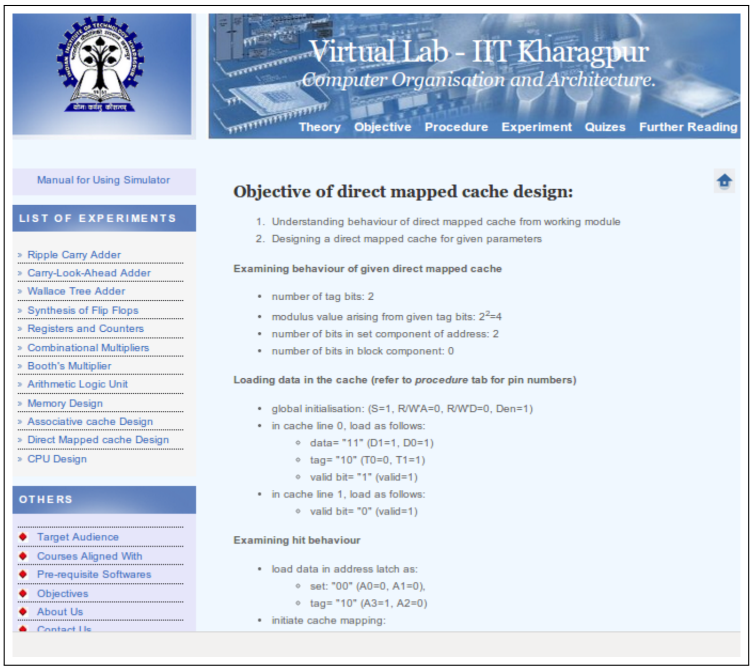
The following paragraphs describe where and how COLDVL has been deployed and used to gather feedback from the prospective end users.

Semester laboratory course COLDVL has been used for conducting laboratory sessions in the first year M.Tech course 'Computing systems lab' for laboratory experimentation in Autumn session, 2012, 2013, and 2014. Students were given assignments on logic design and computer organization, after building their circuit and saving with their identification they uploaded the file to the *Web-Based Course Management system* [36] of IIT Kharagpur over Internet, for manual evaluation.

In the workshops, COLDVL has been demonstrated very briefly to the participants, then they used COLDVL, performed various experiments and returned their feedback.

Workshop for faculty members A workshop was conducted in association with a short term teacher training course, 25-30 June, 2012, at IIT Kharagpur for the AICTE faculty members where faculty members from different engineering colleges all over India had participated.

Workshops for students Workshops were also organized in engineering colleges of three different universities of West Bengal, India, which includes Jadavpur University, Bengal Engineering and Science University and West Bengal University of Technology. Students with computer science, information technology, electronics background participated in the workshops organized within February to March, 2013, as they all have digital logic and computer organization as their core courses.



Objective of direct mapped cache design:

1. Understanding behaviour of direct mapped cache from working module
2. Designing a direct mapped cache for given parameters

Examining behaviour of given direct mapped cache

- number of tag bits: 2
- modulus value arising from given tag bits: 2^2=4
- number of bits in set component of address: 2
- number of bits in block component: 0

Loading data in the cache (refer to *procedure* tab for pin numbers)

- global initialisation: (S=1, R/WA=0, R/WD=0, Den=1)
- in cache line 0, load as follows:
  - data= "11" (D1=1, D0=1)
  - tag= "10" (T0=0, T1=1)
  - valid bit= "1" (valid=1)
- in cache line 1, load as follows:
  - valid bit= "0" (valid=1)

Examining hit behaviour

- load data in address latch as:
  - set: "00" (A0=0, A1=0),
  - tag= "10" (A3=1, A2=0)
- initiate cache mapping:

Figure 2.9: Web interface of the objectives of the *Direct mapped cache* experiment in COLDVL.

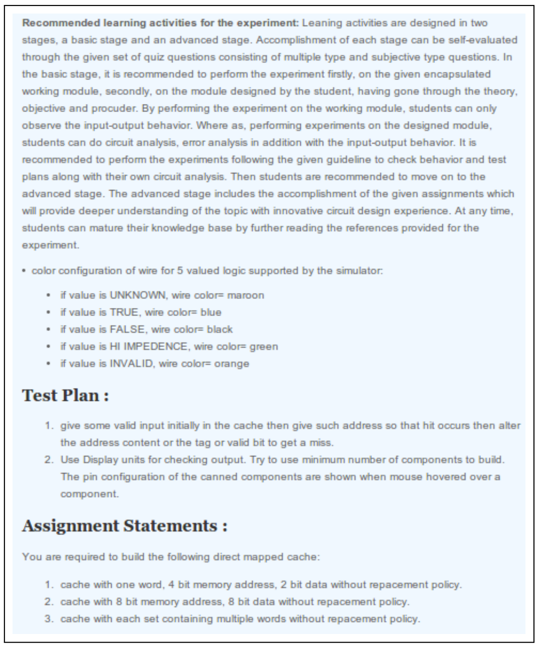
Independent participation The COLDVL was used and evaluated independently (without the benefit of workshop conducted by us) at NIT Jalandhar, India, early in 2013. This participation is of special significance as students are expected to be able to use the COLDVL to conduct experiments, as it is available in the Internet without any special guidance from us.

2.5.3 Summary and analysis of user responses

COLDVL gained very positive and encouraging response from both the students and teachers. Feedback of students who have done the experiments seriously have been considered (18 excluded). Total 162 feedback sets are analyzed including 129 student and 33 teacher responses.

Upon asking if this virtual laboratory package including the set of experiments and theories will really enhance student learning, in a scale of *yes, to some extent, no*, 93.9% participants said *yes* and 6.1% said *to some extent* while none of them said *no* which strongly indicates the success of achieving learning enhancement objective. And 86.9% users find the simulator and the associated experiments will motivate students for self-learning. Figure 2.14 shows some of the significant feedback along with the following. On asking whether participants have a clear understanding of the experiment and related topics 82.6% answers were *yes* and 82.1% participants found the procedure and manual associated with each experiment helpful. 83.3% participants suggested that doing experiments through virtual lab gives scope for more innovative and creative work. These indicate the success in addressing the cognitive issues in COLDVL.

Feedback sets are collected from the students who have done a semester course lab with COLDVL



Recommended learning activities for the experiment: Leaning activities are designed in two stages, a basic stage and an advanced stage. Accomplishment of each stage can be self-evaluated through the given set of quiz questions consisting of multiple type and subjective type questions. In the basic stage, it is recommended to perform the experiment firstly, on the given encapsulated working module, secondly, on the module designed by the student, having gone through the theory, objective and procuder. By performing the experiment on the working module, students can only observe the input-output behavior. Where as, performing experiments on the designed module, students can do circuit analysis, error analysis in addition with the input-output behavior. It is recommended to perform the experiments following the given guideline to check behavior and test plans along with their own circuit analysis. Then students are recommended to move on to the advanced stage. The advanced stage includes the accomplishment of the given assignments which will provide deeper understanding of the topic with innovative circuit design experience. At any time, students can mature their knowledge base by further reading the references provided for the experiment.

- color configuration of wire for 5 valued logic supported by the simulator:
  - if value is UNKNOWN, wire color= maroon
  - if value is TRUE, wire color= blue
  - if value is FALSE, wire color= black
  - if value is HI IMPEDENCE, wire color= green
  - if value is INVALID, wire color= orange

Test Plan :

1. give some valid input initially in the cache then give such address so that hit occurs then alter the address content or the tag or valid bit to get a miss.
2. Use Display units for checking output. Try to use minimum number of components to build. The pin configuration of the canned components are shown when mouse hovered over a component.

Assignment Statements :

You are required to build the following direct mapped cache:

1. cache with one word, 4 bit memory address, 2 bit data without repacement policy.
2. cache with 8 bit memory address, 8 bit data without repacement policy.
3. cache with each set containing multiple words without repacement policy.

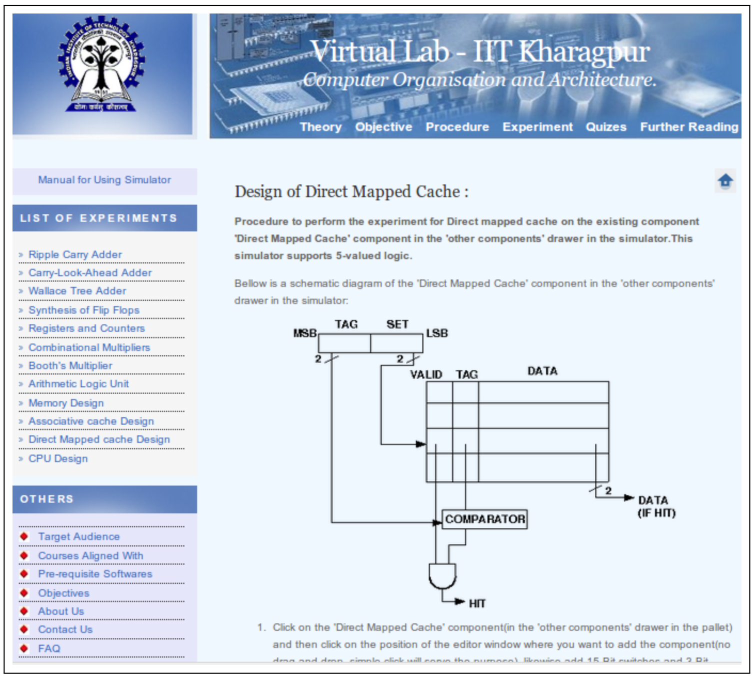
Figure 2.10: Web interface of the test plan and assignments of the *Direct mapped cache* experiment in COLDVL.

package to focus the usefulness of the *order of learning concepts* incorporated. They found that the order in which experiments evolve from easier to harder concepts help understanding and developing a structured overview of the subject. They also found the test plan and challenging assignments associated with each experiment to get introduced with the practical aspects of theoretical concepts along with fostering their skills to apply their learning.

Most of the faculty members found that COLDVL will be extremely helpful for students not only in colleges where real laboratory facilities are inadequate but they found it helpful for students as well as their teachings and worthy of recommending it to their students. Many of the faculty member participants observed that several important features are integrated into the virtual laboratory package. Through the entire workshop session, COLDVL gained high appreciation from all the participating faculty members.

Students who were not given demonstration, performed well and sent their feedback along with their saved circuit designs within a short period of time since they started using the COLDVL package. They have done not only simple experiments successfully, but accomplished complex experiments too. Their activities and feedback show that the web interface of COLDVL stands of its own that they could easily learn COLDVL and use it to perform experiments.

In the two hours of workshops, after a 30 minutes of demonstration students performed experiments successfully. Students found this virtual laboratory package including the COLDVL simulator very interesting and motivating. They also enjoyed doing experiments in COLDVL. While a single student performed simple experiments, a group of 2-3 students performed more complex experiments. The participating students and those remote students could not perform experiments



Design of Direct Mapped Cache :

Procedure to perform the experiment for Direct mapped cache on the existing component 'Direct Mapped Cache' component in the 'other components' drawer in the simulator.This simulator supports 5-valued logic.

Bellow is a schematic diagram of the 'Direct Mapped Cache' component in the 'other components' drawer in the simulator:

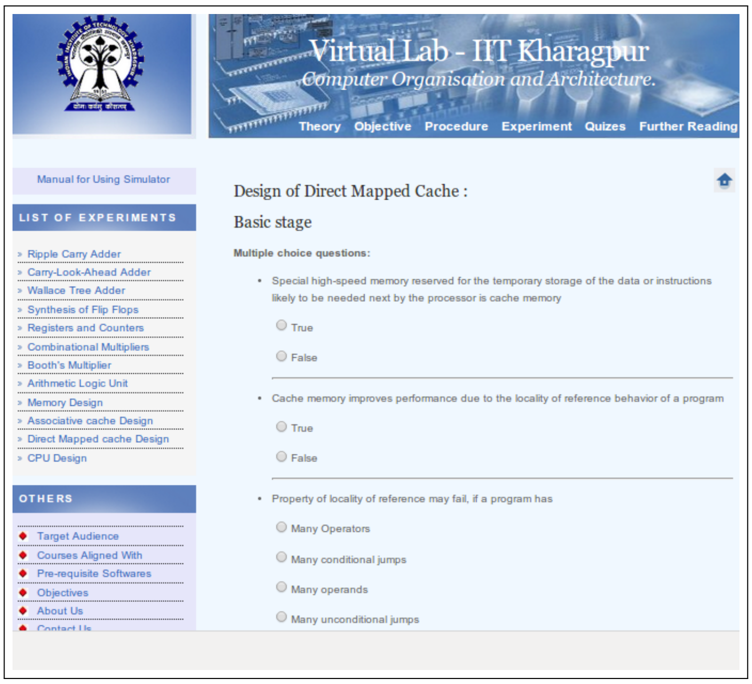
1. Click on the 'Direct Mapped Cache' component(in the 'other components' drawer in the pallet) and then click on the position of the editor window where you want to add the component(no

Figure 2.11: Web interface of the procedure of the *Direct mapped cache* experiment in COLDVL.

if they had not have clear understanding which implies that COLDVL is easy to learn. Many of the students also found that COLDVL is very time and effort saving by reducing the ‘ clumsiness’ of bread board and providing an easy way to implement their circuits. Students also liked the availability of COLDVL 24 hours over Internet without any setup overhead so that they can experiment ‘creatively’ and brush up their concepts with experiments any time.

2.6 Conclusion

The design issues of the virtual laboratory of the present work is discussed in this chapter. The pedagogic considerations of the virtual laboratory is given. This chapter also describes the design of the web interface, experiments design, design of the sequence of learning activities in the virtual laboratory using, features of the COLDVL tool (using well known pedagogic principles) in order to assimilate the pedagogic considerations. The deployment of the COLDVL tool, gathering of satisfactory user feedback and its analysis have also been discussed in this chapter.


Virtual Lab - IIT Kharagpur

Computer Organisation and Architecture.

Theory Objective Procedure Experiment Quizes Further Reading

Manual for Using Simulator

LIST OF EXPERIMENTS

- Ripple Carry Adder
- Carry-Look-Ahead Adder
- Wallace Tree Adder
- Synthesis of Flip Flops
- Registers and Counters
- Combinational Multipliers
- Booth's Multiplier
- Arithmetic Logic Unit
- Memory Design
- Associative cache Design
- Direct Mapped cache Design
- CPU Design

OTHERS

- Target Audience
- Courses Aligned With
- Pre-requisite Softwares
- Objectives
- About Us
- Contact Us

Design of Direct Mapped Cache :

Basic stage

Multiple choice questions:

- Special high-speed memory reserved for the temporary storage of the data or instructions likely to be needed next by the processor is cache memory
  - True
  - False
- Cache memory improves performance due to the locality of reference behavior of a program
  - True
  - False
- Property of locality of reference may fail, if a program has
  - Many Operators
  - Many conditional jumps
  - Many operands
  - Many unconditional jumps

Figure 2.12: Web interface of the quizzes of the *Direct mapped cache* experiment in COLDVL.

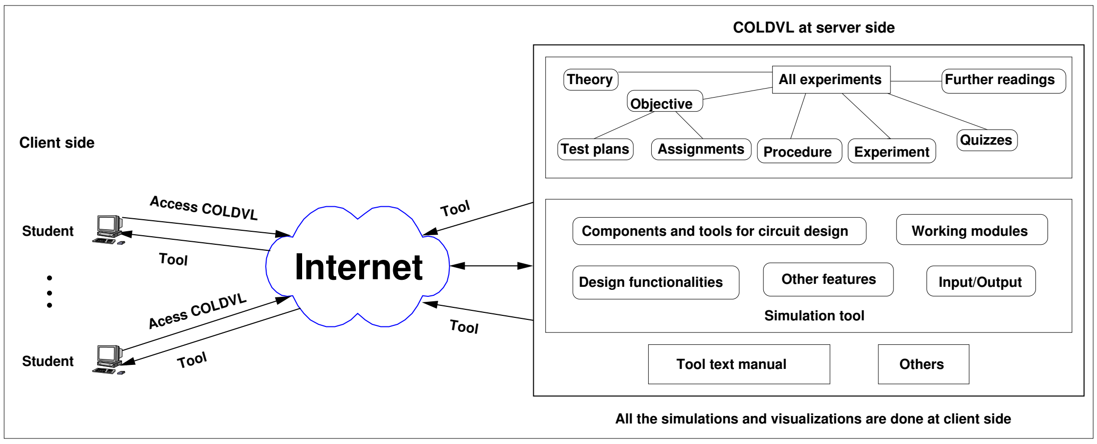


Figure 2.13: Basic components of COLDVL and interactions with end users.

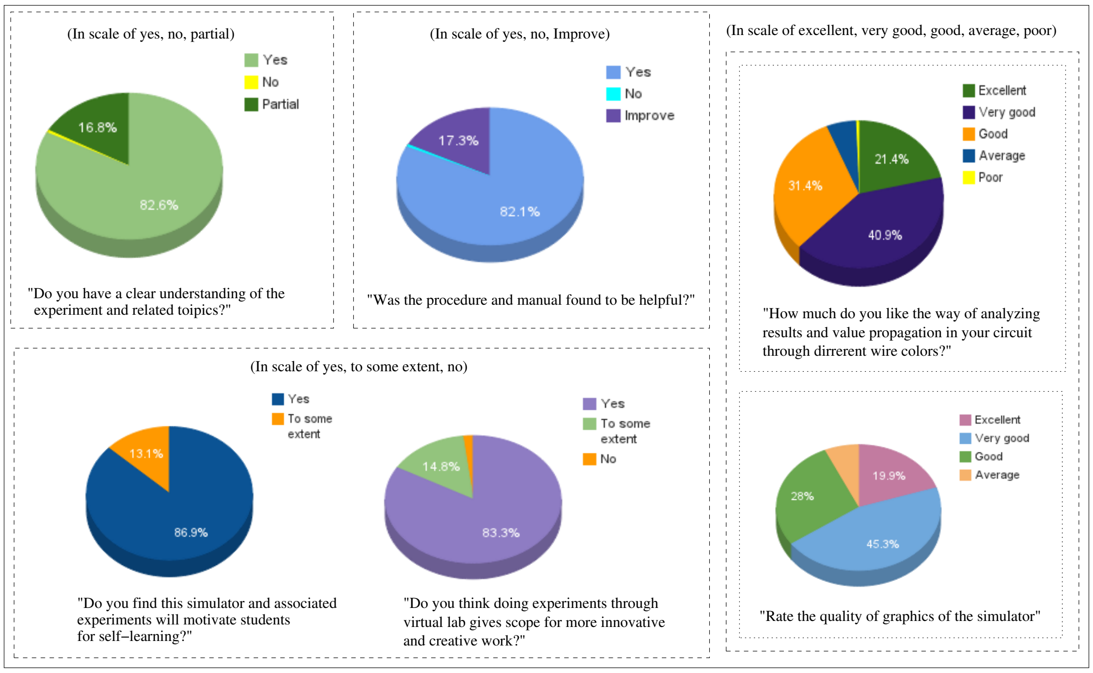


Figure 2.14: User feedback of some questions asked.

Chapter 3

Front end of the COLDVL tool

3.1 Introduction

This chapter focuses on the front end features of the COLDVL tool. The front end features include a brief description of components, circuit design and editing functionalities, other significant features developed, the overall user interface where circuits can be built along with the interface developed for some specific features. The tool architecture is also described very briefly. Case studies are provided to indicate the usability of the front end features along with the usability of COLDVL tool including the features of COLDVL tool to aid learning and the use of COLDVL tool to conduct laboratory courses at IIT Kharagpur is also included in this chapter.

3.2 Features of COLDVL tool

Components and circuit building facilities

The COLDVL tool contains a repertoire of building blocks and reference designs such as basic gates, tri-state buffers, input-output components, combinational components, sequential components. Input-output components contain toggle switch (which can provide both true and false input), free running clock, bit display to show value of a single bit, digital display which will show the digital value of multiple binary bits. Combinational components include different types of adders such as, half adder, full adder, ripple-carry adder, carry-look-ahead adder, Wallace tree adder, subtracter, decoders, multiplexers, comparator, combinational multiplier employing carry-save-addition scheme, arithmetic logic units, etc. This tool provides several sequential components ranging from basic to complex components such as flip flops (both behavioral and structural), different types of registers, counters, RAMs with editable cells, cache memories (without replacement policy) such as associative and direct mapped cache, a single instruction CPU with in-built controller. The tool also contains reference designs related to the guided experiments provided in the virtual laboratory to help novice students perform the experiments on a black box like component where they can only check the input-output behavior. The internal details of the components given in the tool are hidden from the user. This helps students to perform the recommended sequenced learning activities (first step of the basic stage) of the virtual laboratory mentioned in the previous Chapter 2, Section 2.3.3. To accomplish the learning activities of the other stages, students need to to build the circuit using the components provided in the tool or they can build and reuse their own component. Building circuits require the instantiation of the necessary components in the graphical interface provided for circuit building, connecting those components, providing inputs, initiating simulation and finally observing the results using display units. The tool has necessary facilities to connect components including bus connector in order to support the bus-based design with wired AND operation, cloning facility to avoid the overhead of repetitive designs in large circuits. functions.

Logic values The COLDVL tool uses five levels of logic values namely True (T), False (F), High Impedance (Z), Unknown (X) and Invalid (I). The High Impedance is included to support tri-state logic, the Unknown is used as the default initial value of any logic signal which essentially indicates the value of the logic signal either 0 or 1 and the Invalid is used in order to support the bus-based design with wired AND operation. The truth tables of the five valued logic used in this tool are shown in the Table 3.1.

Logic values and wire colors

The 5 different logic values which may present in the nets of a circuit are indicated by five different colors. Thus the connecting wires of a circuit may take five different colors. The colors of wires can be used for circuit debugging, examining the value propagation through the circuit and it can also be very useful for result analysis in the circuit. The logic values along with their corresponding wire colors are listed in the Table 3.2.

Once a circuit is built and the user initiates simulation, the simulation algorithm then finally delivers the results by dynamically changing the visual aspects of the circuit. Initially the wire values are unknown prior to simulation and having colors as maroon. However, after simulation the wire values may take any logic values among the supported five logic values indicated by appropriate colors. The visual aspects of the wires in a circuit before and after simulation is schematically depicted in the Figure 3.1

Design of hierarchical modules

The graphical interface restricts the size of the designed circuit due to its space limitation. Although for smaller experiments the space limitation does not become an issue. However, for complex circuits with bigger data paths such as circuits implementing multiplication, division, CPU, the space limitation imposes a restriction. This limitation can be overcome through hierarchical designs where a set of connected components are made as one module abstracting the visual aspects of the internal connections and components, more like a black box providing only the input and output ports. A module can also be used to create another module in a bottom up manner. COLDVL tool provides facility to create such encapsulated modules designed by users which will be saved as the user component library in local machine, and the user can reuse their previously built encapsulated

	0	1	X	Z	I
0	0	0	0	0	0
1	0	1	X	X	X
X	0	X	X	X	X
Z	0	X	X	X	X
I	0	X	X	X	X

AND gate

	0	1	X	Z	I
0	0	1	X	X	X
1	1	1	1	1	1
X	1	X	X	X	X
Z	1	X	X	X	X
I	1	X	X	X	X

OR gate

	0	1	X	Z	I
0	1	1	1	1	1
1	1	0	X	X	X
X	1	X	X	X	X
Z	1	X	X	X	X
I	1	X	X	X	X

NAND gate

	0	1	X	Z	I
0	1	0	X	X	X
1	0	0	0	0	0
X	X	0	X	X	X
Z	X	0	X	X	X
I	X	0	X	X	X

NOR gate

	0	1	X	Z	I
0	0	1	X	X	X
1	1	0	1	1	1
X	1	X	X	X	X
Z	1	X	X	X	X
I	1	X	X	X	X

XOR gate

	0	1	X	Z	I
0	1	0	X	X	X
1	0	1	X	X	X
X	X	X	X	X	X
Z	X	X	X	X	X
I	X	X	X	X	X

XNOR gate

	0	1	X	Z	I
0	Z	0	X	X	X
1	Z	1	X	X	X
X	Z	X	X	X	X
Z	Z	X	X	X	X
I	Z	X	X	X	X

Tri-state

In	Out
0	1
1	0
X	X
Z	X
I	X

NOT gate

Table 3.1: Truth tables of basic gates used in five valued COLDVL tool

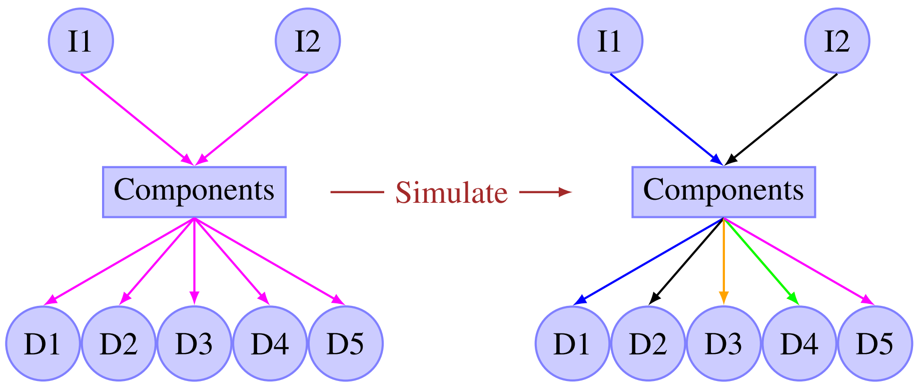


Figure 3.1: Wire values within a circuit before and after simulation (I1, I2 denotes inputs and D1 to D5 denotes display units)

Logic value	Wire color
True (T)	Blue
False(F)	Black
High impedance(Z)	Green
Unknown(X)	Maroon
Invalid(I)	Orange

Table 3.2: Logic values and their corresponding wire colors

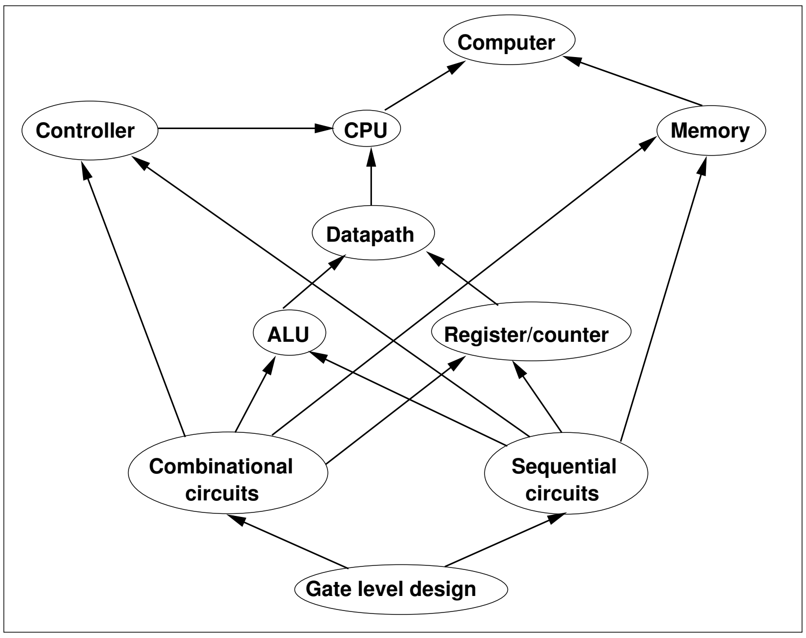


Figure 3.2: Design hierarchy supported by the COLDVL tool.

modules to build larger circuit modules. With this facility the tool provides support for a circuit design hierarchy, shown in Figure 4.5 which allows users to design circuits from basic gate level to large modules and larger modules using smaller modules. The components at the arrow head implies that it is synthesized from the component at its arrow base. For example, combinational circuits such as adder, subtracter, multiplexer, decoder etc. are built from gate level components. Arithmetic logic units can be synthesized from combinational and/or sequential circuits. As the tool contains components belonging to each level of the design hierarchy, students can start designing at any level of the design hierarchy by using the components provided by the tool or their own designed encapsulated modules. This feature is more clarified in a case study in the later section of this chapter.

Automatic encapsulated control unit generation

The COLDVL tool has a facility to automatically generate an encapsulated control unit from a user specified control chart of Moore type. Appropriate interfaces are provided for specifying the control chart. After entering the state chart, the tool builds the corresponding encapsulated controller (whose internal details are hidden from the user) which will act according to the specified state chart in the back end and when the user instantiates the encapsulated controller component from the tool front end, the internally built controller is delivered. After the instantiation of the controller component, it can be connected to the components of the data path in order to build a circuit having a control flow and data transformations where every data transformation is triggered by the control signals. There are many computer arithmetic algorithms such as multiplication, division, complex designs such as CPU with different types of instructions which are very important topics in computer organization course and laboratory experimentation covers those topics so that students can understand how the theories can be practically implemented and develop a clear concept regarding the theoretical aspects along with the design issues of a topic. In practice, most of the implementations of such complex topics require the data path to work in association with a controller. However, realization of a hardwired control unit corresponding to a simple control state chart may not be a cumbersome process, for large and complex state charts it introduces overheads in terms of time and effort. Therefore, we have developed this feature of control signal generation from user specified state chart to reduce the design complexity of hardwired control unit and thus giving students more scope to concentrate upon learning issues regarding the circuit design for complex computer arithmetic algorithms and CPU designs.

Visualization of controller state transition

After instantiating the controller component from a given control state chart, it is often needed to test the controller behavior i.e. to test whether the state chart is designed correctly. Testing can be done connecting input-output components to the controller unit. However, to make this testing easy, the tool has options to visualize the state transitions of the controller both in a text field and as a state model during simulation. In order to do that, the state model is extracted from the user given state chart and during simulation the transitions are shown by dynamically changing the view in the model.

Detection of possible race around condition before simulation

The COLDVL tool can detect the possible occurrence of the race around conditions prior to the simulation that are generated due to the use of non-master-slave flip flops in a circuit. When the possible race condition is detected, the user is warned and in that case the user may want to modify the circuit and proceed or can proceed by ignoring the warning. This feature is developed to help students learning safe sequential circuit design.

Regular and advanced simulation features

An efficient simulation technique is developed for the sequential circuits conforming to the Huffman model. Another case based analysis technique is developed where regular simulation can not produce output. The detail description of the simulation related techniques are the content of the next chapter.

Working memory to support CPU experimentation

As CPU design is an important topic in the computer organization course, students are required to design basic CPU with single or multiple instructions in laboratory experimentation. Once a CPU is built, its behavior can be checked with the execution of a program. Students can build memory having multiple address line from the basic single bit RAM cell included in the tool. However, building memories for such purpose is time consuming and topic of an individual experiment related to memory design. Therefore, in order to focus on particularly CPU design without associating the overhead of memory design, an in-built working memory is provided in the tool which can be loaded with binary program along with the data and connected with the CPU. Then the CPU will execute the program loaded in the working memory reading and writing from and to the working memory.

Saving circuits with unique identification

This tool has a feature of saving user circuits with identification to prevent plagiarism to a great extent. When students will save their circuits in order to submit their assignments for evaluation, they are bound to provide their name, roll number, then only the circuit can be saved to a file. Once a file is saved with an identification, the identification can not be further modified even after re-opening and re-saving the file. Thus one can not copy files from one another.

Structural verilog code generation

A structural gate level verilog module for a circuit can also be generated in this tool. Behavioral verilog module is realized for behavioral component of a circuit. As COLDVL tool supports hierarchical circuit design, verilog realization of a circuit containing several modules is also modular but structural.

Other features

Apart from the above mentioned features, the COLDVL tool has several other features. Some of the other features are as follows.

Connection related: The connection of components during circuit building has some restriction in order to help students learning good design practice along with reducing design errors such as two input terminals can not be connected directly, connection is allowed from output terminal to input terminal, two non-tristated outputs can not be connected to the same input terminal. Connection between terminals can also be dragged to another terminal. Two types of connections are supported which are *shortest path type* and *Manhattan type* and the connection type of a circuit can be changed anytime. This facility is provided for the ease of circuit debugging.

Design editing functionalities: During building circuits *undo*, *redo* and *deletion* facilities are provided in the COLDVL tool. A component or a connection of a circuit can be deleted or a part or the full circuit can be deleted. Before using the deletion operation, the parts to be

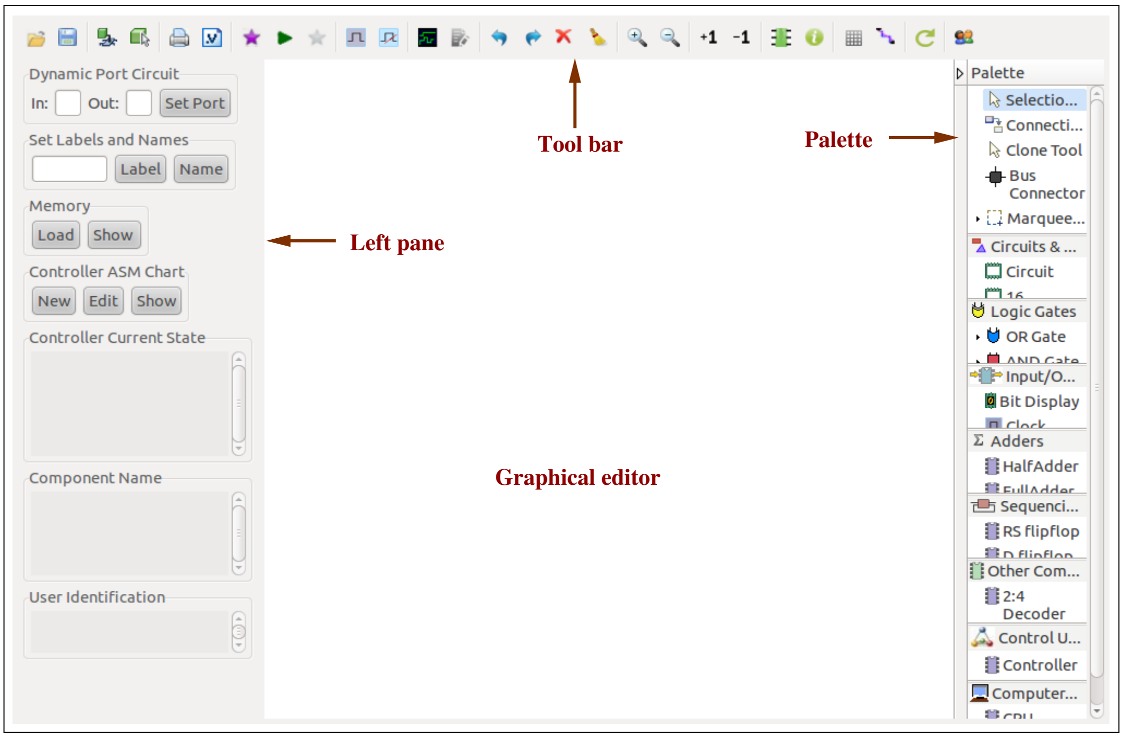


Figure 3.3: Graphical interface of the COLDVL tool.

deleted must be selected. The tool provides several selection facility for deletion of of only components or only connections or both.

Graphical editor related: The editor view is gridded (grid view can also be turned off) and does an automatic calculation of co-ordinates for the placements of the components. Circuits or part of a circuit can be re-positioned by dragging to the desired position in the graphical editor. As the tool does not provide multiple tabs, designs are restricted with the single graphical editor. Once a circuit is done it can be either saved with identification of made as encapsulated components to be used in other circuits or it can be discarded by clearing the whole graphical editor so that new designs can be built. The view of the graphical editor can also be zoomed in or zoomed out.

3.3 User interface of COLDVL tool

Basic user interface of the tool is presented in Figure 3.3 which consists of four parts as follows.

1. A graphical editor for building circuits.

2. A palette containing a list of design components such as logic gates, input/output units, adders, flip flops, registers, counters, decoders, multiplexers, arithmetic logic units, tri-state buffers, bus connector, RAM cells, cache memory and many other complex components including control unit, single instruction CPU, a working memory to be loaded with binary

program to check CPU behavior and tools for creating connections between design components, cloning operation. Several drawers of the pallete are shown in the Table 3.3.

3. A tool bar on the top of the graphical editor provides facilities for file operations, edit functionalities, simulation related functions and other operations. The file operations include saving and reopening of circuits, saving circuits as encapsulated component modules and reusing them. Delete, zoom in-out, undo-redo, clear editor are some of the operations which are provided for editing purpose. Simulation related functions consist of start simulation, turning case analysis feature on or off, optional simulation of structural memory elements while reconstructing the circuit, log generation after case analysis, start-stop of clock pulse generator. Other operations include exporting user circuit to pdf format, plotting a graph to see the input-output behavior, generating structural verilog netlist of the circuit, showing pin configuration and individual component name, changing connection type from Manhattan to shortest path, resetting controller, showing user identification for a saved circuit, grid enable-disable.

4. A left pane consists of functionalities such as setting input-output port for dynamic circuit, setting label text and component name other than system generated name, loading and showing working memory content, loading, editing and showing state model along with current state of an ASM chart and panels to show system generated component name and user identification of a saved circuit.

Interface to visualize input-output behavior

The interface for visualizing the wave form is shown in the Figure 3.4. A JK flip flop is simulated with an advanced case based analysis feature (elaborated in the next chapter) and the input-output behavior can be observed. There is a facility to view the system generated names of the basic gate components and the input-output components of a circuit, which is also shown in the figure in the *component name* text field in the left pane of the tool. The names of the input-output components must be known in order to see the wave forms corresponding to the signals present in any input or output component with respect to time.

Interface for specifying control state chart

The tool accepts state charts those are of Moore type for the automatic generation of the encapsulated control unit from the given state chart. Initially, when user initiates the specification of a controller, the tool takes inputs about the number of states, inputs, outputs, names of the inputs and outputs. The interfaces for these are not shown here. After getting those numbers of state, inputs and outputs, the tool then generate the interface for specifying the state chart of Moore type by generating all possible combinations of the inputs. Figure 3.5 shows an interface for state chart specification given the number of states as five with three inputs and four outputs. Once a state chart is specified, it can be edited and the editing functionalities are also provided in the interface.

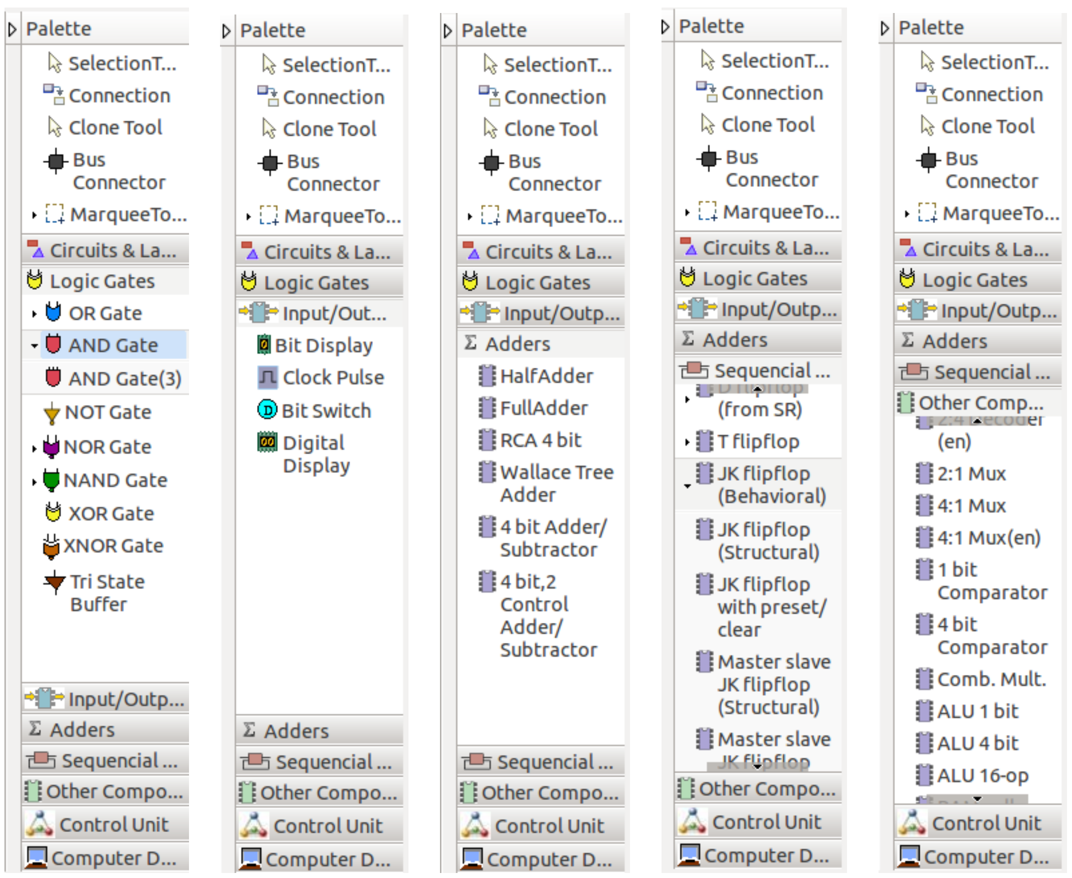


Table 3.3: Different drawers of the pallete containing different components

Interface related to load working memory

Figure 3.6 shows the interface to load the working memory with binary program and data. The loading can be done either manually or it can read from a text file containing the binary program and data. The content of the working memory can also be reset. Figure 3.6 shows the content of the working memory after loading it with binary program and data.

The structural verilog code generation is shown in Figure 3.6 for a JK flip flop. The generated verilog code is automatically shown through the text editor present in the local machine. As mentioned earlier, a circuit can be exported to the pdf format which is shown in Figure 3.9 for the JK flip flop. The interfaces of some other features are given during explaining some case studies in the later sections.

3.4 COLDVL tool architecture

Figure 3.10 shows the architecture of the COLDVL tool. The tool architecture consists of two parts, one is the front end and another is the back end. The front end contains a repertoire of components including basic gates, encapsulated components, encapsulated reference designs, design

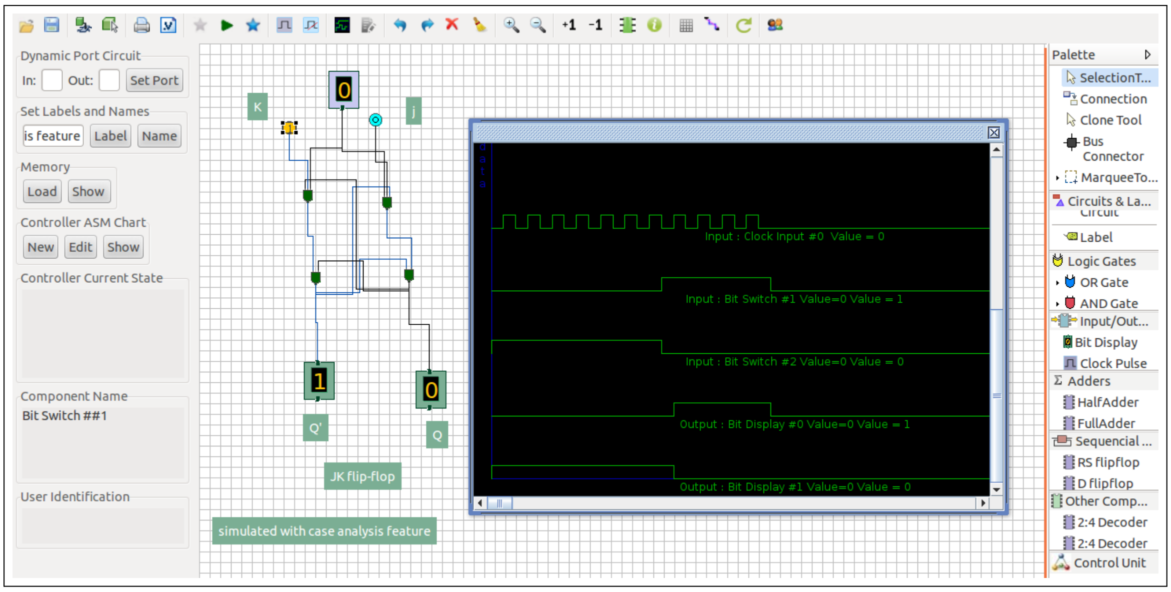


Figure 3.4: Wave forms of input-output components of a JK flip flop.

functionalities and a graphical interface to build a circuit. Whenever simulation is initiated, the circuit is delivered to the back end. The back end contains the all the simulation related algorithms implemented on top of some system libraries. The simulation algorithms includes main simulation along with some advanced techniques which are discussed in the next chapter. After simulation, the output is delivered to the circuit in the front end by changing its visual aspects. Apart from this, on demand of the user, some other interface is also updated after the simulation such as if user has wants to see the wave form for the input-output, current controller state or working memory content, etc.

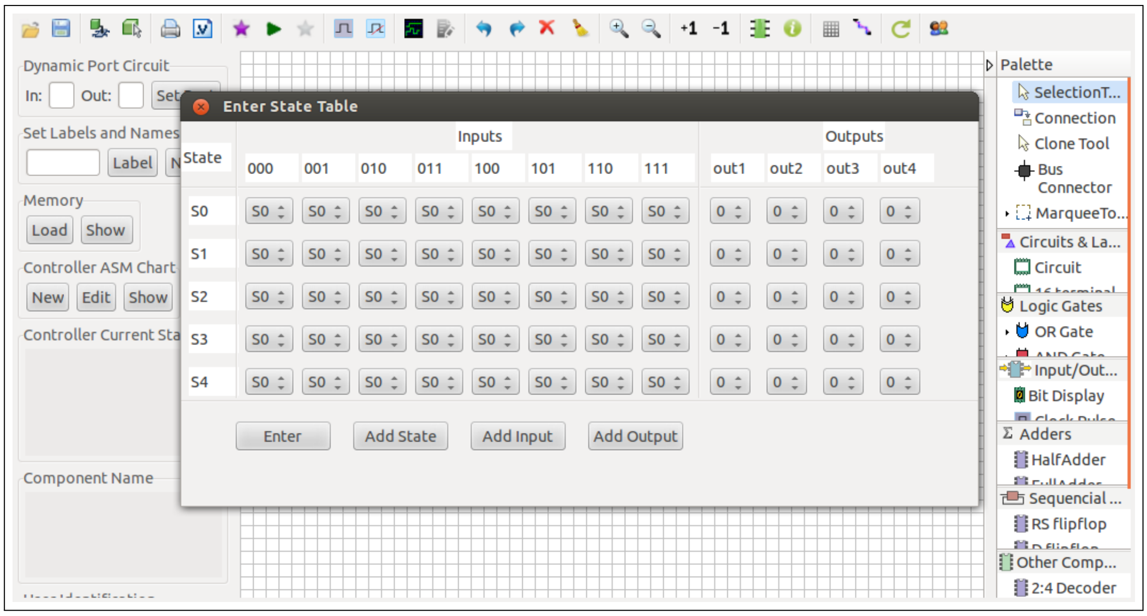
3.5 Case studies and usability of COLDVL tool

This section includes three case studies which explain the usability of some important features of the COLDVL tool and how the features can help building basic circuits and as well as complex circuits. The practical use of COLDVL tool is also discussed.

3.5.1 Simple combinational and sequential circuits

A simple combinational circuit

Figure 3.11 shows a simple combinational circuit built in the graphical interface of the tool using the basic gates provided in the tool. Bit switches are connected to provide both the inputs logic value 0 and 1 which is indicated through distinct colors. If the bit switch provides logic value 0 then it takes cyan color and if it provides logic value 1 then it becomes yellow. With a double click the



	Inputs								Outputs			
State	000	001	010	011	100	101	110	111	out1	out2	out3	out4
S0	S0	S0	S0	S0	S0	S0	S0	S0	0	0	0	0
S1	S0	S0	S0	S0	S0	S0	S0	S0	0	0	0	0
S2	S0	S0	S0	S0	S0	S0	S0	S0	0	0	0	0
S3	S0	S0	S0	S0	S0	S0	S0	S0	0	0	0	0
S4	S0	S0	S0	S0	S0	S0	S0	S0	0	0	0	0

Figure 3.5: Interface for specifying control state chart.

bit switch toggles its current value and as well as changes its visual aspect. Bit displays are also connected to the output terminal of some gates in the circuit from which the output of the circuit can be viewed as binary value. Along with the conventional shapes, different basic gates also have different colors such as AND gates are red, OR gates are blue, etc. so that in large circuits with many gates, the value propagation can be easily tracked for circuit debugging and circuit analysis. Text labels are used in this circuit to denote which input-output component denotes which boolean variable belonging to the boolean function realized in this circuit.

A simple sequential circuit

Figure 3.12 shows another circuit built and simulated using COLDVL tool which is a sequential circuit. The user identification associated with the circuit is also shown in the left pane. The circuit is a 4-bit parallel load registers and gate level structural D flip flop components are used as memory elements. The D flip flops used in the circuit are instantiated from the pallete of the tool which are the encapsulated components i.e. is its internal structural gate level design are hidden from the user, provided by the tool. All the upper and lower terminals of an encapsulated component is the input and output terminals respectively. The components are designed in such manner in order to maintain a similarity with the conventional hardwire chip. Input-output components are also attached to the circuit and labels shows which terminal is least significant bit and which is most significant bit.

3.5.2 A complex sequential circuit with controller and data path

Shift and add multiplication algorithm

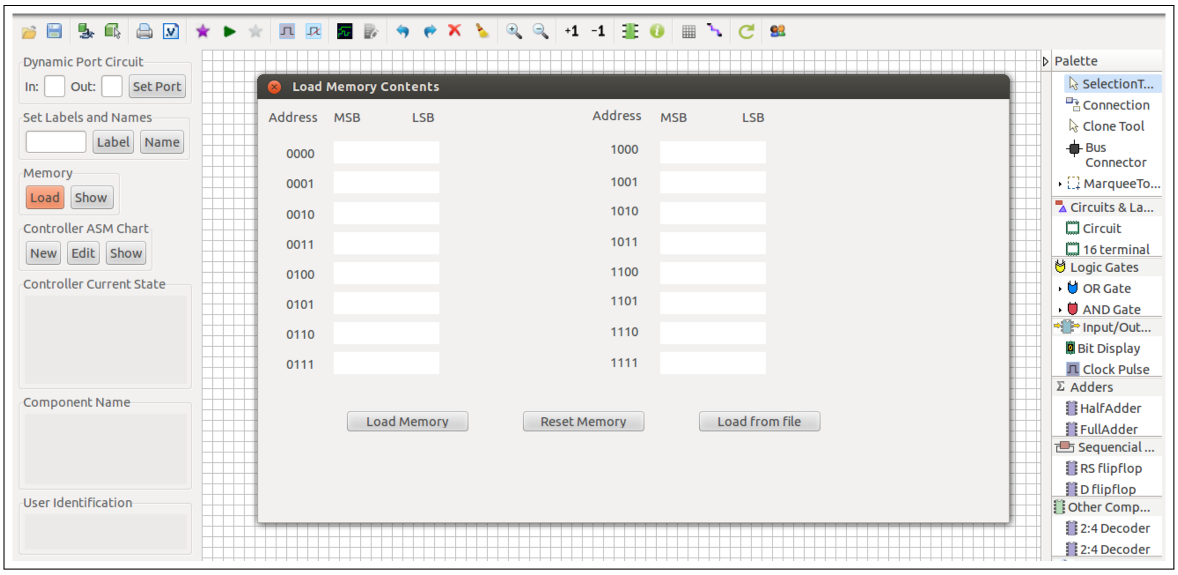


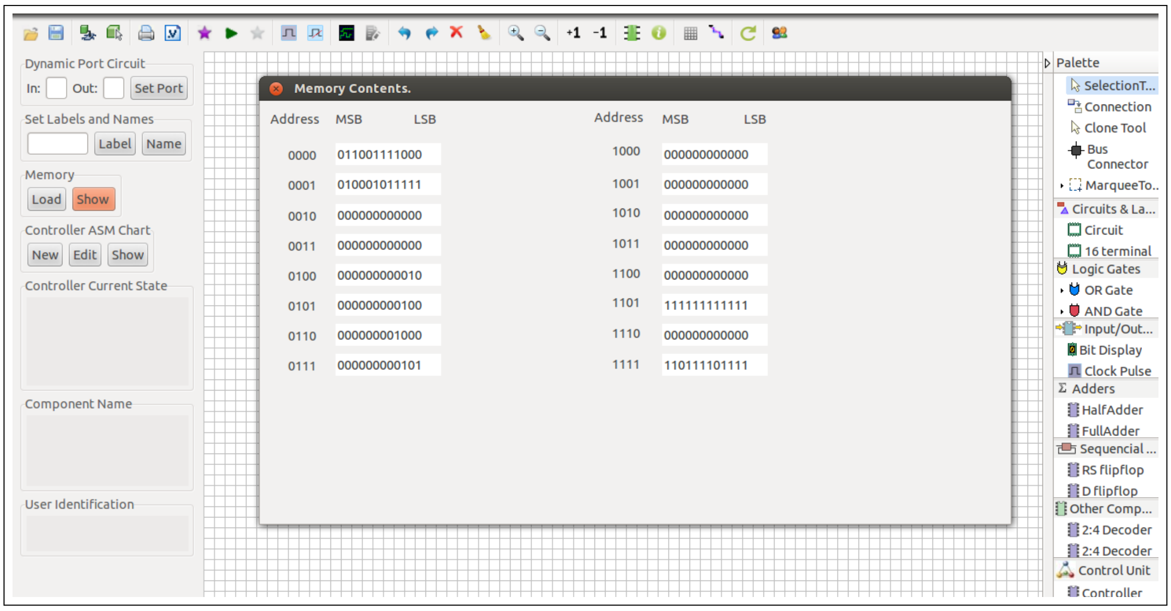
Figure 3.6: Interface to load working working memory.

Figure 3.13 shows the flowchart for shift and add multiplication of two 8-bit data. The multiplication algorithm will produce 16 bits output. In the flowchart, A[7:0] is the accumulator, M[7:0], Q[7:0] and prod[15:0] hold the multiplicand, multiplier and final product respectively. cnt[3:0] is used for loop count and initialized to the binary value corresponding to the integer value 8. The loop will iterate exactly for 8 times. It can be clearly observed that the flowchart contains some arithmetic operations, logical shift operation performed on some registers and conditional branching. Therefore implementation of this computer arithmetic algorithm in a digital circuit will certainly contain a data path and a controller where all the data transformations will actuated through control signals generated by the control unit.

Implementation of the shift and add multiplier

In order to perform the data transformations, the data path will have components such as accumulator, registers, arithmetic logic unit (ALU), adder units, shift registers, counters, comparators. The overall circuit is built through the hierarchical bottom up approach. All the significant components (including the controller) of the circuit is built individually and so that they can be tested to ensure that every component behaves correctly. Once every component is built correctly, the data path is built using the previously built components and then the data path is checked if it is working properly or not by providing the control signals manually. Once the data path is built and tested for its functional behaviors, it is then integrated with the controller and finally the behavior of the complete multiplier circuit is tested.

Figure 3.14 shows the 8-bit accumulator designed for this multiplier data path. The shifting of the data stored in the accumulator is employed in the accumulator circuit. The pin configurations of the encapsulated tool components (components provided by the tool) used in the circuit such as



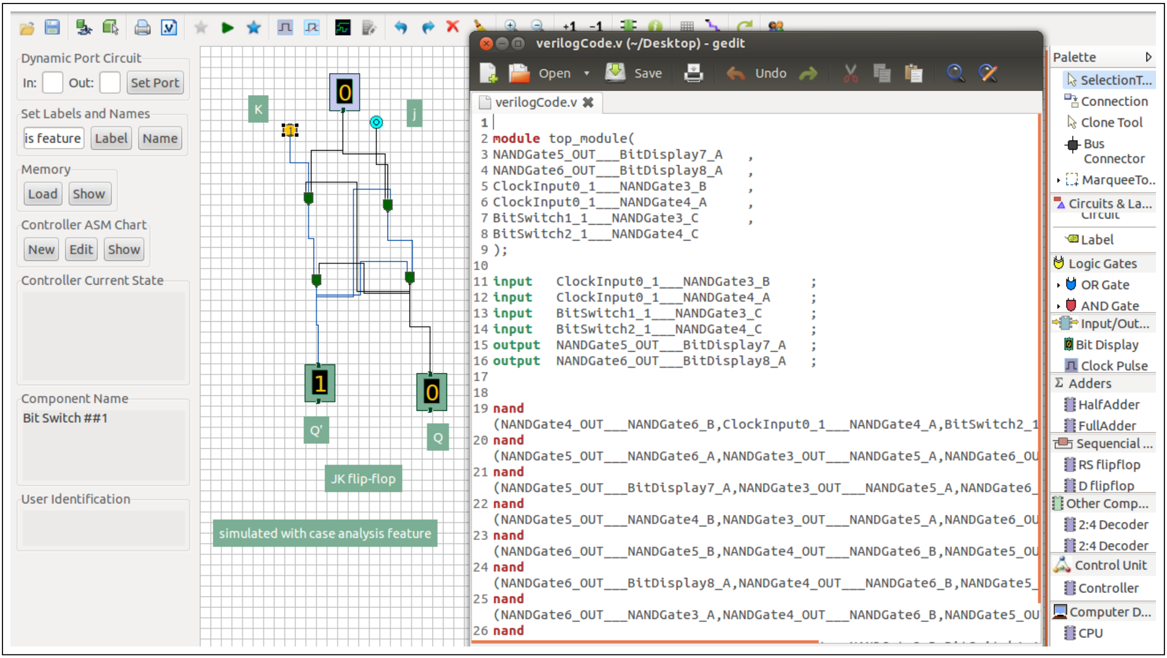
Address	MSB LSB	Address	MSB LSB
0000	011001111000	1000	000000000000
0001	010001011111	1001	000000000000
0010	000000000000	1010	000000000000
0011	000000000000	1011	000000000000
0100	000000000010	1100	000000000000
0101	000000000100	1101	111111111111
0110	000000001000	1110	000000000000
0111	000000000101	1111	110111101111

Figure 3.7: Working memory content.

bidirectional shift registers with 4-bit parallel load, multiplexers are also shown. The ALU for the implemented multiplier is shown in Figure 3.15 which uses the tool components such as full adders, multiplexers, basic gates.

In order to build the controller for the multiplier circuit, first a control chart is designed. Conventionally, after designing the control chart, the controller circuit is derived from the chart using classical method or one hot method which then is realized with the flip flops and basic gates. For complex control chart, the derived control circuit is also complex with several components and hence realization of such circuit becomes cumbersome. However, the time and effort of such controller circuit realization from the state chart can be reduced using the automatic encapsulated control unit generation (from user specified state chart) feature of the COLDVL tool as mentioned earlier in this chapter. Figure 3.16 shows the control state chart designed for the multiplier circuit and specified in the interface to build the controller unit by the tool. After entering the state chart, the encapsulated controller component is instantiated in the graphical editor. The controller component is then tested by giving the manual inputs and the state transitions are visualized using another feature provided in the tool which extracts the state model from the state chart and dynamically shows the state transitions during simulation. Figure 3.17 shows the individual controller component generated from the state chart and the state transitions in its corresponding state model.

After building the major components of the data path and testing their correctness, the data path has to be built using those components. However, the space limitation of any graphical imposes limitation on the size of the circuit. As it it can be seen from the figures 3.14 and 3.15 that the accumulator and ALU designed for the multiplier data path is itself large, and integrating both along with other components will result much large circuit which may not fit within the space of the graphical editor. This serious problem can be overcome by using the *save as component* feature



```

module top_module(
NANDGate5_OUT___BitDisplay7_A    ,
NANDGate6_OUT___BitDisplay8_A    ,
ClockInput0_1___NANDGate3_B      ,
ClockInput0_1___NANDGate4_A      ,
BitSwitch1_1___NANDGate3_C       ,
BitSwitch2_1___NANDGate4_C
);

input   ClockInput0_1___NANDGate3_B     ;
input   ClockInput0_1___NANDGate4_A     ;
input   BitSwitch1_1___NANDGate3_C      ;
input   BitSwitch2_1___NANDGate4_C      ;
output  NANDGate5_OUT___BitDisplay7_A   ;
output  NANDGate6_OUT___BitDisplay8_A   ;


nand
  (NANDGate4_OUT___NANDGate6_B,ClockInput0_1___NANDGate4_A,BitSwitch2_1
nand
  (NANDGate5_OUT___NANDGate6_A,NANDGate3_OUT___NANDGate5_A,NANDGate6_OU
nand
  (NANDGate5_OUT___BitDisplay7_A,NANDGate3_OUT___NANDGate5_A,NANDGate6_
nand
  (NANDGate5_OUT___NANDGate4_B,NANDGate3_OUT___NANDGate5_A,NANDGate6_OU
nand
  (NANDGate6_OUT___NANDGate5_B,NANDGate4_OUT___NANDGate6_B,NANDGate5_OU
nand
  (NANDGate6_OUT___BitDisplay8_A,NANDGate4_OUT___NANDGate6_B,NANDGate5_
nand
  (NANDGate6_OUT___NANDGate3_A,NANDGate4_OUT___NANDGate6_B,NANDGate5_OU
nand
```

Figure 3.8: Structural verilog code generated for the circuit.

of the tool which creates an encapsulated module out of an user designed circuit which may in turn contain other user created encapsulated components. With this feature, the user can build hierarchical abstracted modules recursively and thus larger circuits can be designed within a limited space. Using this feature of abstraction, the accumulator, ALU shown in Figures 3.14 and 3.15 are abstracted into encapsulated modules and then the multiplier data path is built using them along with the other tool components and the data path is shown in Figure 3.18. The components with label *New* on them in the data path are the user created encapsulated modules. The data path circuit is then provided manual inputs including to the control signal terminals and tested. When the data path functionality is found to be correct then the final multiplier circuit is built using the data path and the controller component which is shown in the Figure 3.19. In the multiplier circuit, again the encapsulated module corresponding to the data path is used.

3.5.3 Regular CPU design with multiple instructions

Figure 3.20 shows a CPU circuit (before simulation) having four instructions such as ADD, JZ (jump if zero), LOAD and NEG (negation) instructions along with its control unit. As the initial default wire value is unknown which is denoted by maroon color, the wire colors in the circuit is maroon before simulation. The CPU circuit also includes the working memory provided by the tool. The designer has to load the working memory with a binary program and data before initiating the simulation. A binary program along with data is loaded the screen shot is presented in the Figure 3.21. When the simulation is initiated, the CPU fetches the instructions and data, execute the instruction accordingly and write back the data in the memory. Figure 3.22 shows the memory

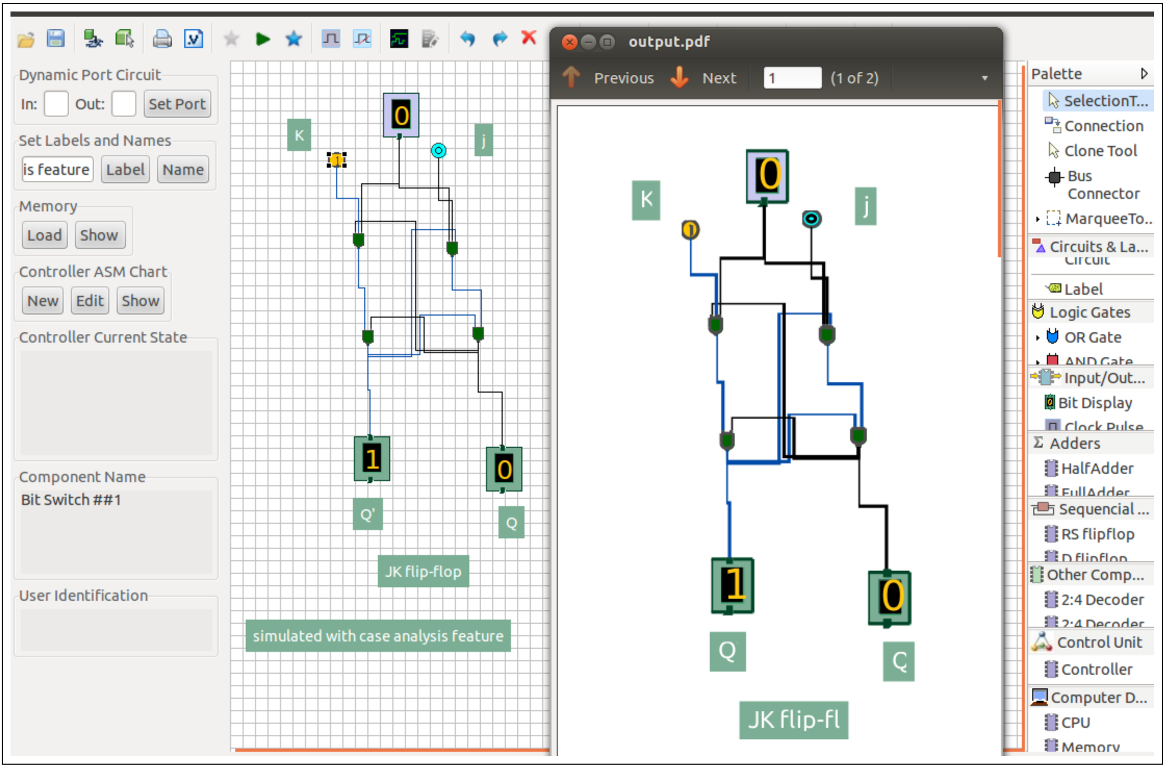


Figure 3.9: A circuit exported to the pdf format.

content and controller state model during the execution of the binary program present in the memory.

3.5.4 Usability of COLDVL tool

COLDVL tool as a teaching aid

The case analysis feature along with the feature of simulation while reconstructing the circuit in an ordered manner is developed in the COLDVL tool for supporting the learning experience. These features are needed in order to accomplish the experiment, *Synthesis of flip flops* which provides a detail theory. The tool identifies whether a circuit designed by the user follows the Huffman structure or not. There is also a feature of detecting the possible occurrence of race around conditions in the circuit before simulation which focuses on the fact that creating the sequential part of a data path without master-slave flip flop may produce race around condition in the circuit (where non-master-slave flip flop is used). These features help students learning a safe circuit design. Apart from these features, some of the features mentioned in the front end of the tool, such as, control signal generation from user-given ASM chart, which essentially helps teaching the controller specification and synthesis, by reducing the design complexity of hardwired control unit and thus giving students more scope to concentrate upon learning issues regarding control unit state chart design, structural verilog netlist generation, creating and reusing user-defined encapsulated modules which help students in their learning process and teachers for their teaching as they can add their own experiment with this facility.

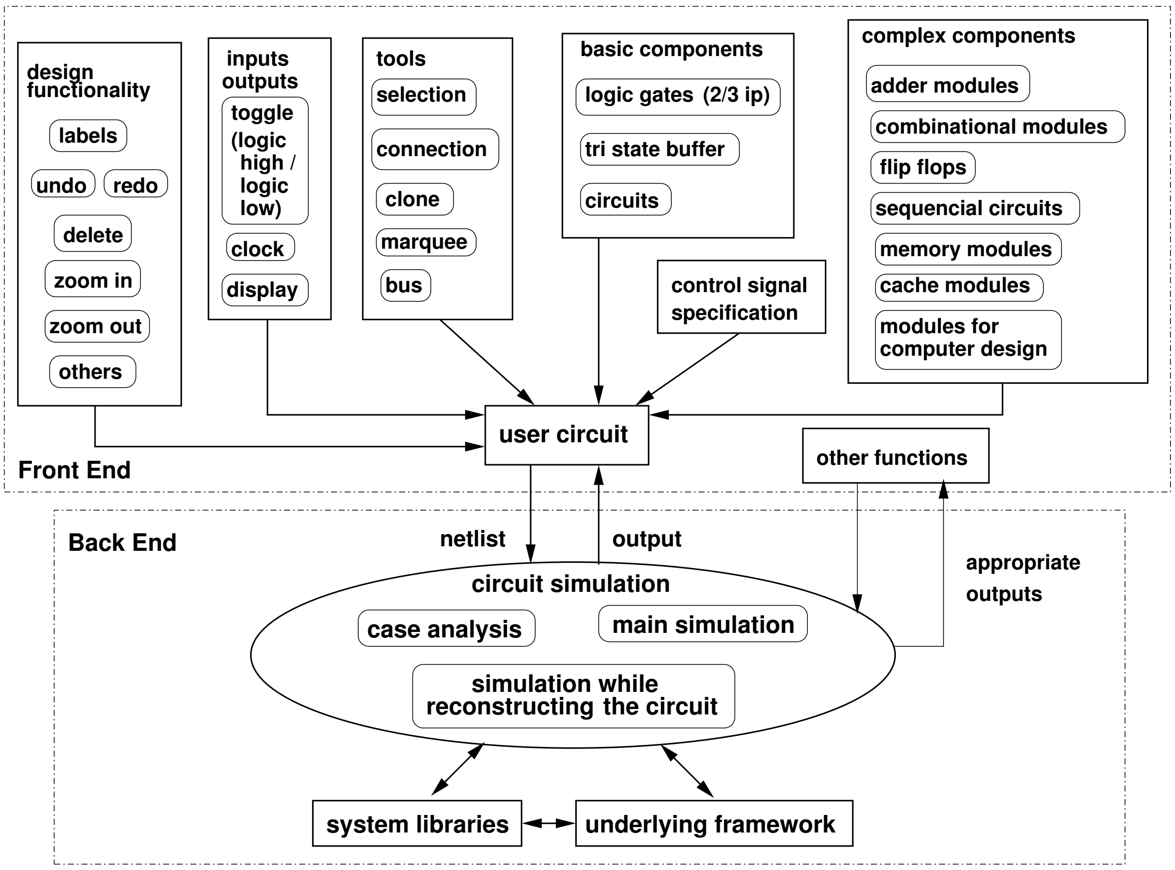


Figure 3.10: System architecture of the COLDVL tool.

Use of COLDVL tool

This tool has been used to conduct undergraduate and postgraduate level laboratory course at IIT Kharagpur. Students are given assignments, after designing and building their circuit and saving with their identification (using name and roll number for unique identification) they have uploaded the file to the *Web-Based Course Management system* [36] of IIT Kharagpur over Internet, for manual evaluation. A few intern students have also used COLDVL tool to accomplish their assignments. Following are the assignments which students have accomplished using the COLDVL tool.

1. Implementation, simulation and analysis of the restoring divider and non-restoring divider where divisor is of five bits and the dividend is of ten bits along with the comparison of these two types of dividers for cost and speed.

2. Implementation, simulation and analysis of the following along with the comparison of these two types of adders for cost and speed.

   (a) Ripple carry adder (RCA), for four bits

   (b) Carry lookahead adder (CLA), for four bits, generating carry generate and carry propagate functions

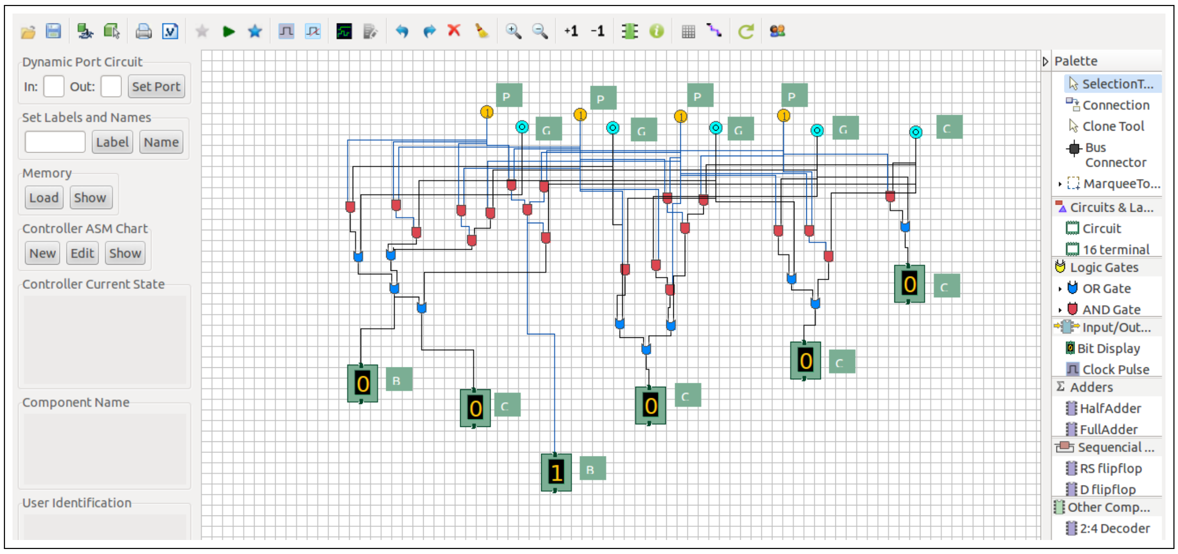


Figure 3.11: 4-bit carry-look-ahead adder designed in the COLDVL tool.

   (c) Block carry lookahead adder (BCLA), to add two 16-bit numbers, using 4-bit CLAs and 4-bit BCLA units

3. Implementation, simulation and analysis of shift and add multiplier, doing the multiplication for two 8-bit 2's complement number satisfying the following criteria.

   (a) The sign necessary for arithmetic shifting is properly generated using a suitable sign generation logic

   (b) The design should be modular and should have two top-level components, viz samDP and samCtrl ensuring that the modules are well labeled (pins should be labeled to indicate their functionality)

   (c) The data path should have all the data processing elements, appropriate data path control signal inputs and appropriate status signal outputs

   (d) The controller should be designed as a finite state machine, using the status signals of the data path as its inputs and generating outputs which are used to drive the data path control signals

4. Implementation, simulation and analysis of a radix-4 Booth's multiplier for two 8-bit numbers with a comparison with the shift and add multiplier and also the radix-2 Booth's multiplier for cost and speed.

5. Designing a direct mapped cache with the following specifications.

   (a) Four address lines

   (b) Two data lines

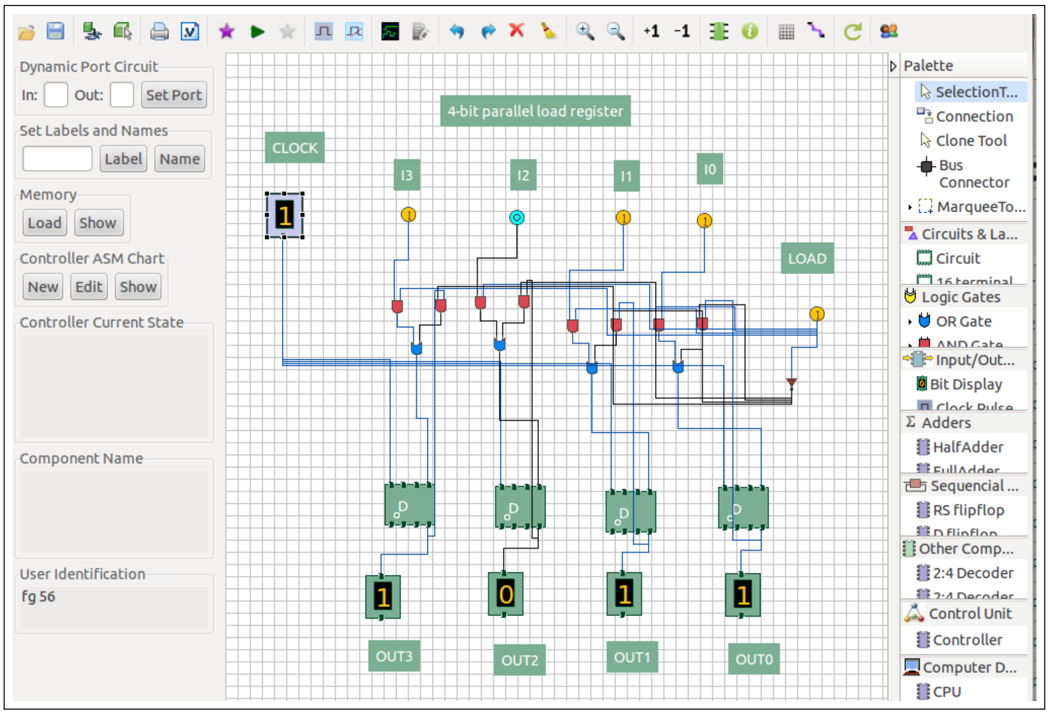


Figure 3.12: 4-bit parallel load register designed in the COLDVL tool using D flip flop.

(c) Cache flush control lines

(d) Cache miss indication line

(e) Read/write control lines

(f) Four cache lines each of a single 2-bit word

6. Implementation, simulation of a single instruction CPU having SBN (subtract and branch if negative) instruction along with its controller.

7. Implementation, simulation of a multiple instruction CPU having ADD, JZ (jump if zero), LOAD and NEG (negation) instructions along with its controller.

3.6 Conclusion

In this chapter the features of the COLDVL tool are described along with the corresponding interfaces. The tool architecture is explained briefly. Few case studies have presented in order to explain how the tool features developed are helpful in order to learn the computer organization and logic design. The use of COLDVL tool in conducting undergraduate and postgraduate level laboratory courses in IIT Kharagpur has also been mentioned along with the descriptions of the assignments performed by the students.

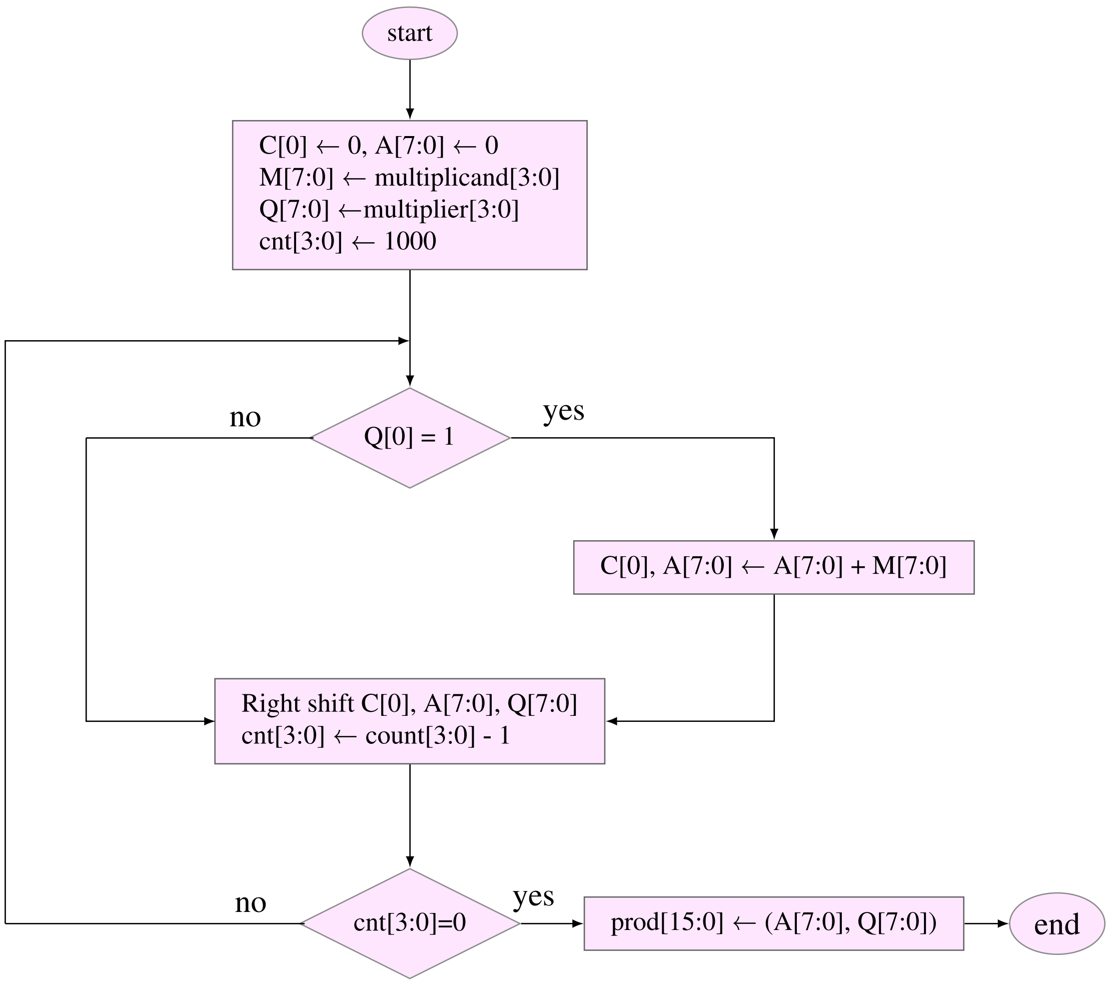
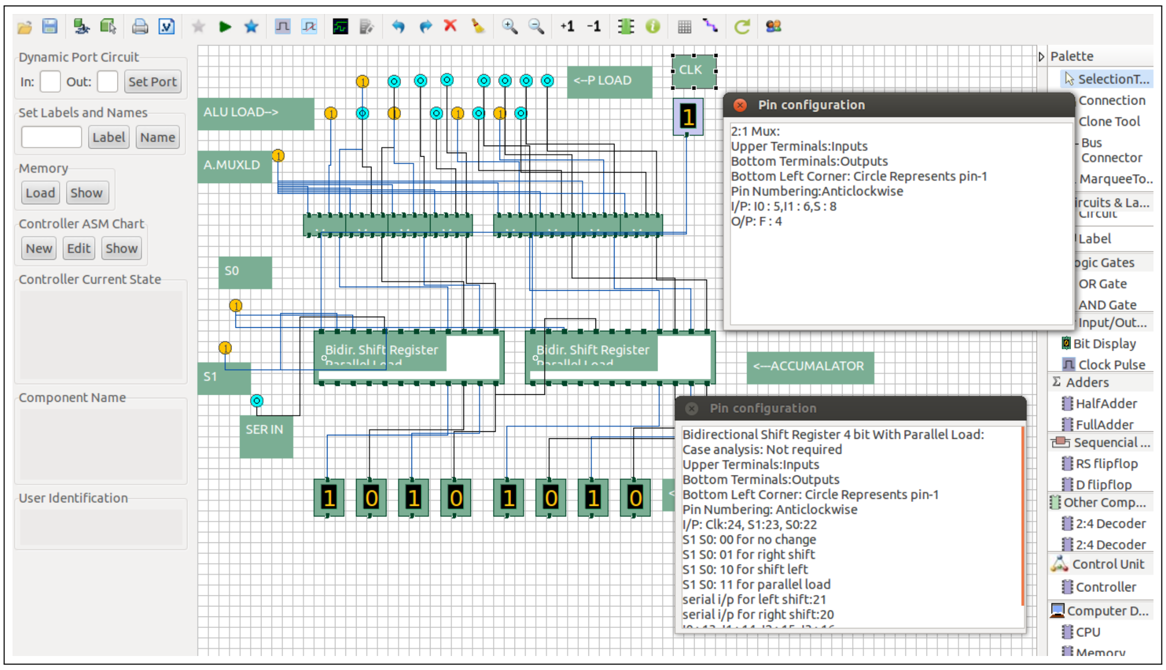

start
C[0] ← 0, A[7:0] ← 0
M[7:0] ← multiplicand[3:0]
Q[7:0] ←multiplier[3:0]
cnt[3:0] ← 1000
no
Q[0] = 1
yes
C[0], A[7:0] ← A[7:0] + M[7:0]
Right shift C[0], A[7:0], Q[7:0]
cnt[3:0] ← count[3:0] - 1
no
cnt[3:0]=0
yes
prod[15:0] ← (A[7:0], Q[7:0])
end

Figure 3.13: Flowchart for shift and add multiplication of two 8-bit data



Pin configuration

2:1 Mux:
Upper Terminals:Inputs
Bottom Terminals:Outputs
Bottom Left Corner: Circle Represents pin-1
Pin Numbering:Anticlockwise
I/P: I0 : 5,I1 : 6,S : 8
O/P: F : 4

Pin configuration

Bidirectional Shift Register 4 bit With Parallel Load:
Case analysis: Not required
Upper Terminals:Inputs
Bottom Terminals:Outputs
Bottom Left Corner: Circle Represents pin-1
Pin Numbering: Anticlockwise
I/P: Clk:24, S1:23, S0:22
S1 S0: 00 for no change
S1 S0: 01 for right shift
S1 S0: 10 for shift left
S1 S0: 11 for parallel load
serial i/p for left shift:21
serial i/p for right shift:20

Figure 3.14: Accumulator circuit designed for 8-bit shift and add multiplier.

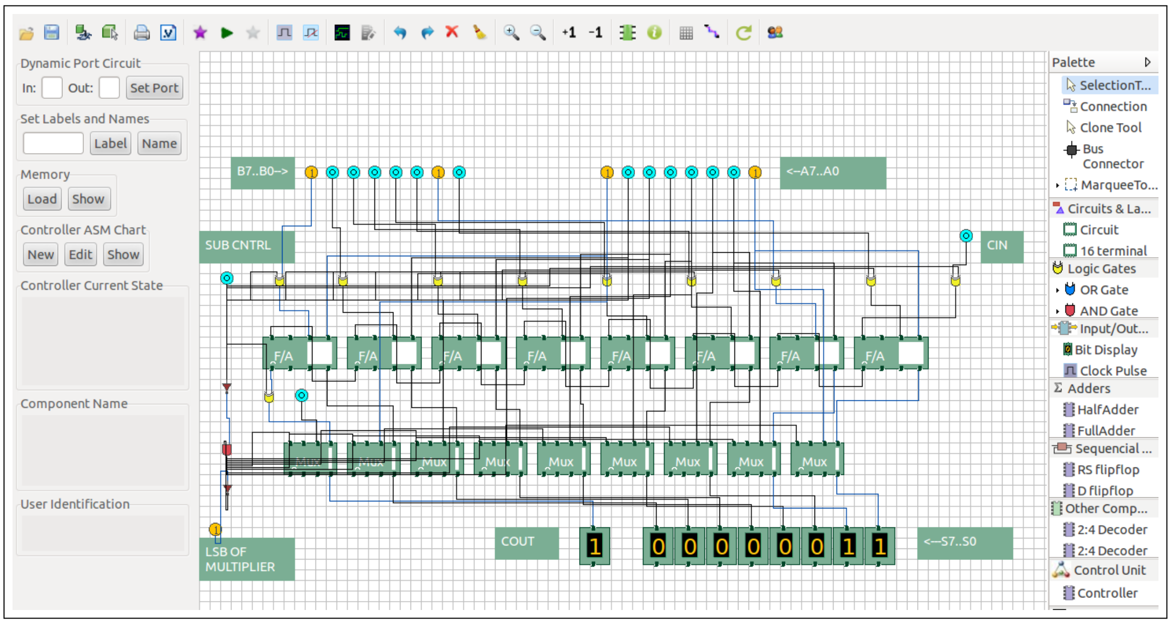
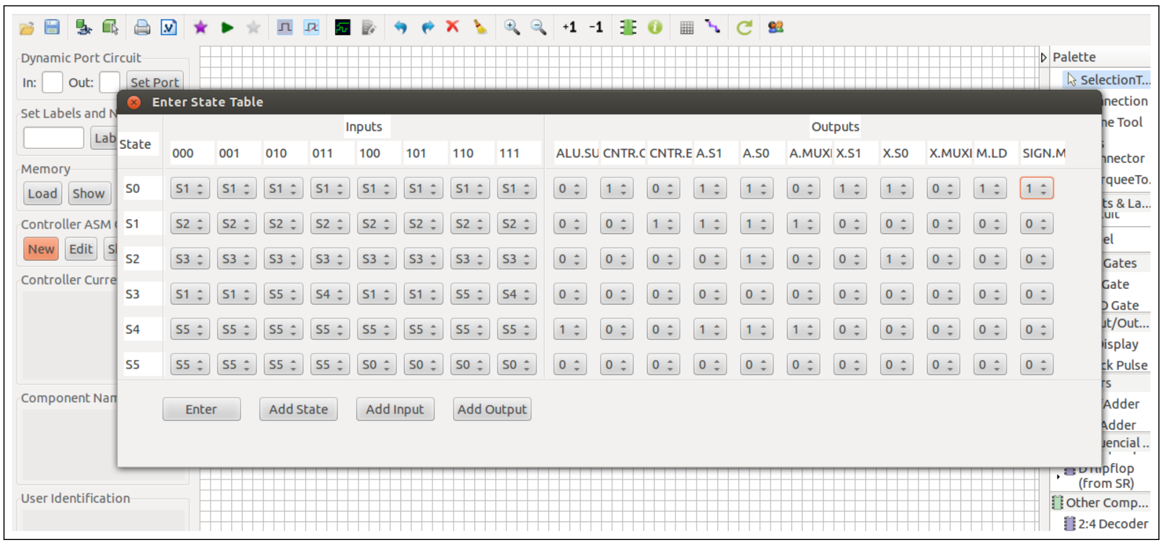
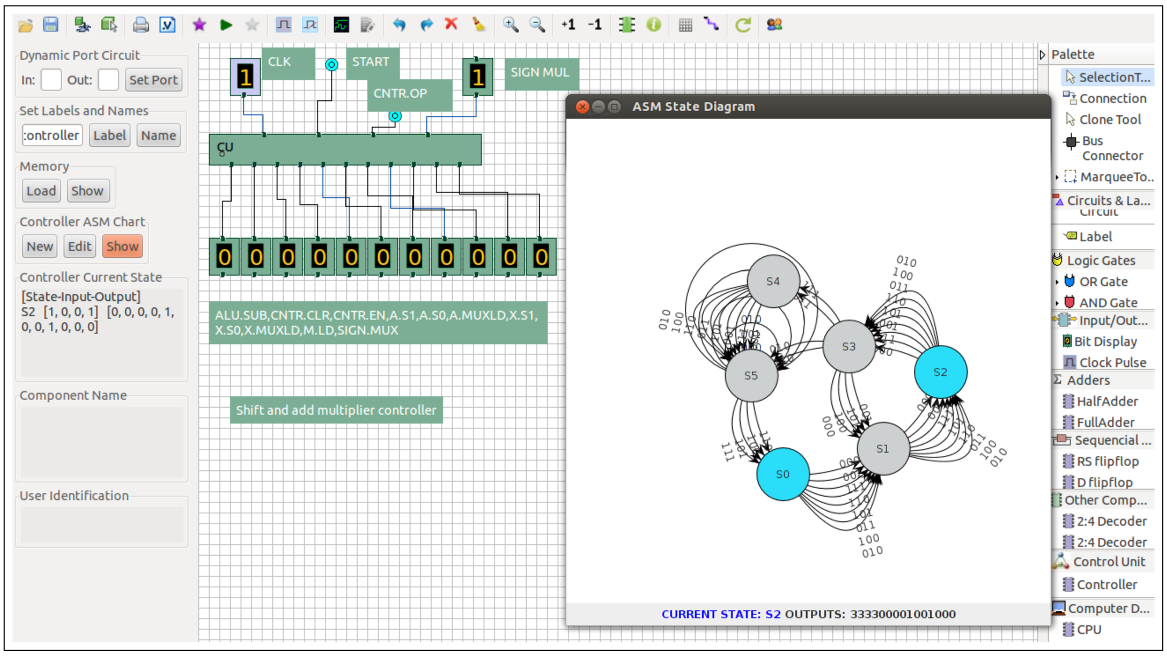


Figure 3.15: ALU circuit designed for 8-bit shift and add multiplier.



	Inputs								Outputs										
State	000	001	010	011	100	101	110	111	ALU.SU	CNTR.C	CNTR.E	A.S1	A.S0	A.MUXI	X.S1	X.S0	X.MUXI	M.LD	SIGN.M
S0	S1	S1	S1	S1	S1	S1	S1	S1	0	1	0	1	1	0	1	1	0	1	1
S1	S2	S2	S2	S2	S2	S2	S2	S2	0	0	1	1	1	1	0	0	0	0	0
S2	S3	S3	S3	S3	S3	S3	S3	S3	0	0	0	0	1	0	0	1	0	0	0
S3	S1	S1	S5	S4	S1	S1	S5	S4	0	0	0	0	0	0	0	0	0	0	0
S4	S5	S5	S5	S5	S5	S5	S5	S5	1	0	0	1	1	1	0	0	0	0	0
S5	S5	S5	S5	S5	S0	S0	S0	S0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.16: Control state chart for 8-bit shift and add multiplier.



[State-Input-Output]
S2 [1, 0, 0, 1] [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0]

ALU.SUB,CNTR.CLR,CNTR.EN,A.S1,A.S0,A.MUXLD,X.S1,X.S0,X.MUXLD,M.LD,SIGN.MUX

CURRENT STATE: S2 OUTPUTS: 333300001001000

Figure 3.17: Controller designed for 8-bit shift and add multiplier.

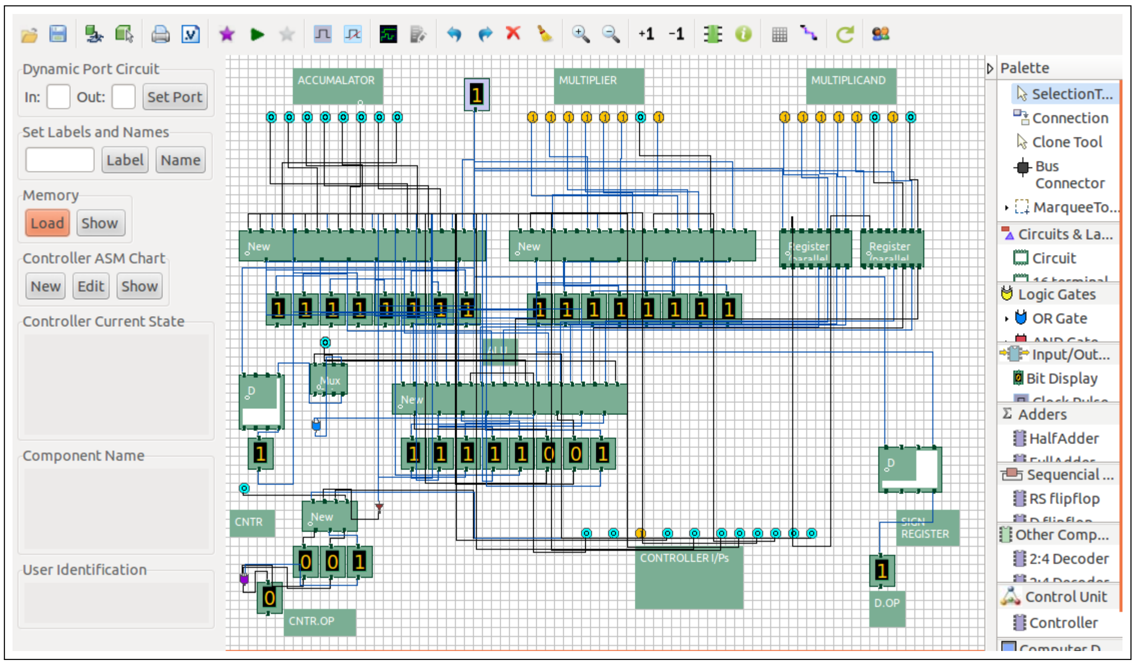


Figure 3.18: The data path designed for 8-bit shift and add multiplier.

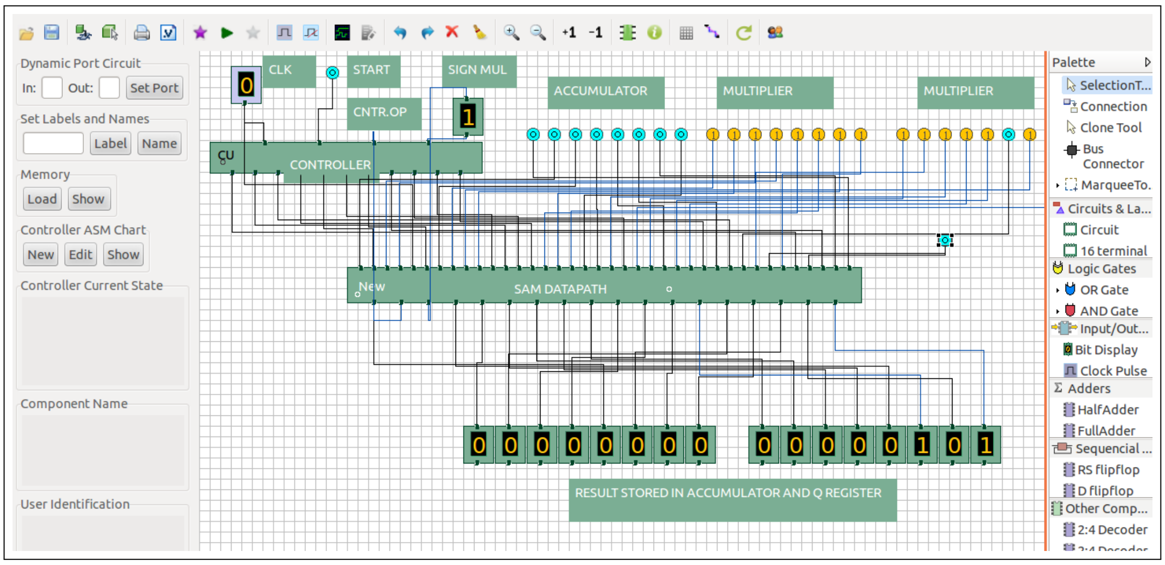


Figure 3.19: 8-bit shift and add multiplier having a controller and data path.

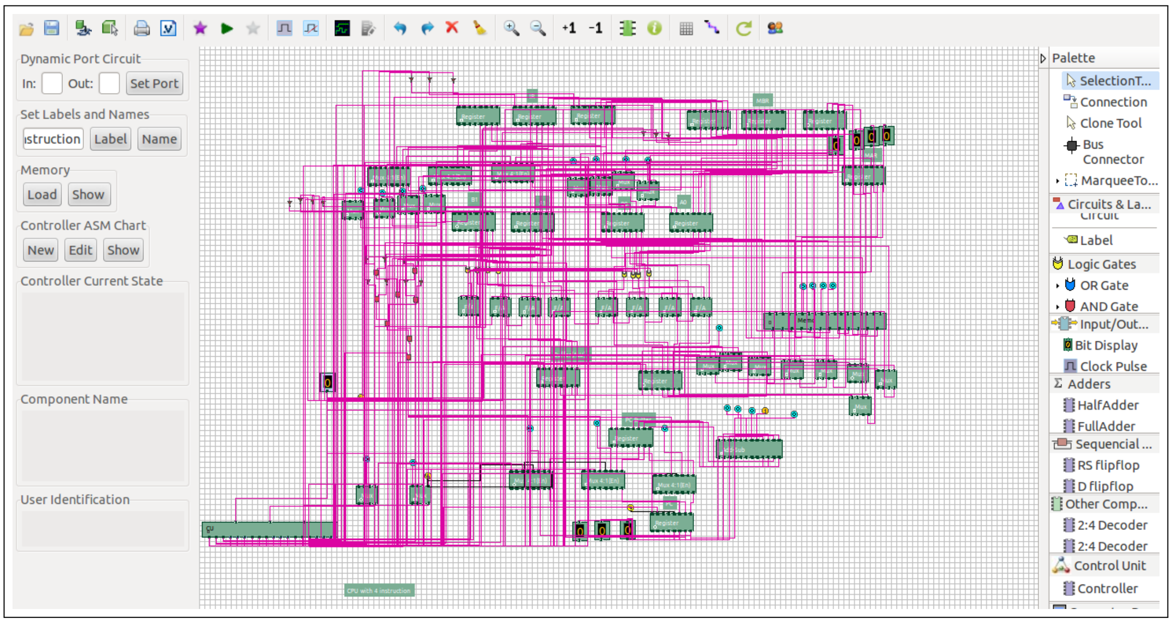
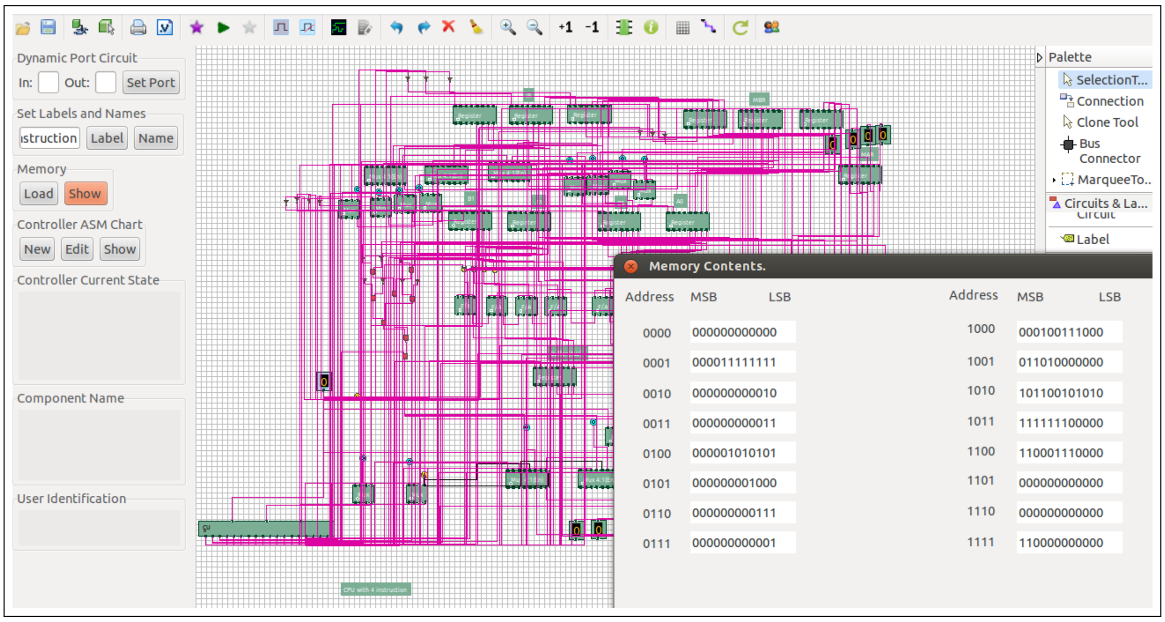


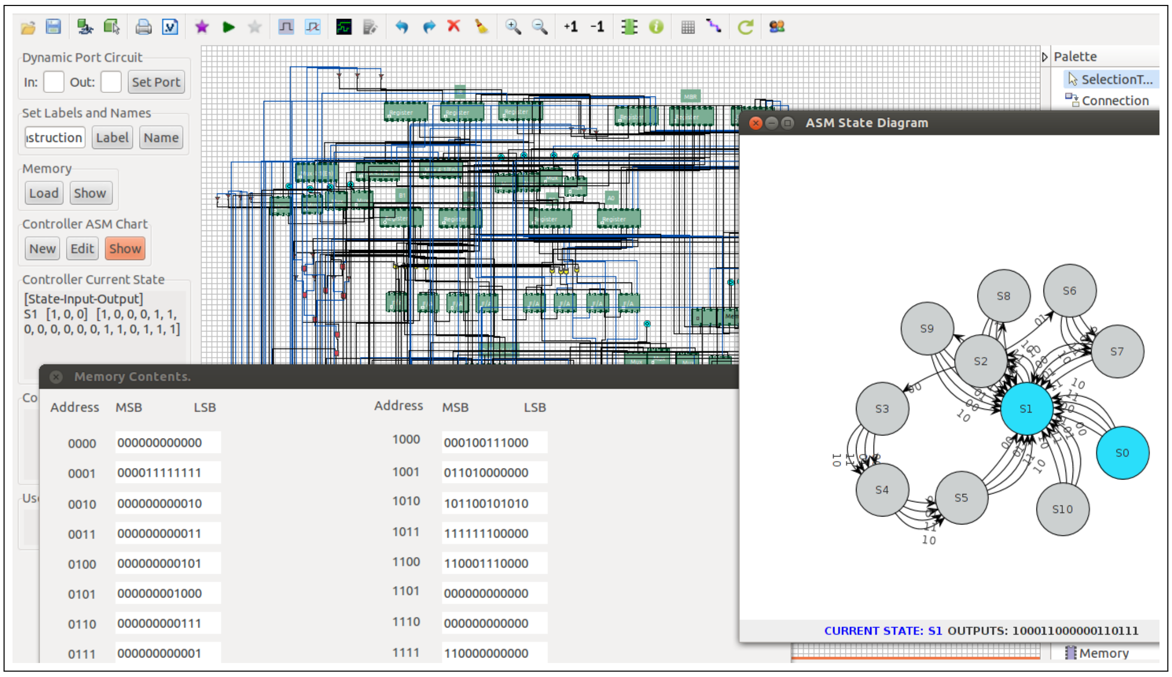
Figure 3.20: A regular CPU having four instruction with controller unit and working memory.



Memory Contents.

Address	MSB LSB	Address	MSB LSB
0000	000000000000	1000	000100111000
0001	000011111111	1001	011010000000
0010	000000000010	1010	101100101010
0011	000000000011	1011	111111100000
0100	000001010101	1100	110001110000
0101	000000001000	1101	000000000000
0110	000000000111	1110	000000000000
0111	000000000001	1111	110000000000

Figure 3.21: Working memory of a regular CPU having four instruction is loaded with binary program and data.



[State-Input-Output]
S1 [1, 0, 0] [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1]

Memory Contents.

Address	MSB LSB	Address	MSB LSB
0000	000000000000	1000	000100111000
0001	000011111111	1001	011010000000
0010	000000000010	1010	101100101010
0011	000000000011	1011	111111100000
0100	000000000101	1100	110001110000
0101	000000001000	1101	000000000000
0110	000000000111	1110	000000000000
0111	000000000001	1111	110000000000

CURRENT STATE: S1 OUTPUTS: 100011000000110111

Figure 3.22: Working memory content and controller state diagram of a regular CPU having four instruction after execution of the binary program loaded in the working memory.

Chapter 4

Back end of the COLDVL tool

4.1 Introduction

This chapter focuses on the back end simulation related methodologies along with the algorithms, case studies and illustrative example. The efficient simulation for circuits conforming to Huffman model has been discussed in detail along with newly developed partitioning technique in a gate level circuit in order to identify the conformance of a circuit with the HUffman model. This chapter also describes the newly developed techniques for handling unknown signal values of some nets which remain indeterminate after regular round of simulation. The implementation of the COLDVL tool along with its results and scope have also been included.

4.2 Some issues related to simulation of circuits

4.2.1 Efficient simulation

COLDVL tool uses an efficient algorithm which is a combined approach of standard event driven [13] and topologically ordered levelized simulation [60] technique which is also known as levelized simulation in the literature, to simulate sequential circuits restricted to Huffman model [29]. Standard event driven technique is used for the simulation of sequential circuits not conforming to the Huffman model (detailed in subsequent sections) and levelized simulation technique is used to simulate combinational circuits. Levelized approach is essentially an ordered simulation technique where a component is simulated after all of its inputs are available. In order to do so, the circuit must be acyclic so that the components of the circuit can be ordered topologically and simulated according to that order. In contrast, a component, in event driven technique is simulated repeatedly if any one of its input signals changes, until it reaches to its fixed point, thus allowing simulation of both cyclic and acyclic circuits. However, event driven approach may encounter unnecessary events for some kind of circuits, yielding bad performance. For example, event driven technique generates extra events for purely combinational circuit which is supposed to be simulated in levelized

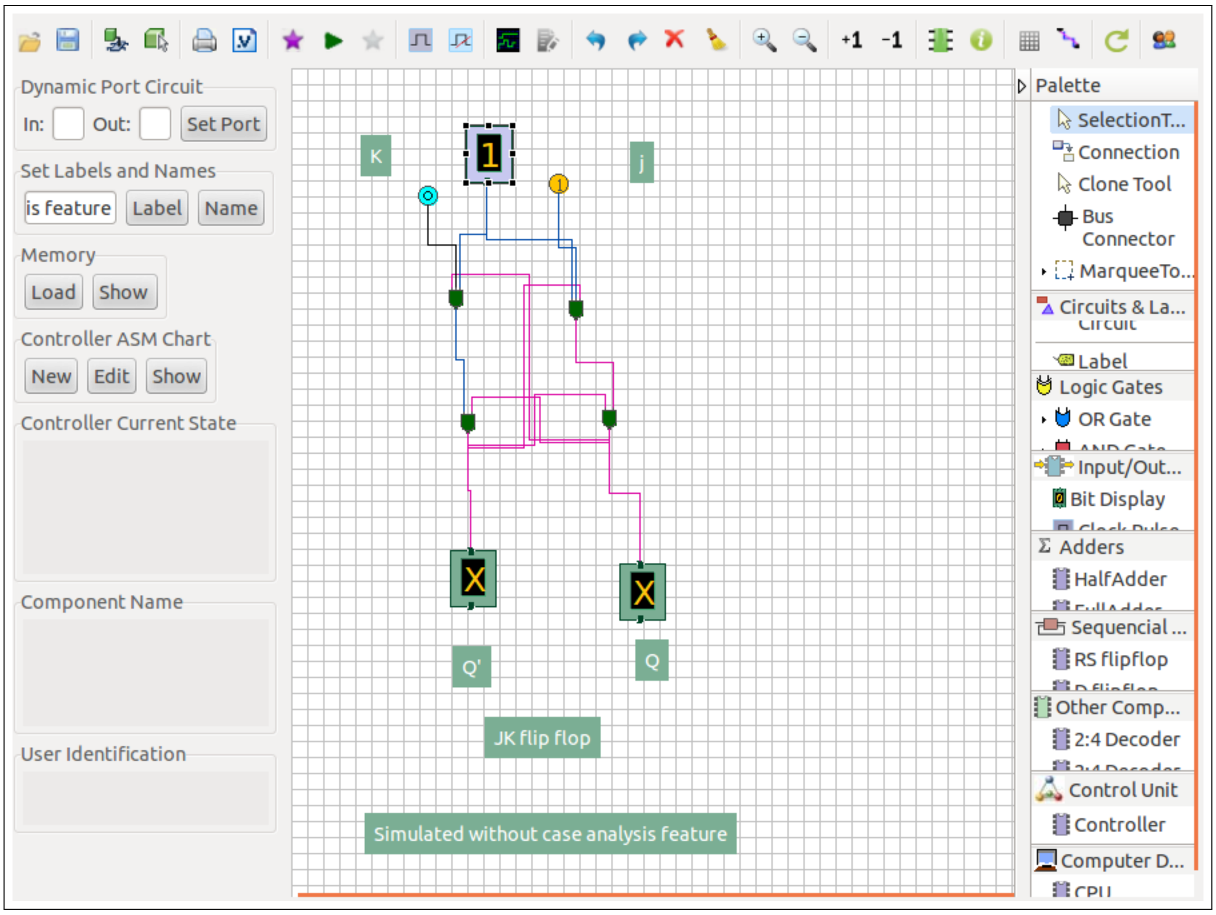


Figure 4.1: JK flip flop (NAND implementation) after simulation gives indeterminate result.

manner in order to achieve best performance. Similarly, if a circuit has a combination of both sequential and combinational domains, event driven simulation will yield non-optimized performance too. Because, if the sequential domain and the combinational domain are identified within a circuit and if the simulation of the circuit can be scheduled in a manner such that the event driven and levelized approaches are applied to the sequential and combinational domain respectively, then the simulation performance will be better than simulating the whole circuit in event driven manner. Addressing these issues, COLDVL tool achieves better simulation performance for sequential circuits conforming to the Huffman model, shown in Figure 4.11(a), by simulating the combinational part of the circuit in levelized manner and simulating the sequential part using event driven approach. For this purpose, before simulation, first the circuit is partitioned into sequential and combinational part. The sequential part contains all the structural flip flops and behavioral sequential components. Then it is checked whether the circuit follows the Huffman structure or not. If the circuit conforms the Huffman model, combined simulation approach is used. The approach is restricted to the Huffman model, because identification of sequential and combinational domain in a gate level sequential circuit is possible if the circuit conforms the Huffman model and thereafter, simulation can be scheduled between those two domains.

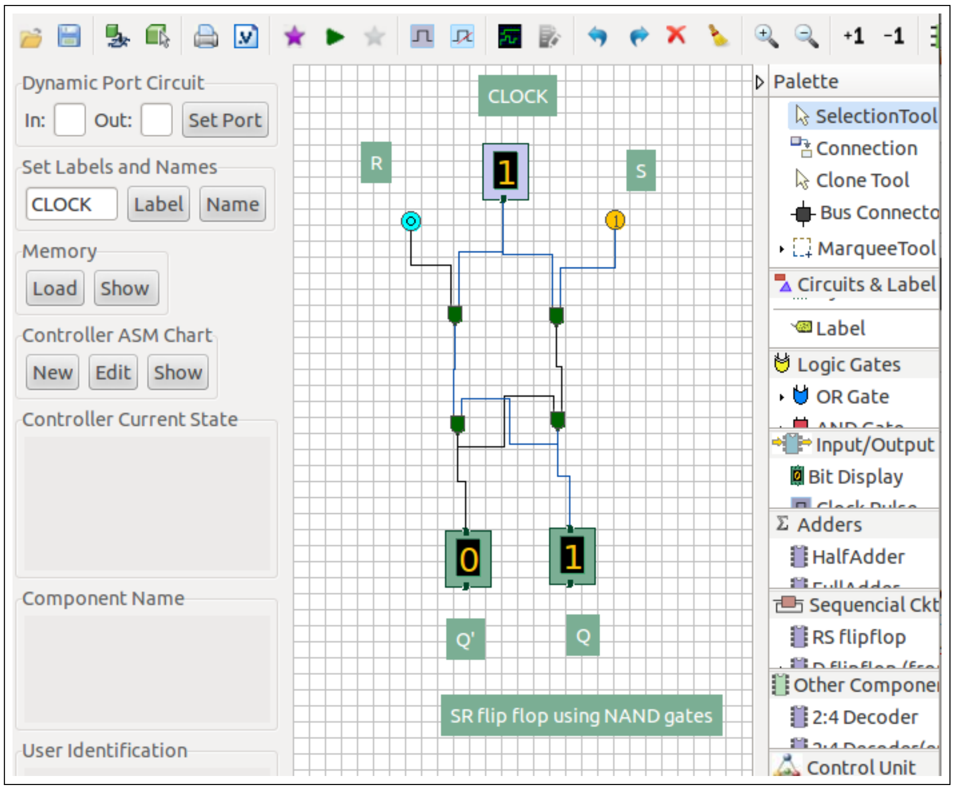


Figure 4.2: SR flip (NAND implementation) flop after simulation gives proper result.

4.2.2 Resolving indeterminate values of structural memory elements

Case based analysis

As the tool uses unknown as the default initial value for all logic signals, for some specific gate level flip flop circuits, standard event driven simulation technique can not produce definite output for the given inputs when the circuit should give definite outputs corresponding to those inputs. For example, the aforesaid situation occurs in the JK flip flop (shown in Figure 4.6) with the corresponding circuit shown in Figure 4.1. Where as the simulation gives determinate value for SR flip flop, D flip flop realized from SR flip flop, edge triggered D flip flop which is shown in Figures 4.2, 4.3 and 4.4 respectively. The JK flipflop circuit (shown in Figure 4.6(a)) has been tested in some of the public domain and commercial simulators having unknown as one of its logic values which exhibits unknown outputs with inputs such as J = 1, K = 0 at Clock = 1. Figure 4.6(a) shows the circuit diagram of the JK flip flop before simulation (maroon colored dashed line denotes unknown value) and Figure 4.6(d) shows the verilog code for the JK flip flop. Figure 4.6(b) shows the same flip flop after standard event driven simulation whose output remains unknown (blue colored solid line and black dotted line denotes logic value 1 and 0 respectively). Although this situation may not be confusing for an expert CAD tool user, however, a student may get confused as the circuit in bread board gives definite outputs for appropriate definite inputs while a simulator (using the standard event driven simulation technique) gives unknown output for the same circuit with the same inputs. Hence, our

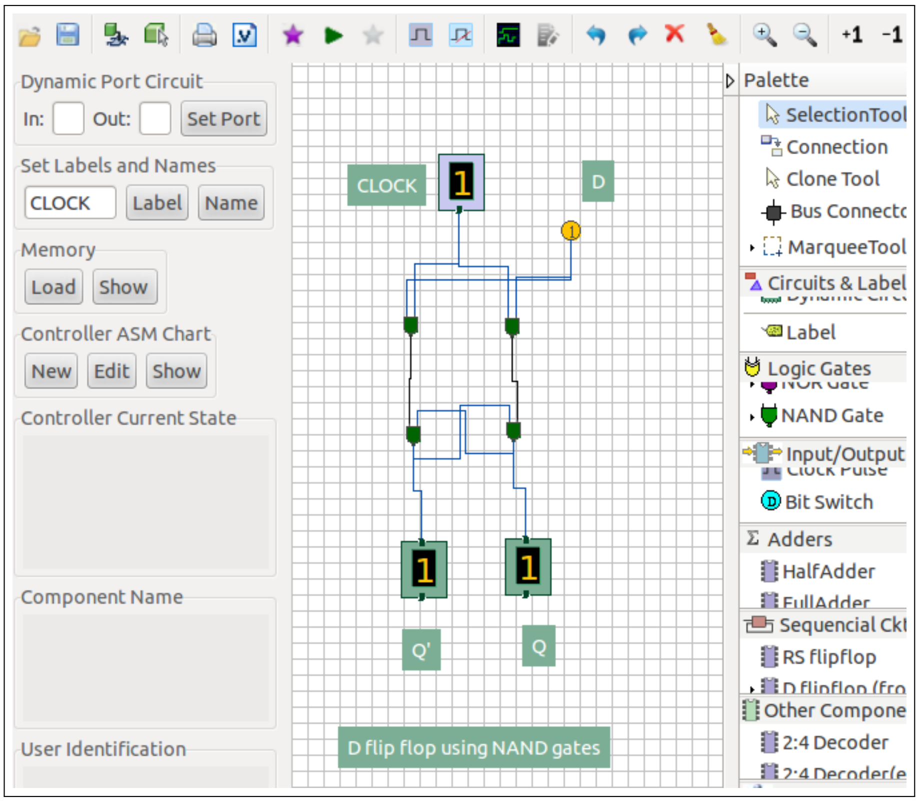


Figure 4.3: D flip flop (from SR flip flop) after simulation gives proper result.

simulation technique performs case analysis of unknown signal values of some nets which are part of the loops in structural memory elements which remain unknown after a regular round of standard event driven simulation. As unknown value indicates either logic value 0 or 1, therefore, during case analysis, unknown loops are resolved in all possible combinations of 0 and 1. For n unknown loops, case analysis will check 2^n combinations on the internal netlist representation of the original circuit. As, whenever a circuit is switched on, all unknown loops may take value either 0 or 1 arbitrarily, therefore, checking all possible combinations of 0 and 1 are necessary. To avoid state explosion, the case analysis is performed on each structural flip flop identified so far keeping the number of loops very small. The case analysis converges only when the values of the elements in the structural flip flop remain same in all cases, then the result is reflected in the display netlist i.e., original circuit. Convergence of case analysis essentially depicts that at the start up no matter how arbitrarily unknown loops are set, however, they will eventually reach to the same state. Figure 4.6(c) shows the resolved JK flip flop after case analysis converges.

Dealing with the failure of case analysis

The non-convergence of the case analysis may occur due to the legitimate initial uncertainty in the circuit such as, the value of a flip flop before it has been loaded. These types of non-convergence are eventually resolved as the circuit operates. However, the real problem of non-convergence occurs due to the failure to resolve the initial unknown in course of normal operation of the circuit. For

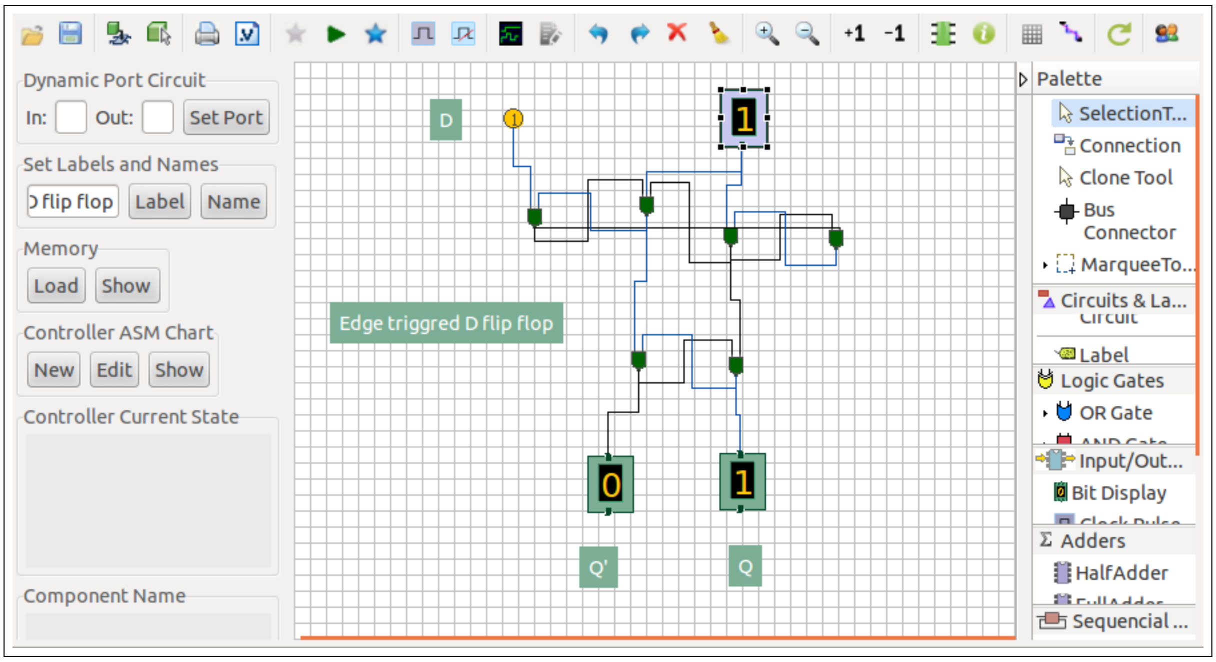
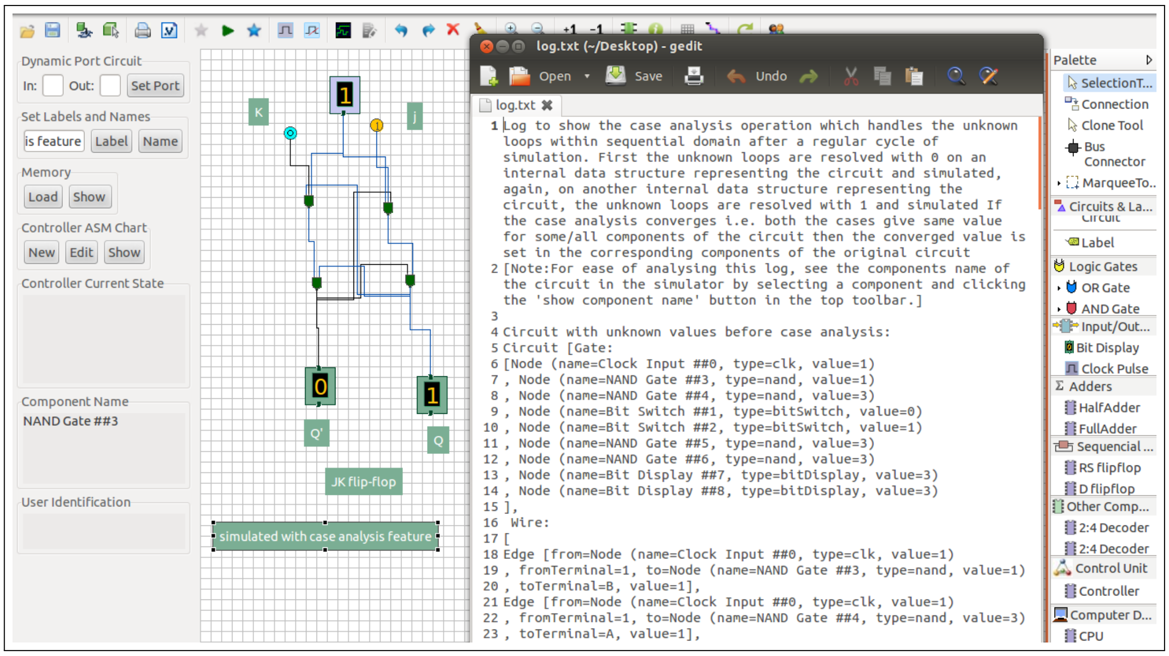


Figure 4.4: Edge triggered D flip flop after simulation gives proper result.

example, the master-slave JK flip flop (shown in Figure 4.10) remains non-convergent after case analysis and the corresponding circuit is shown in Figure 4.7. In all the states, the circuit does not give same output, for example, in one state the circuit may have the values shown in Figure 4.10(a), where at J = 0, the Q output of the master gives 1 while it should give 0, which essentially means that the whole flip flop malfunctions. This situation occurs because, before the value of the master flip flop is set to a definite value, the slave flip flop is set to a value in such manner which causes the master to malfunction.

For such non-converging circuits mentioned above, the circuit may be simulated with the elements' values of the flip flop from any case, chosen at random, which may lead to a malfunctioning circuit. This can also happen in real life, although the situation is rarely manifested. Therefore the tool is not in a position to make the desirable choice to avoid malfunctioning state of the circuit which is inconvenient for the user. Hence, as a solution the unknown circuits are resolved through the simulation while reconstructing the circuit. In this simulation approach, the circuit is reconstructed and simulated after each connection is made and for unconnected input terminals of gates, simulation is carried out with default input as 1 (logical value assumed when the terminal is floating). Even in this approach, the circuit may enter the malfunctioning state depending on the order in which the circuit is reconstructed and Figure 4.9 shows one such order of building the circuit. Figure 4.10(a) shows this type of situation for master-slave JK flip flop which is occurred due to the construction of slave before the master in a manner so that it can adversely affect the output of the master and the master malfunctions. Therefore, COLDVL tool simulates the circuit while reconstructing it in an ordered manner to prevent the circuit from entering such malfunctioning state. The order of reconstruction is determined by the breath-first-search starting from the clock, a netlist is constructed in ascending



Component Name: NAND Gate ##3

```
Log to show the case analysis operation which handles the unknown
loops within sequential domain after a regular cycle of
simulation. First the unknown loops are resolved with 0 on an
internal data structure representing the circuit and simulated,
again, on another internal data structure representing the
circuit, the unknown loops are resolved with 1 and simulated If
the case analysis converges i.e. both the cases give same value
for some/all components of the circuit then the converged value is
set in the corresponding components of the original circuit
[Note:For ease of analysing this log, see the components name of
the circuit in the simulator by selecting a component and clicking
the 'show component name' button in the top toolbar.]

Circuit with unknown values before case analysis:
Circuit [Gate:
[Node (name=Clock Input ##0, type=clk, value=1)
, Node (name=NAND Gate ##3, type=nand, value=1)
, Node (name=NAND Gate ##4, type=nand, value=3)
, Node (name=Bit Switch ##1, type=bitSwitch, value=0)
, Node (name=Bit Switch ##2, type=bitSwitch, value=1)
, Node (name=NAND Gate ##5, type=nand, value=3)
, Node (name=NAND Gate ##6, type=nand, value=3)
, Node (name=Bit Display ##7, type=bitDisplay, value=3)
, Node (name=Bit Display ##8, type=bitDisplay, value=3)
],
 Wire:
[
Edge [from=Node (name=Clock Input ##0, type=clk, value=1)
, fromTerminal=1, to=Node (name=NAND Gate ##3, type=nand, value=1)
, toTerminal=B, value=1],
Edge [from=Node (name=Clock Input ##0, type=clk, value=1)
, fromTerminal=1, to=Node (name=NAND Gate ##4, type=nand, value=3)
, toTerminal=A, value=1],
```

Figure 4.5: JK flip flop (NAND implementation) after case analysis gives proper output along with the case analysis log.

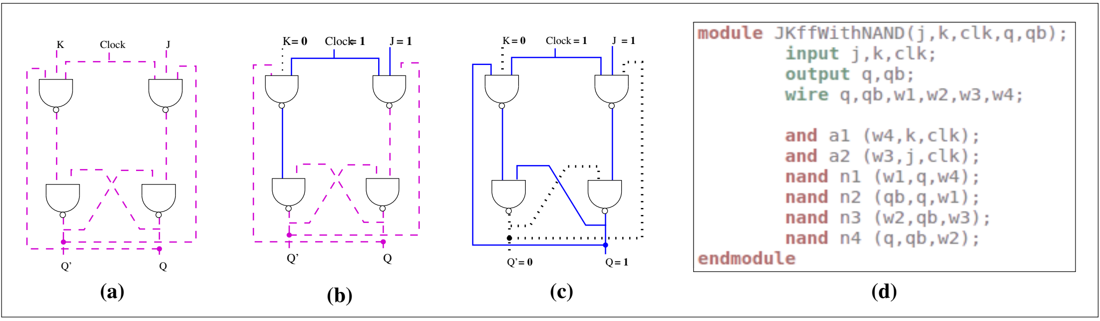
order of the labels which ensures that the master will be constructed before the slave. The ordering is shown in Figure 4.10(b) for master-slave JK flip flop and Figure 4.10(c) shows the same circuit behaving correctly after simulation while ordered reconstruction and the corresponding simulated circuit is shown in Figure 4.8. The detail descriptions of the procedures mentioned in this section are given in the next section.

4.3 Simulation techniques and algorithms

This section describes the simulation technique for simulating circuits and its associated techniques to provide efficient simulation for a category of circuits, a partitioning technique in a gate level circuit, an advanced case based analysis etc. An analysis is also been presented to show how the simulation performs better in terms of generation of unnecessary events with respect to the standard event driven simulation.

4.3.1 Simulation of circuits

As mentioned in the previous section (section 4.1), the tool uses levelized simulation technique for simulating combinational circuits, standard event driven technique for simulation of sequential circuits which do not conform to the Huffman model and a combination of levelized and event driven simulation technique for sequential circuits conforming Huffman model for better simulation performance. Before simulating a sequential circuit, first, the circuit is partitioned into the combinational



```verilog
module JKffWithNAND(j,k,clk,q,qb);
        input j,k,clk;
        output q,qb;
        wire q,qb,w1,w2,w3,w4;

        and a1 (w4,k,clk);
        and a2 (w3,j,clk);
        nand n1 (w1,q,w4);
        nand n2 (qb,q,w1);
        nand n3 (w2,qb,w3);
        nand n4 (q,qb,w2);
endmodule
```

Figure 4.6: (a) JK flip flop before simulation with all unknown value (maroon dashed line). (b) The same unresolved JK flip flop after standard event driven simulation (blue solid line and black dotted line denotes logic value 1 and 0 respectively). (c) The flip flop after case analysis giving definite outputs. (d) Verilog code for the JK flip flop.

and the sequential part, only if the circuit has single clock domain and loop detected through depth-first-search (DFS) back edges. Then, using those partitions, the circuit is verified if it conforms the Huffman model or not, if it does, then the whole simulation is scheduled for better performance. And if it does not, then standard event driven simulation is done. After simulation, if some nets of the loops are left unknown within the structural storage elements of the circuit, the simulation technique performs case analysis of those unknown nets to determine whether or not a definite output value is assumed irrespective of the cases considered. If the case analysis fails to resolve the initial unknown signals in a structural flip flop, there is an option to resolve the unknown flip flops through the simulation while reconstructing the circuit in an ordered manner. As the non-convergence of a structural storage element may be caused due to the legitimate unknown uncertainty which resolves eventually as the circuit operates, the feature of simulating that non-converged circuit while ordered reconstruction is made optional for the user. The whole simulation is outlined in Algorithm 1. Major computational tasks are done on internal netlist representing the original circuit and finally the result is reflected on the original circuit i.e., the display netlist. The logic gates and the other basic components of the circuit are represented by the nodes of the netlist. The internal netlist representation of a display netlist is denoted here with IntlNetlist and the display netlist is denoted by the DispNetlist. A detailed description is given in the subsequent paragraphs.

Partitioning a gate level circuit

The partitioning is based on a heuristic that all gate level flip flop circuits contain single length latch, shown in Figure 4.11(b), at their output level. As single length latch exhibits data retaining behavior in gate level memory elements and the rest of the elements contribute to synchronize the operations with the clock, the set of single length latches are identified in a circuit, before partitioning. Single length latches are identified from the set of back edges obtained from standard breath-first-search (BFS) starting from the clock. If a circuit does not contain any such latch, partition fails. The procedure for identification of latches is described in Algorithm 2.

After listing out latches, the shortest paths are computed from clock to the latches shown with

ALGORITHM 1: MainSimulation(DispNetlist)

```
Input  : A display netlist with all nets having unknown values
Output: Simulated DispNetlist
1  $backEdges_{DFS} \leftarrow DFS(DispNetlist, I)$; /* list DFS back edges (if any) on the netlist, starting from primary inputs I */
2  if backEdges_{DFS} $\neq \emptyset$ and single clock domain then
3      $backEdges_{BFS} \leftarrow BFS(DispNetlist, clock)$;   /* list back edges from BFS starting from clock on the netlist */
4      $latches \leftarrow GetLatches(backEdges_{BFS})$;
5      if $latches \neq \emptyset$ then
6          $\langle seqSet, combSet, latchBoundarySet\rangle \leftarrow Partition(DispNetlist, latches)$; /* partition the netlist into sequential set and combinational set */
7          if PossibleRaceAround(DispNetlist, latchBoundarySet) then
8              Stop simulation if opted by user;
9          end
10         $\langle inputBounds, outputBounds, outsFromSeq\rangle \leftarrow GetBoundaries(DispNetlist, latches)$;
11         if IsHuffman(DispNetlist, seqSet, combSet, outputBounds, outsFromSeq) then
12             $\langle RPI, RSI\rangle \leftarrow ClassifyCombSet(DispNetlist, seqSet, combSet, outsFromSeq)$;
13             ScheduledSimulation(RPI, seqSet, RSI);   /* Huffman model */
14         else
15             EventDriven(DispNetlist, I);   /* non-Huffman model */
16         end
17     else
18         EventDriven(DispNetlist, I);   /* loop exists but no latch */
19     end
20 else if $backEdges_{DFS} \neq \emptyset$ then
21     EventDriven(DispNetlist, I);   /* loop exists but no clock */
22 else
23     $topoOrder \leftarrow TopologicalSort(DispNetlist, I)$;
24     process all the nodes of topoOrder list in topological order;
25 end
26 if $latches \neq \emptyset$ and $\exists$ unknown edge $(u, v) \in backEdges_{DFS}$ then
27     $FFset \leftarrow IdentifyFlipflops(DispNetlist, latchBoundarySet)$; /* compute the set of structural flip flops */
28     CaseAnalysis(IntlNetlist, FFset);   /* resolve flip flops through case analysis */
29     ReconstructAndResolve(IntlNetlist, FFset); /* if user opts, resolve non-converging flip flops through simulation while ordered reconstruction */
30 end
```

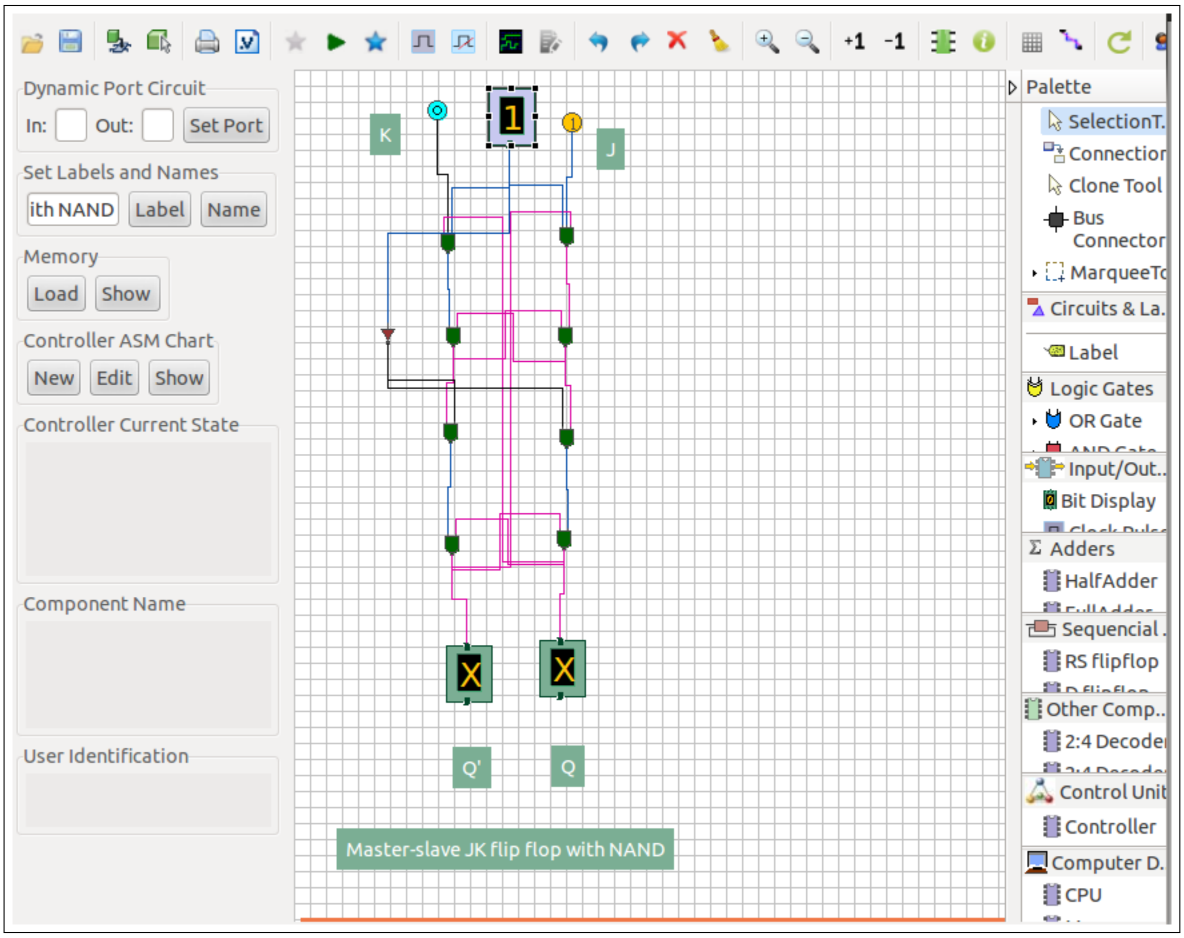


Figure 4.7: Master-slave JK flip flop (NAND implementation) after case analysis failed.

the thick solid brown colored line in Figure 4.12 which presents a schematic diagram corresponding to a circuit shown in the Figure 4.13. All the elements residing in the shortest paths computed from clock to all the latches along with the behavioral sequential components are called the sequential domain of the circuit to be simulated. The rest of the circuit is considered as combinational part. The elements on the shortest path from clock to the latches include the *clock* for the latch connected to the *clock* (shown with the dashed boundary in Figure 4.14(a)), and include the NOT gate which is connected to the *clock* for the latch connected to the $\overline{clock}$ (shown with the dotted boundary in the same figure). The partition algorithm also computes the latch boundaries excluding the *clock* for the latches connected to the *clock* and both *clock* and the NOT gate connected to the *clock* for latches connected $\overline{clock}$. Figure 4.14(b) shows two latch boundaries identified so far, one with dashed and another with dotted boundary and the Figure 4.15 shows four latch boundaries with dashed boundaries identified within a shifting circuit. Algorithm 3 describes the partition algorithm.

Detection of possible race around condition

After a successful partition, the existence of possible race around condition is checked through checking the existence of non-master-slave flip flop (when non-master-slave storage elements are

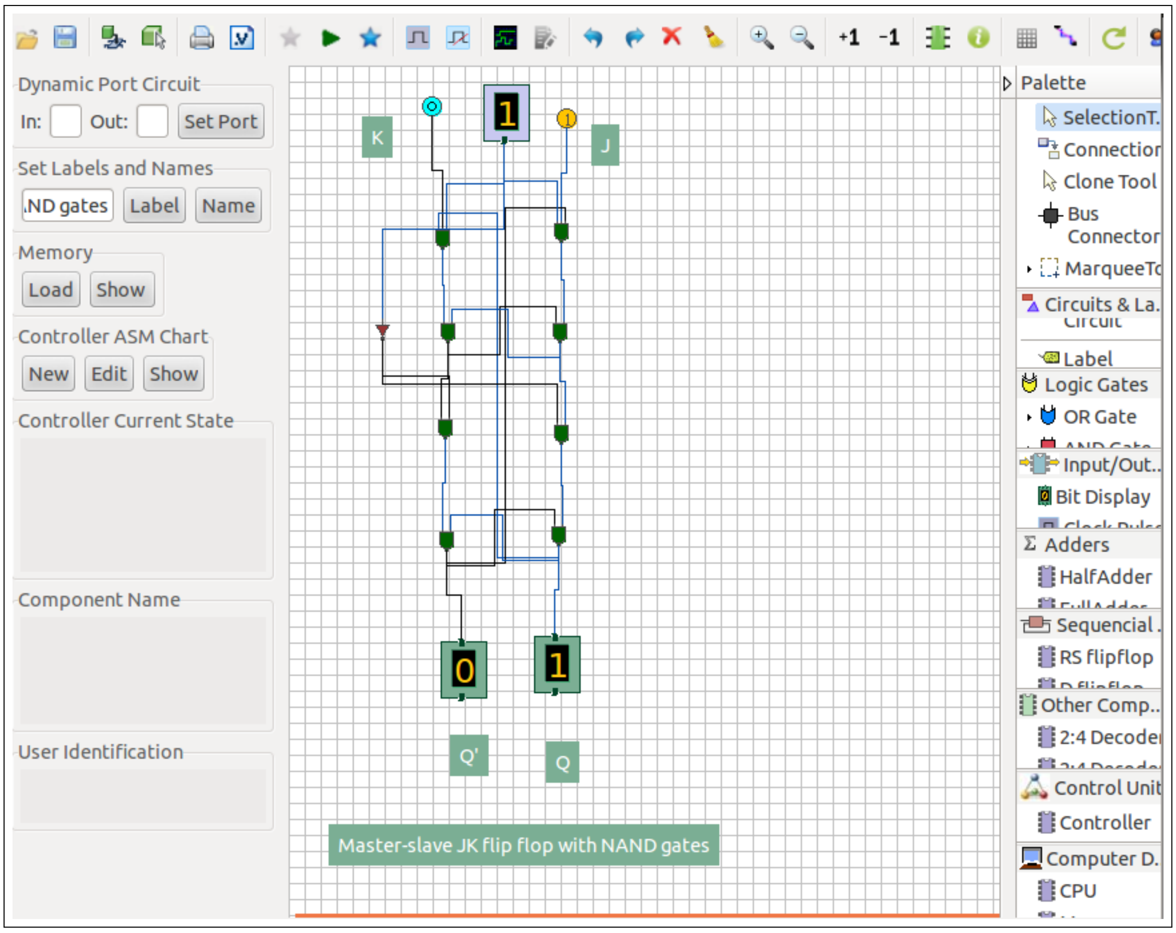


Figure 4.8: Master-slave JK flip flop (NAND implementation) after ordered simulation with gate-by-gate reconstruction gives proper result.

used). This feature is not related to the simulation as such. This has been developed for the learning purpose. If the possibility of race around is detected then the user is notified to be more careful during the design. The master-slave pattern is detected from the set of latch boundaries. Among the set of latch boundaries, they are classified into two sets, one containing all the latch boundaries connected to *clock* and another contains all the latch boundaries connected to the $\overline{clock}$. Then, if for each latch boundary connected to the *clock* has outgoing connection to another latch boundary connected to the $\overline{clock}$, mark them as master-slave and they will not be checked further. After this, if any latch boundary is left un-marked, there is a possibility of race around condition in the circuit and the user is notified so that user can stop the simulation to modify the circuit and re-simulate the circuit. Figure 4.16 shows an example where latch boundaries connected to *clock* are shown with the boundary of dashed line, latch boundaries connected to the $\overline{clock}$ are bounded with the dotted line and the boundaries with dot-dashed line indicate the identified master-slave pattern. The procedure is described in Algorithm 5.

ALGORITHM 2: GetLatches(eSet)

```
Input  : eSet is set of edges where $(u, v)$ is a directed edge from u to v
Output: Set of single length latches, comprise nodes whose levels in BFS,
        $level_{\mathrm{BFS}}(node)$, is greater than 0
1  $maxLatchLevel \leftarrow 0$;
2  foreach $((u, v), (v, u)) \in$ eSet do
3      $maxLatchLevel \leftarrow max(maxLatchLevel, level_{BFS}(u), level_{BFS}(v))$;
4      $latches \leftarrow latches \bigcup \{(u, v)\}$;   /* listing one edge per latch */
5  end
6  if $maxLatchLevel > 0$ then
7      return latches;      /* if latch exists, return latch set */
8  else
9      return $\emptyset$;
10 end
```

ALGORITHM 3: Partition(Netlist, latches)

```
Input  : netlist and latches is set of single length latches
Output: Partition the netlist into combinational part and sequential part, also
        compute set of latch boundaries comprising only nodes of structural flip
        flops
1  foreach $(u, v) \in latches$ do
2      $LB \leftarrow \{GetPath(Netlist, clock, u), GetPath(Netlist, clock, v)\}$;
3      $seqSet \leftarrow seqSet \bigcup LB$ ; /* compute set of all structural flip
       flops */
4      $latchBoundarySet \leftarrow latchBoundarySet \bigcup \{LB - \{clock, \overline{clock}\}\}$;
5  end
6  $seqSet \leftarrow seqSet \bigcup \{all\ behavioral\ sequential\ components\}$;
7  $combSet \leftarrow Netlist \setminus seqSet$;       /* generate combinational part */
```

ALGORITHM 4: GetPath(Netlist, n_1, n_2)

```
Input  : netlist and two nodes $n_1$ and $n_2$ within the netlist
Output: Returns the shortest path from $n_1$ to $n_2$
1  $\Pi \leftarrow BFS(n_1, n_2)$; /* compute predecessor list, $\Pi$, by doing BFS
   on the netlist, start with $n_1$ and stop at $n_2$ */
2  back track the predecessor list from $n_2$ to $n_1$ and store each node in list path;
3  Return path;
```

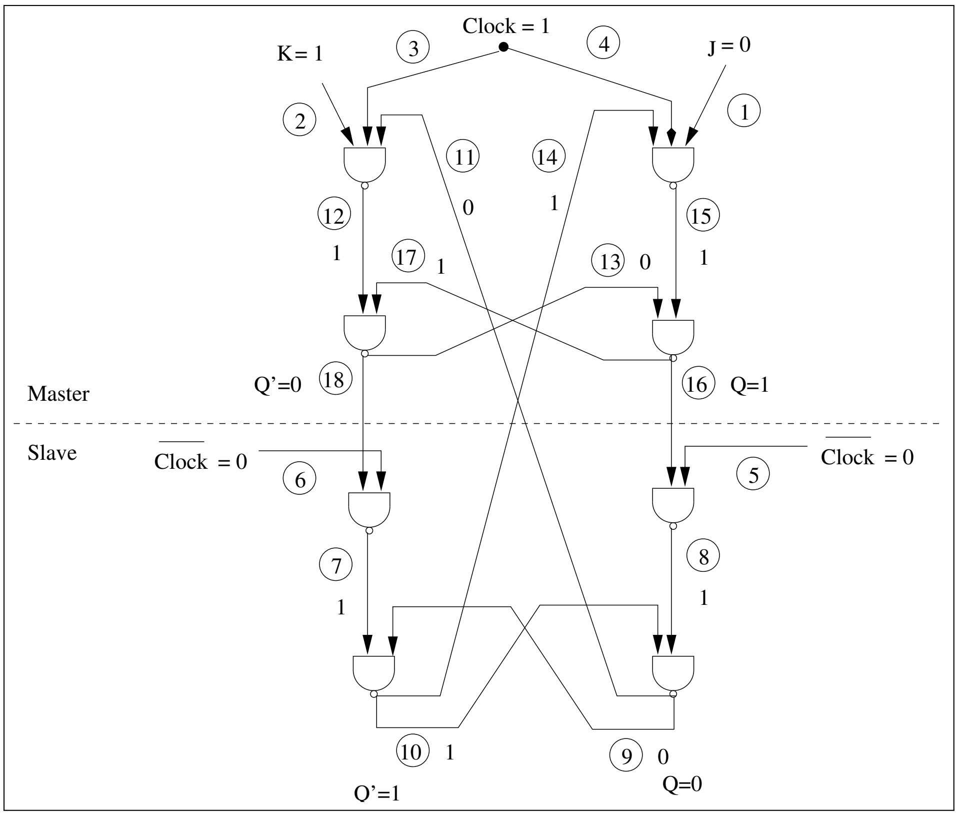


Figure 4.9: An order of building master-slave flip flop which causes malfunctioning circuit where master gives wrong output (the order of connection is shown with the numbers in circle and the value (1 or 0) of a wire is shown).

Identifying Huffman model

The circuit is verified whether it conforms the Huffman model after accepting the notification from the user to proceed if there was possible race around notification. In a Huffman model, depicted in Figure 4.11(a), all the secondary outputs of combinational part becomes the inputs of the memory elements and all the secondary inputs of combinational part emerge from the outputs of the flip flips. This property need not be checked for the behavioral components as they are encapsulated modules having predefined input-output terminals. However, this property has to be verified for structural flip flops as in the structural flip flop, output emerges from some particular elements. Therefore, two boundaries have been defined, input and output bounds, for the structural flip flops within the sequential domain to validate these properties. Set of the elements of the structural flip flops which directly receive the *clock* or $\overline{\textit{clock}}$ pulse is listed as input boundary. The list of elements of the structural flip flops, comprises all the latches, is called output boundary (shown in Figure 4.12). Before detecting whether a circuit is a Huffman model, the input-output boundaries of the structural flip flops are identified. All the secondary inputs which emerge from both structural flip flops and behavioral sequential components are also determined. Algorithm 6 gives procedure for identifying them.

After getting the input-output boundaries of the structural flip flops, secondary inputs and sec-

ALGORITHM 5: PossibleRaceAround(Netlist, latchBoundarySet)

Input : netlist and latchBoundarySet is set of latch boundaries
Output: Detect possible race around condition by detecting master-slave pattern among the latch boundaries

```
1  $LBfromClk \leftarrow$ set of all latch boundaries connected to clock;
2  $LBfrom\overline{Clk} \leftarrow$ set of all latch boundaries connected to $\overline{clock}$;
3  foreach $x \in LBfromClk$ do
4      if $\exists\, y \in LBfrom\overline{Clk}$, s.t. $\exists$ edge $(u, v)$ in Netlist, $u \in x, v \in y$ then
5          $LBfromClk \leftarrow LBfromClk - x$;
6          $LBfrom\overline{Clk} \leftarrow LBfrom\overline{Clk} - y$;
7      end
8  end
9  if $LBfromClk \neq \emptyset$ and $LBfrom\overline{Clk} \neq \emptyset$ then
10     Possible race around exists;
11 end
```

ALGORITHM 6: GetBoundaries(Netlist, latches)

Input : netlist, and latches is set of single length latches
Output: Input and output boundaries of sequential set where *inputBounds* is a set of nodes having direct input from *clock* or $\overline{clock}$ and *outputBounds* is a set of nodes creating latches
Notations: *outsFromBhv*: all outgoing edges from sequential set to combinational set emerging from behavioral sequential components

```
1  $inputBounds \leftarrow inputBounds \bigcup \{children(clock), children(\overline{clock})\}$;
2  foreach $(u, v) \in latches$ do
3      $outputBounds \leftarrow outputBounds \bigcup \{u, v\}$;
4  end
5  $outsFromBhv \leftarrow outputBounds \bigcup outsFromBhv$;
```

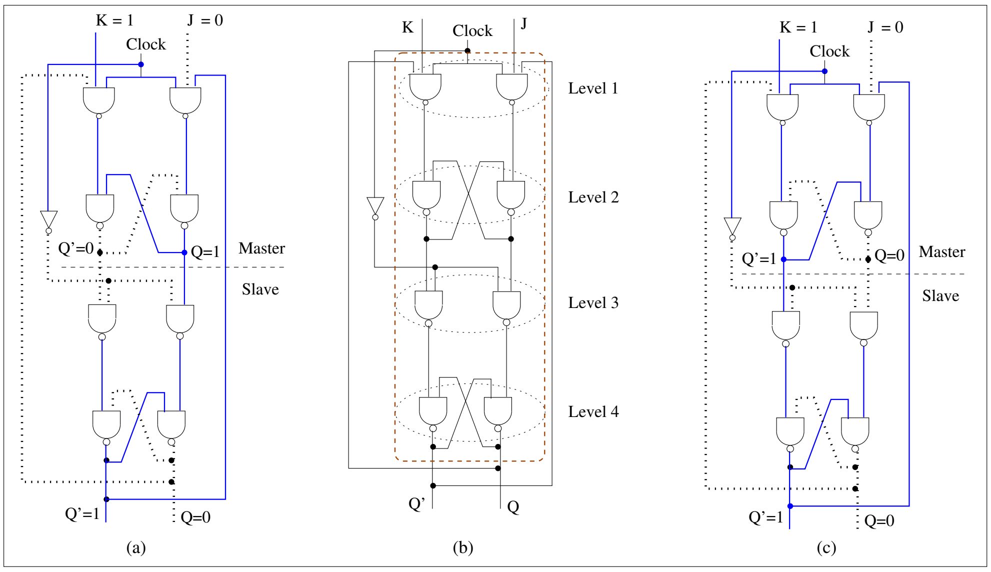


Figure 4.10: (a) Malfunctioning master-slave JK flip flop (blue solid line and black dotted line denotes logic value 1 and 0 respectively). (b) Breath-first levels of the elements of the master-slave JK flip flop (shown within a brown dashed boundary) for simulation while ordered reconstruction. (c) The same flip flop functions properly after simulation while ordered reconstruction.

inputBounds: set of nodes having direct input from *clock* or $\overline{clock}$
outputBounds: set of nodes creating latches of memory elements
outsFromSeq: all secondary inputs from sequential to combinational domain
latchBoundarySet: set of latch boundaries comprising only nodes of structural flip flops
RPI: list of nodes reachable from primary inputs in topological order
RSI: list of nodes reachable from secondary inputs

Table 4.1: Notations used in Algorithm 1

ondary outputs, associated with the structural flip flops, are identified from/to the structural flip flops respectively. The secondary outputs are determined by considering the incoming edges from the combinational cloud to the structural flip flops and checked whether they incident only on the elements belonging to the input boundary. Thereafter, secondary inputs are determined by considering the outgoing connections from the structural flip flops to the combinational cloud and verified whether the secondary inputs coming out from the structural flip flops, actually emerge from the elements belonging to the output boundary. Thereafter, the combinational cloud, identified by the partition algorithm, is checked whether it is purely combinational in nature or it contains any random loop. Loops are identified by the back edges given by depth-first-search (DFS). However, if DFS is executed with primary inputs as starting point within the combinational set, there may exist some elements which will be left unvisited by DFS, as they are only reachable from the elements of output boundaries of the structural flip flops, shown in Figure 4.12 or from the behavioral sequential components. Therefore, DFS is done starting from both the set of primary inputs and set of all

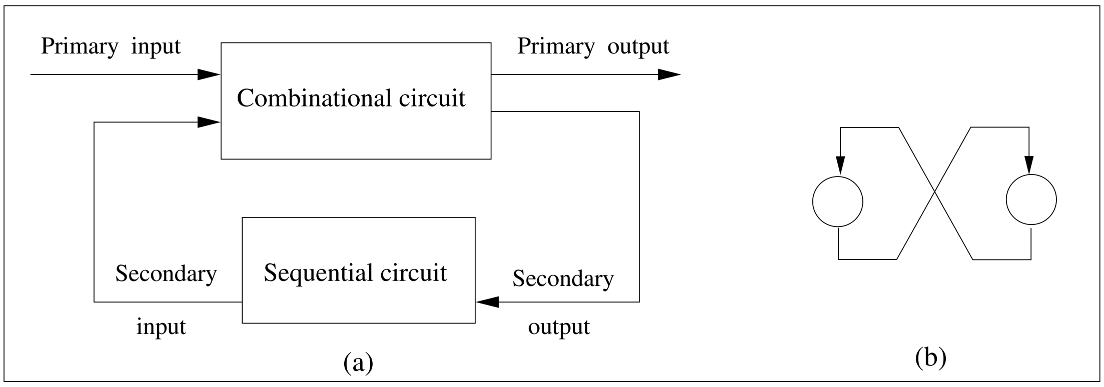


Figure 4.11: (a) Huffman model. (b) Single length latch, circle denotes gate.

secondary inputs emerging from the sequential domain. If all the conditions are satisfied, the circuit is conformed as Huffman model. Algorithm 7 describes the Huffman detection procedure.

Scheduling of simulation

When a sequential circuit is recognized as Huffman model, it is simulated in a scheduled manner as mentioned earlier in order to achieve better performance. Combinational part is simulated in levelized manner and sequential part in event driven approach. All the elements of combinational domain are required to be listed in topological order in order to simulate them in levelized manner which ensures that an element will be processed only after all of its predecessors are processed. In order to levelize combinational part, elements are classified in terms of reachability. Reachability of an element is an important concern here, because levelization procedure must visit every element to give it a level. In combinational set, there exist a set of elements which are reachable from primary inputs is called RPI, some elements which are reachable from the outputs of structural flip flips and behavioral sequential components or all the secondary inputs are named as RSI and some elements which are reachable from both the primary and secondary inputs, will reside in both RPI and RSI set, shown in Figure 4.12. This classification will further help in simulation scheduling. Because, for the simulation of combinational part in levelized order, the elements which are reachable only from primary inputs can be processed without depending on the results of flip flips, however, the elements of RSI set which are reachable from the elements of the sequential domain have to wait until those elements give their outputs. Therefore, the simulation is scheduled in such a way that the aforesaid behavior is preserved. Two topologically ordered list corresponding to RPI and RSI set are generated and the procedure is outlined in Algorithm 8.

RPI list is generated by using standard topological sort algorithm starting from primary inputs. Similarly RSI list is generated but the topological sort algorithm starts from all the elements of the sequential domain from which secondary inputs emerges. One example shows these lists in Figure 4.12. At first, the RPI list is simulated in levelized order generating the secondary outputs then simulate the sequential set in event driven manner. After the flip flips reached their fixed point, the RSI list is simulated in topological order. If race around condition occurs i.e., if any flip flip does

ALGORITHM 7: IsHuffman(Netlist, outputBounds, combSet)

```
Input  : netlist, outputBounds is output boundaries of sequential set and combSet is
         combinational set
Output: Returns true if Netlist conforms the Huffman model
1  if $\forall v \in$ inputBounds then
2      $b_1 \leftarrow$ true, where $(u, v) \in SO$;  /* checking whether all secondary
       outputs incident within input boundaries of sequential
       set, SO is set of incoming edges of sequential set
       from combinational set */
3  if $\forall u \in$ outputBounds then
4      $b_2 \leftarrow$ true, where $(u, v) \in SI$;  /* checking whether all secondary
       inputs emerge from output boundaries of sequential
       set, SI is set of incoming edges of combinational set
       from sequential set */
5  end
6  $p \leftarrow$ DFS(combSet, I);     /* checking loops within combinational
   nodes */
                                   /* reachable from primary inputs, I */
7  $s \leftarrow$ DFS(combSet, allOutEdgesFromSeq);     /* checking loops within
   combinational nodes reachable from outgoing edges of
   sequential set */
8  if $(b_1 \,\&\, b_2 = true)$ and $p = \emptyset$ and $s = \emptyset$ then
9      isHuffman $\leftarrow$ true;
10 end
11 Return isHuffman;
```

ALGORITHM 8: ClassifyCombinationalSet(Netlist, outputBounds, combSet)

```
Input  : netlist, outputBounds is output boundaries of sequential set and combSet is
         combinational set
Output: Classified combinational nodes into two sets, RPI, nodes reachable from
         primary inputs and RSI, nodes reachable from all the nodes of the sequential
         domain from which secondary inputs emerges
1  RPI $\leftarrow$ TopologicalSort(combSet, I);  /* standard topological sort */
2  RSI $\leftarrow$ TopologicalSort(combSet, outsFromSeq);
```

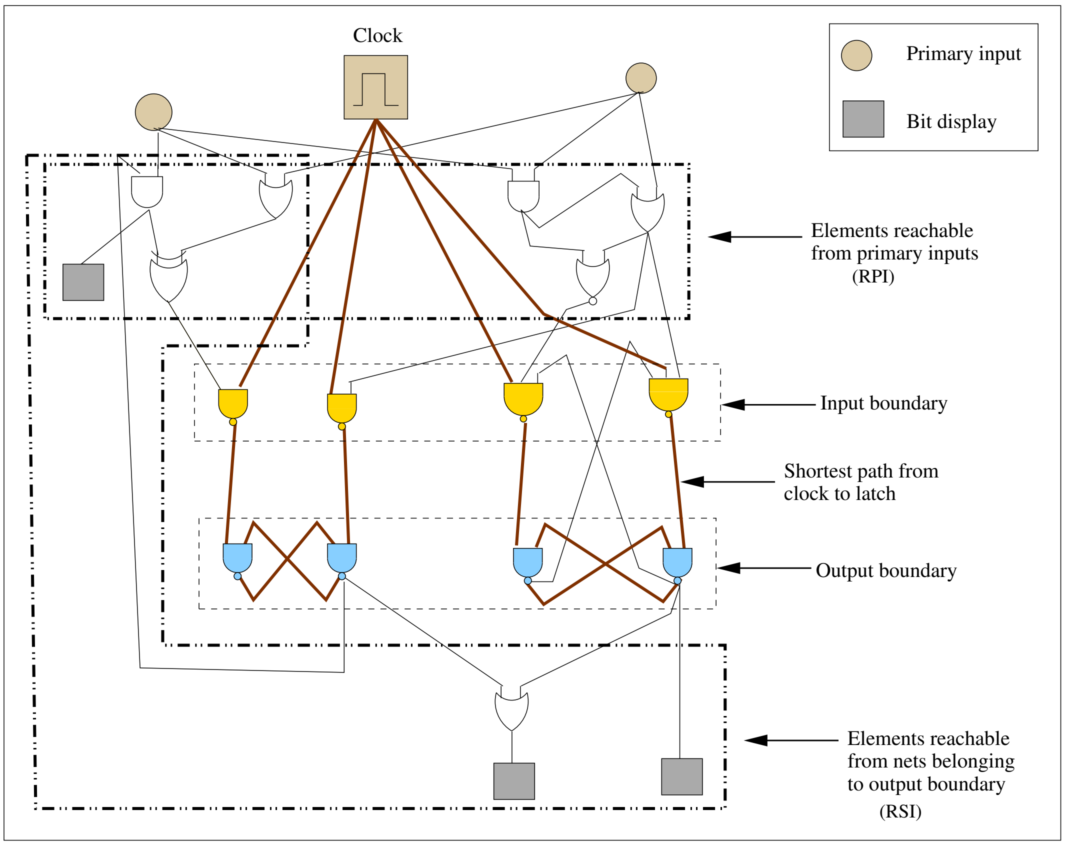

Clock
Primary input
Bit display
Elements reachable from primary inputs (RPI)
Input boundary
Shortest path from clock to latch
Output boundary
Elements reachable from nets belonging to output boundary (RSI)

Figure 4.12: Different sets of elements are shown in a simple schematic circuit conforming to the Huffman model which are identified during simulation.

not reach to its fixed point then the simulation does not proceed to the RSI list of the combinational set. Algorithm 9 describes the scheduled simulation algorithm and Algorithm 10 briefly outlines the regular event driven simulation technique. In event driven simulation all the behavioral sequential components are updated only once.

4.3.2 Resolving indeterminate values of structural memory elements

As mentioned earlier, for some gate level flip flop circuits, due to the unknown initial value of all the logic signals, standard event driven simulation can not produce definite output for the given inputs, even though these circuits should produce definite outputs (for the given inputs). To handle that situation, the simulation technique performs case analysis of unknown signal values of the nets which are part of the loops in gate level memory elements to determine whether or not a definite output value is assumed irrespective of the cases considered.

Identification of structural flip flops

This case analysis is required only for structural flip flops. Therefore, before case analysis, structural flip flops are identified from the latch boundaries identified so far. each pair of latch boundaries

ALGORITHM 9: ScheduledSimulation(RPI, seqSet, RSI)

```
Input  : RPI is set of nodes reachable from primary inputs, seqSet is sequential set,
         RSI is set of nodes reachable from all secondary inputs. These inputs
         comprise the display netlist
Output: Simulated display netlist, more efficiently than event driven approach
1  process all the nodes of RPI list in topological order;
2  EventDrivn(seqSet);          /* event driven simulation within
   sequential part */
3  foreach $n \in$ outputBounds do
4      if n does not reaches its fixed point after k number of updations then
5          raceAround $\leftarrow$ true;       /* race around condition occurs
           within sequential set, halt simulation */
6      end
7  end
8  if raceAround = false then
9      process all the nodes of RSI list in topological order;
10 end
```

ALGORITHM 10: EventDriven(X, I)

```
Input  : X is a netlist or a part of netlist to be simulated and I is a set of nodes to
         start simulation with
Output: Simulated input netlist in event driven manner
1  Enqueue(Q , I );  /* insert start nodes into event queue, Q */
2  while $Q \neq \emptyset$ do
3      n $\leftarrow$ Dequeue(Q);
4      if n is a behavioral sequential component then
5          update(n) only for once;
6      else
7          update(n);                    /* compute output of n */
8      end
9      if visitCount(n) <= maxCount and n has not reached to its fixedpoint then
10         put all the output nodes of n within X in Q;
11     end
12 end
```

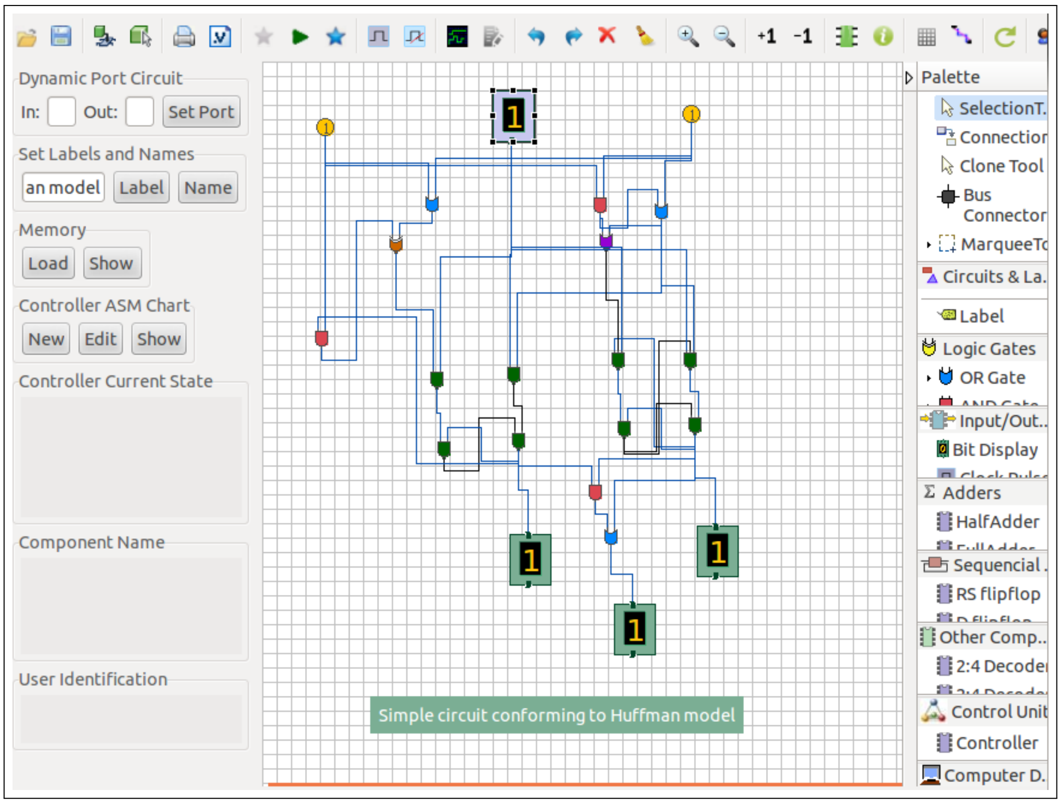


Figure 4.13: Actual circuit corresponding to the schematic presented in Figure 4.12.

are checked if they have direct connections between them from one another. If they have such connections, the two latch boundaries are merged and inserted into the set of flip flops, they will not be considered for further checking of direct connections between two latch boundaries. Even if any pair of latch boundaries does not have direct connections from one another, they are also inserted in the set of flip flops. Then the set of flip flops is refined by checking if any two sets have common elements between them, they are merged and the set of flip flops finally contains the identified flip flop boundaries. One example is shown in the Figure 4.14(c), where master-slave JK flip flop is identified by the procedure (shown with the boundary of dot-dashed line). Also in the Figure 4.15, the latch boundaries identified so far, essentially becomes the flip flop boundaries. Algorithm 11 describes the procedure for identification of structural flip flops.

Case based analysis procedure

After identification of structural flip flops, case analysis is done on each unresolved structural flip flop in the internal netlist representing the partially simulated display netlist. After the case analysis is done for all unresolved structural flip flops, the definite values of the converged flip flops are set in the display netlist. The case analysis procedure is depicted in Algorithm 12. Case analysis for a structural flip flop, described in Algorithm 13, starts with computing the total number of loops within the flip flop through the DFS back edges. If there are n back edges, there will be 2^n cases to be checked which is essentially the all possible combinations of 0 and 1 for n variables. A stack is

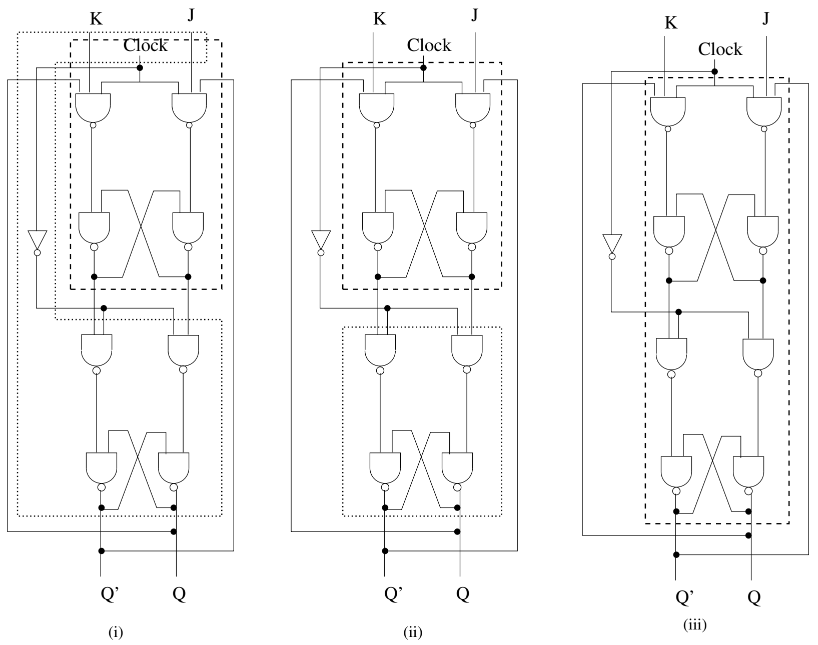


Figure 4.14: (a) Elements within the boundary of the latch connected to *clock* including *clock* element are shown within a dashed boundary and the dotted boundary shows the gates within the boundary of the latch connected to $\overline{clock}$ including the *clock* and the NOT gate connected to the *clock*. (b) Latch boundaries excluding the *clock* and the NOT gate connected to the *clock*. (c) Master-slave JK flip flop (shown within dot-dashed line) is identified from the latch boundaries.

used for checking all possible cases. Initially, all the back edges are unknown, then one of the back edges is set to 0, pushed into the stack and the flip flop is simulated in event driven manner form the target element of that back edge. After simulation if any back edge is left unknown, again the same procedure is applied until all unknown back edges are resolved and this procedure is described in the Algorithm 14. Then the output values of all the elements of that flip flop are stored in a vector which will be checked with the output of those elements in other cases which is described in Algorithm 15. This procedure essentially stores the output values of the elements of the flip flop in the first case which essentially means that all the unknown loops are resolved with 0, Then it proceeds to check the output value of the elements stored in the vector with the output values of the elements in another case. If the stack is not empty, the stack is popped. If the popped back edge was pushed with its value set to 0, it is set to 1 then again pushed into the stack and the back edges which are not in the stack are set to unknown value. Then again simulation procedure described in Algorithm 14, is performed whose execution signifies the completion of another case. After completion of a case, current value of the elements of the flip flop is matched with the stored value of the corresponding elements in the vector whose procedure is described in in Algorithm 15. If any one of the outputs does not match, case analysis fails to converge and if all outputs match, case analysis proceeds to check other cases. This procedure is described for the popped back edge from the stack with value set to 0 during pushing in the stack, however, if the back edge was pushed with value set to 1, the

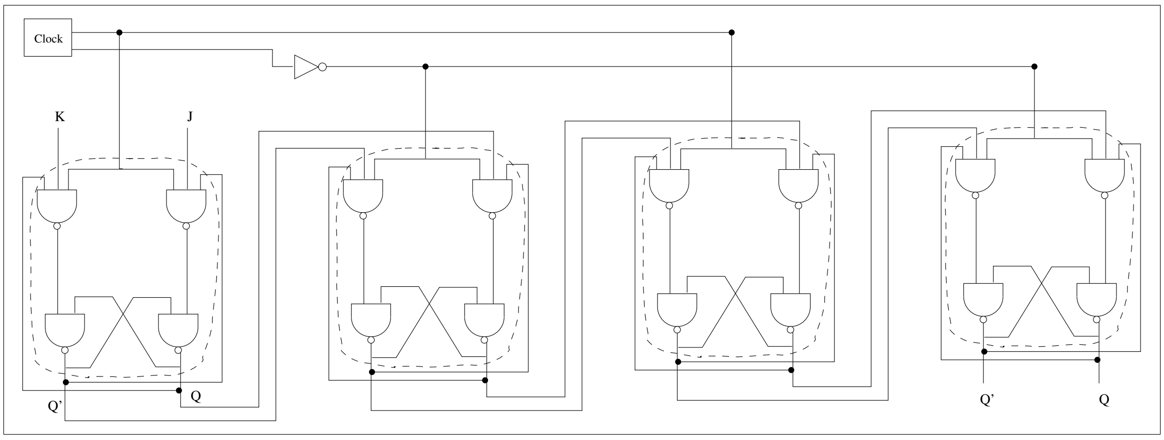


Figure 4.15: Four latch boundaries with dotted boundaries identified within a shifting circuit.

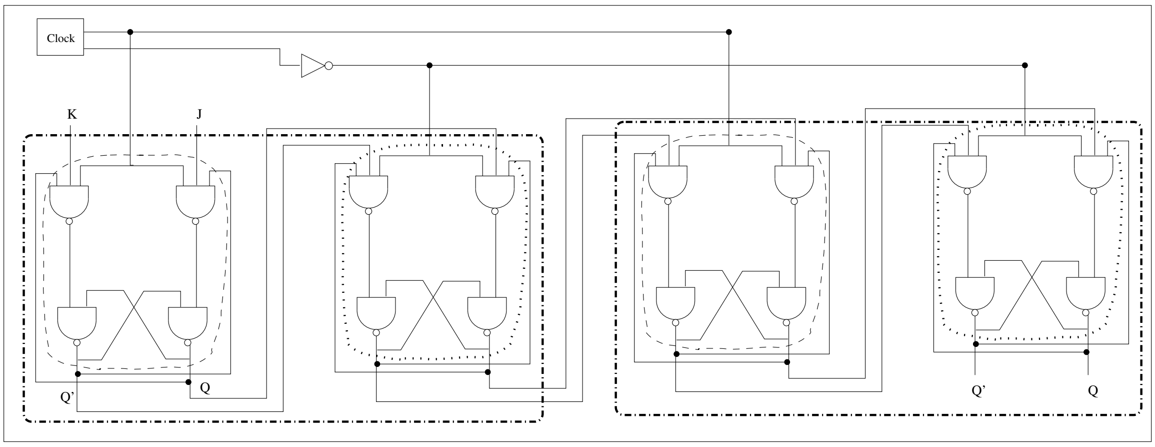


Figure 4.16: Identifying master-slave pattern within latch boundaries (latch boundaries connected to *clock* are shown with dashed boundary, latch boundaries connected to the $\overline{clock}$ are sown with dotted boundary and the dot-dashed boundaries indicate the identified master-slave pattern).

back edge is skipped and simply another back edge is popped from the stack. This whole recursive process is described in the Algorithm 16.

Handling failure of case analysis

As discussed in the previous section (section 4.1), the non-convergence of case analysis which occurs due to the failure of resolving the initial unknown signal values through the normal operation of the circuit, is resolved through the simulation of the circuit while reconstructing it in an ascending breath-first order from clock (mentioned in the section 4.1) which is briefly outlined in the Algorithm 17 and shown in Figure 4.10.

ALGORITHM 11: IdentifyFlipflops(Netlist, latchBoundarySet)

```
Input  : netlist, latchBoundarySet is set of latch boundaries
Output: Returns set of flip flops, FFset
1  foreach pair $(L_1, L_2)$, where $L_1, L_2 \in latchBoundarySet$ and $L_1 \neq L_2$ do
2      if $\exists$ edge $(u_1, v_1)$ and $(u_2, v_2)$ in Netlist, s.t. $u_1, v_2 \in L_1$ and $u_2, v_1 \in L_2$ then
3          $L_1 \leftarrow L_1 \bigcup L_2$;                    /* merge $L_2$ with $L_1$ */
4          $FFset \leftarrow FFset \bigcup \{L_1\}$;
5          $latchBoundarySet \leftarrow latchBoundarySet - \{L_1, L_2\}$;
6      else
7          $FFset \leftarrow FFset \bigcup \{L_1, L_2\}$;
8      end
9  end
10 foreach pair $(F_1, F_2)$, where $F_1, F_2 \in FFset$ and $F_1 \neq F_2$ do
11     if $F_1 \bigcap F_2 \neq \emptyset$ then
12         $F_1 \leftarrow F_1 \bigcup F_2$;   /* merge $F_2$ with $F_1$ as both have common
           nodes */
13         $FFset \leftarrow FFset - \{F_2\}$;
14     end
15 end
16 Return $FFset$;
```

ALGORITHM 12: CaseAnalysis(Netlist, FFset)

```
Input  : netlist of partially simulated display netlist and FFset is set of structural flip
         flops
Output: Does case analysis of structural flip flops and completes the simulation of
         partially simulated display netlist by setting the values of converged flip
         flops
1  foreach $FF \in FFset$ do
2      $backEdges_{FF} \leftarrow DFS(FF, clock/\overline{clock})$; /* list back edges from DFS
       starting from clock or $\overline{clock}$ depending upon whether the
       flip flop is connected to clock or $\overline{clock}$, within the
       flip flop */
3      if $\exists$ edge $\in backEdges_{FF}$ s.t. value(edge) = unknown then
4          AllCaseAnalysisFF(Netlist, FF, $backEdges_{FF}$);    /* do all possible
           case analysis for the unresolved flip flop */
5      end
6  end
7  Set the value of nodes of all the converged flip flops in the original display netlist;
```

ALGORITHM 13: AllCaseAnalysisFF(Netlist, FF, $\text{backEdges}_{\text{FF}}$)

Input : netlist, FF is the flip flop for analyzing all possible cases, $\text{backEdges}_{\text{FF}}$ is set of DFS back edges of the flip flop

Output: Does all possible case analysis for the flip flop and if it converges, stores the values of the nodes of the flip flop

```
1  edge = ∅;
2  stack = ∅;
3  CaseSimulate(Netlist, stack, FF, $backEdges_{FF}$, edge);
4  CheckAllResolved(Netlist, FF);
5  PopStack(Netlist, stack, FF, $backEdges_{FF}$);
6  if if case analysis for FF converges then
7      Store the values of nodes of the converged flip flop, FF;
8  end
```

ALGORITHM 14: CaseSimulate(Netlist, stack, FF, $\text{backEdges}_{\text{FF}}$, edge)

Input : netlist, stack for performing all possible case analysis, FF is the flip flop for analyzing all possible cases, $\text{backEdges}_{\text{FF}}$ is set of DFS back edges of the flip flop, edge is a connection between two nodes

Output: Simulate Netlist during case analysis

```
1  if edge ≠ ∅ then
2      EventDriven(FF, target(e));  /* simulate the flip flop in event
       driven manner starting from the incident node of the
       edge */
3  end
4  foreach e ∈ $backEdges_{FF}$ do
5      if value(e) = unknown then
6          value(e) ← 0; /* set the unknown back edge value to 0 */
7          Push e into the stack;
8          EventDriven(FF, target(e))
9      end
10 end
```

ALGORITHM 15: CheckAllResolved(Netlist, FF)

Input : netlist, FF is the flip flop for analyzing all possible cases

Output: Checks the values of nodes of the flip flop of one case with another, only if the values matches in all the cases, case analysis converges

```
1  if $valVector = \emptyset$ then
2      foreach $node \in FF$ do
3          $value_{valVector}(node) \leftarrow value_{Netlist}(node)$;    /* store the value of
           the node obtained from the netlist, in a vector if
           it is empty */
4      end
5  else
6      foreach $node \in FF$ do
7          if $value_{Netlist}(node) \neq value_{valVector}(node)$ then
8              Return false;   /* if the vector is not empty, and the
               value of node from the netlist does not match
               with the stored value in the vector, case
               analysis for the flip flop does not converge */
9          end
10     end
11 end
12 Return true;
```

4.4 Comparison between event driven and the combined simulation

In event driven simulation of a circuit, any change in the input to a component causes it to be simulated. Many of these simulations are redundant as the outputs determined by those are subsumed by later simulations. For a circuit with n gates, $O(n^2)$ simulation events may be generated. However, if that circuit conforms to the Huffman model, we shall show that our combined simulation technique offers a great improvement by simulating the circuit in only $O(n)$ time. Let x, y are the number of gates in combinational and sequential domain respectively and $n = x + y$. In case of the combined approach of circuit simulation, the combinational domain, where there are no cyclic dependencies between components, is simulated by topologically sorting the components based on their signal interdependencies; that way each component needs to be simulated only once enabling the simulation of this part to be done in $O(x)$ time. We next show that an event driven simulation of the sequential domain (comprising storage elements) of such a circuit only generates O(y) events.

Let there be p number of storage elements (flip flops) in the sequential domain of the circuit and the number of gates in each flip flop be: $z_1, z_2, \ldots, z_p$. These values typically range from two to eight. For a circuit conforming to the Huffman model, the clocking mechanism ensures that signals are not directly propagated between these storage elements. As a result, each flipflop is simulated

ALGORITHM 16: PopStack(Netlist, stack, FF, $\text{backEdges}_{\text{FF}}$)

Input : netlist, FF is the flip flop for analyzing all possible cases, $\text{backEdges}_{\text{FF}}$ is set of DFS back edges of the flip flop

Output: Pops a stack element which is a back edge and based on the value set to the back edge before pushing into the stack, performs different operations to generate all possible cases

```
1  if stack $\neq \emptyset$ then
2      edge $\leftarrow$ pop the stack;
3      if edge was set to 0 when pushed to stack then
4          value(edge) $\leftarrow$ 1;                /* value of edge set to 1 */
5          Push edge into the stack;
6          foreach $e \in backEdges_{FF}$ and $e \notin stack$ do
7              value(e) $\leftarrow$ unknown;
8          end
9          CaseSimulate(Netlist, stack, FF, $backEdges_{FF}$, edge);
10         if CheckAllResolved(Netlist, FF) then
11             if stack $\neq \emptyset$ then
12                 PopStack(Netlist, stack, FF, $\text{backEdges}_{FF}$);
13             end
14         end
15     else
           /* edge was set to 1 when pushed to stack         */
16         if stack $\neq \emptyset$ then
17             PopStack(Netlist, stack, FF, $backEdges_{FF}$);
18         end
19     end
20 end
```

ALGORITHM 17: ReconstructAndResolve(Netlist, FFset)

Input : netlist of partially simulated display netlist and FFset is set of structural flip flops

Output: Resolve non-converging flip flops through simulating them while reconstructing them in an ordered manner and finally sets the values of resolved nodes in the display netlist

```
1 foreach FF $\in$ FFset do
2     if FF did not converge after case analysis then
3         ReconstructFFinOrder(Netlist, FF);
4     end
5 end
6 Set the value of nodes of all the resolved flip flops in the display netlist;
```

ALGORITHM 18: ReconstructFFinOrder(Netlist, FF)

Input : netlist of partially simulated display netlist and FF is the flip flop to be resolved by simulation using ordered reconstruction

Output: Simulate the flip flop while reconstructing it node by node in breath-first manner (starting from *clock*) and stores the definite value of the nodes of the flip flop

```
1  Construct empty netlist NetlistNew;
2  Label all the nodes of FF through breath-first-search starting from clock;
3  foreach node ∈ FF in ascending order do
4      Read input-output connections of node from Netlist;
5      Construct node and its input-output connections from/to other nodes in NetlistNew;

6      foreach $node_{NetlistNew}$ ∈ NetlistNew and having unconnected input terminals do
7          Set 1 to all unconnected input terminals;
8      end
9      EventDriven(NetlistNew, node);
10 end
11 Store the value of nodes of the resolved flip flop;
```

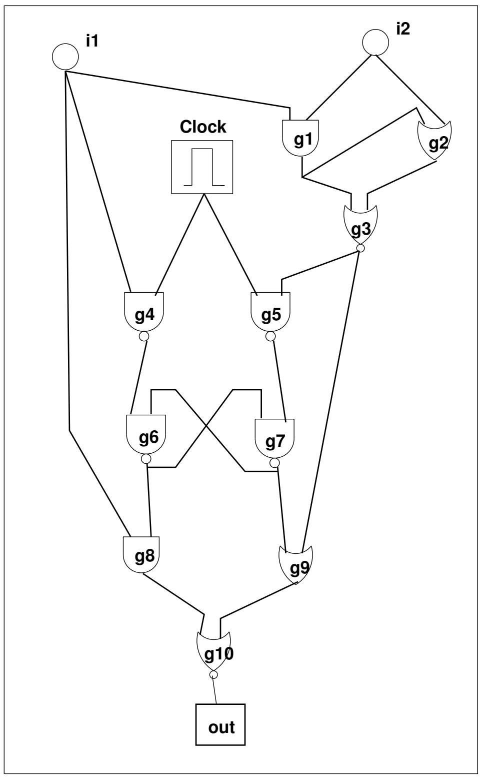


Figure 4.17: A sample schematic circuit conforming to Huffman model

independently in the event driven framework. As none of the elements from the combinational domain is simulated till the simulation of sequential domain is finished, the gates triggered while simulating the sequential domain of the circuit will always be limited to the sequential domain only. So the number of events generated will be $O(\sum_{i=1}^{i=p} z_i^2)$ and z_i is the number of gates in the i^{th} flip flop. As $z_i \in [2, 8]$, number of events raised for sequential domain can be written as $O(p)$, which is $O(y)$. Therefore, the number of events generated in combined simulation approach for the whole circuit conforming to the Huffman model is O(x) + O(y) which is O(n).

The circuit in Figure 4.17 conforms to the Huffman model as it can be clearly partitioned into combinational and sequential domains. In this circuit, after partitioning, gates g_1, g_2, g_3 (*RPI* set as defined in this chapter), g_8, g_9, g_{10} (RSI set) are in combinational domain where the rest of the gates are in sequential domain. Executing event driven simulation on the whole circuit, on input i_1 and i_2, the gates g_1, g_2, g_4 and g_8 will be simulated and this will in turn raise events for gates g_2, g_3, g_6, g_{10}. After simulating these gates, the successors of these gates will be triggered for simulation and this process will be followed for each gate of the circuit. By this method, gates in the combinational domain of the circuit may get simulated unnecessarily multiple times to get the output of the circuit. In case of combined simulation technique, the gates in *RPI* set can be simulated in the order $g_1 \rightarrow g_2 \rightarrow g_3$ and the order in the *RSI* set can be as $\{g_8, g_9\} \rightarrow g_{10}$. To illustrate it further, only gate g_1 will be simulated after input i_1 and i_2 has given, as all the inputs are available only for gate g_1 and the simulation will continue for the gates g_2, g_3 after the inputs of these gates become available. The sequential part of the circuit will be simulated by event driven simulation technique. The gates g_8, g_9 and g_{10} will not be executed until the simulation of the sequential part has completed. After the values of g_6 and g_7 has been determined, the gates g_8, g_9 and g_{10} will be executed in levelized order. This method will ensure that every gate in the combinational domain of a circuit will be executed for once and thus reducing the number of events raised.

4.5 Implementation and results

Implementation of COLDVL tool The COLDVL tool has been implemented on the Java platform.The development of the tool is based on Eclipse GEF framework [5]. This GEF framework was first provided by Eclipse as plug-in application. Then Architexa [4] with the collaboration of IBM and Eclipse developer team has extended the framework as a stand alone Java SWT application suitable for web based application. Dropping out their web based aspect of the application, we have further extended the framework to develop the tool by adding more components, extending logic value and implementing our simulation approach along with other significant features mentioned earlier. Figure 2.13 shows the way COLDVL interacts with the clients. One of the prime goals of COLDVL tool is to handle very large user base at a time which has been achieved here by its minimal dependency on the server. All the experiments of COLDVL and the matters pertaining to those experiments reside in the server side whereas the CPU intensive task of simulation to carry out the experiment is performed on the client side. The COLDVL tool is available in two forms, it can either be launched or downloaded. If the tool is launched, every time the user gets the updated version of the tool automatically but if the tool is downloaded and used then user does not get the updated version automatically. The user has to re-download the tool to get the latest version. This approach abolishes the network latency between the user and the tool and thus speeding the tool response, as it does not rely on any interaction with the server [17].

Results: The tool is being used to conduct semester laboratory course at IIT Kharagpur since 2012. Table 4.2 shows some under graduate and post graduate student assignments accomplished at the IIT Kharagpur using the tool and run on a machine with the following configuration. Intel® Core™ i3 CPU 550 @ 3.20GHz×4 processor, 32-bit ubuntu 12.04 LTS operating system and 3.6 GiB memory. The assignment statements are given in the subsection 3.5.4. The execution time (CPU time measured in milliseconds) for the initial cycles is much more due the overhead of several library loading, however, once they are leaded, later execution cycles takes much less time which is shown in a range within the execution time varies. The circuit is built and simulated using several features provided in the COLDVL tool which are explained using some case studies in the previous chapter.

4.6 Conclusion

In this chapter the back end part of the COLDVL tool is covered which comprises of the efficient simulation technique for circuits conforming to the Huffman model. A partitioning technique is developed for a gate level circuit in order to check whether it conforms to the Huffman model or not. For some signal values which remain indeterminate after regular round of simulation in structural memory elements, a case based analysis is developed so that no discrepancy arises for a novice student between real laboratory experimentation and simulated experimentation in a virtual environment. The algorithms for the techniques along with the case studies and examples are presented in this chapter. Implementation of the COLDVL tool along with the results indicating execution time for some student assignments are presented.

Experiment Name	**Components**		**CPU Time range (ms)**		**Remarks**
	Logic Gates	**Behavioral Comp.**	**Initial Cycles**	**Avg.**	
16-bit block carry look ahead adder (BCLA)	298	basic gates	650-800	2-5	pure combinational circuit
Single instruction CPU	806	basic gates, controller, working memory to load binary program	970-1200	4-8	sequential circuit having data path, controller and working memory. Inst: SBN
Regular CPU with 4 instructions	719	basic gates, controller, working memory to load binary program	900-1100	4-10	sequential circuit having data path, controller and working memory. Inst: ADD, jz (jump if zero), LOAD, NEG (negate)
8-bit shift and add multiplier	570	basic gates, controller	900-1200	5-10	sequential circuit having data path and controller
Radix-4 8-bit Booth's multiplier	660	basic gates, controller	920-1300	5-12	sequential circuit having data path and controller

Table 4.2: Execution time of some student assignments performed in COLDVL tool.

Chapter 5

Checking student designs for correctness

5.1 Introduction

Evaluation of student designs is an important issue in addition with the laboratory experimentation. Manual and simulated evaluations of the student designs against a given reference design provided by the course instructor are some what restricted. Therefore an equivalence checking method would be desirable to check the correctness of the student designs against a given reference design. This work proposes an equivalence checking method which is performed on the formal models representing the student design as well as the reference design specifications. The design behaviors are modeled as finite state machines with data paths (FSMD). The equivalence is checked between the two FSMDs. In an FSMD, a *path* is a finite sequence of states where the first and the last state may be same. A computation in a FSMD, can be viewed as a computation along some concatenated paths in that FSMD. *Path cover* of an FSMD is a finite set of paths and any computation of that FSMD can be looked upon as a concatenation of paths belonging to the set. An FSMD, M_0 is said to be contained in another FSMD M_1 if M_0 has a finite path cover and and for each path in that path cover, M_1 has an equivalent path. The equivalence between two FSMDs are established by proving that one is contained in another. While finding the equivalent path for a path during equivalence checking, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Moreover, as the actual student designs are the digital circuits where data transformations are performed over finite data paths, the equivalence checker must be able to handle the finite precision data in bit level.

This chapter describes the proposed bit-level equivalence checking method which has been developed for the purpose of automatic checking of student designs against the given assignment. This chapter also includes the challenges addressed along with the algorithms for this method and an illustrated example. A brief description of the FSMD model and the basic equivalence checking method is also included. Finally, the results of the current implementation of the bit-level equivalence checking method is given.

5.2 Finite State Machine with Datapaths

A brief formal description of the finite state machines with data paths (FSMD) model is given in this section. A detailed description of FSMD models can be found in [33]. The FSMD model [20], used in this work to model the initial behaviour and the transformed behaviour, is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, f : Q \times 2^S \rightarrow Q, h : Q \times 2^S \rightarrow U \rangle$, where Q is the finite set of control states, q_0 is the reset (initial) state, I is the set of input variables which are never changed in course of a computation of the model, V is the set of storage variables, O is the set of output variables, f is the state transition function, h is the update function of the output and the storage variables, U represents a set of storage and output assignments and S represents a set of relations over arithmetic expressions and Boolean literals.

Path in FSMD

A *path* α in an FSMD model is a finite sequence of states where at most the first and the last states may be non-distinct and any two consecutive states in the sequence are in f. The initial (start) and the final states of a path α are denoted as α^s and α^f, respectively.

Condition of execution of a path

The *condition of execution R_α of the path α* is a logical expression over the variables in V and the inputs I such that R_α is satisfied by the (initial) data state of the path iff the path α is traversed.

Data transformation of a path

The *data transformation r_α of a path α* over V is the tuple $\langle s_\alpha, \theta_\alpha \rangle$; the first member s_α is an ordered tuple $\langle e_i \rangle$ of algebraic expressions over the variables in V and the inputs in I such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state of the path; the second member θ_α, which represents the output list along the path α, is typically of the form $[OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$. More specifically, for every expression e output to port P along the path α, there is a member $OUT(P, e)$ in the list appearing in the order in which the outputs occur in α. The condition of execution and the data transformation of a path are computed using the method of symbolic execution.

Computation in FSMD

A *computation* of an FSMD is a finite walk from the reset state back to itself without having any intermediary occurrence of the reset state.

Definition 1 (Equivalence between two computations). *Two computations μ_1 and μ_2 having the characteristic formulae τ_{μ_1} and τ_{μ_2}, respectively, are said to be equivalent if $R_{\mu_1} = R_{\mu_2}$ and $r_{\mu_1} = r_{\mu_2}$.*

The computational equivalence of two computations μ_1 and μ_2 is denoted as $\mu_1 \simeq \mu_2$. The computational equivalence of two paths p_1 and p_2 can be defined in a similar manner and is denoted as $p_1 \simeq p_2$. Equivalence checking of paths, therefore, consists in establishing the computational equivalence of the respective conditions of execution and the respective data transformations.

An FSMD may consist of an infinite number of computations. However, any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of M such that, for $1 \leq i < k$, α_i terminates in the initial state of the path α_{i+1}, the path α_1 emanates from the reset state q_0 and the path α_k terminates in q_0 of M; α_i's may not all be distinct. Hence, we have the following definition.

Definition 2 (Path cover of an FSMD). *A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.*

In order to obtain a path cover for an FSMD each loop is to be cut in at least one cut-point. The set of all paths from a cut-point to another cut-point without having any intermediary cut-point is a path cover of the FSMD [19]. In this work, a path cover is obtained by setting the reset state and the branching states (i.e., states with more than one outward transition) of the FSMD as cut-points.

Equivalence of two FSMDs

It is to be noted that if two behaviors are to be the same, then their outputs must match. So, when some variable is output, its counterpart in the other FSMD must attain the same value. Then the following can be stated. Let the reference behavior be represented by the FSMD $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and the user behavior be represented by the FSMD $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$. An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that $\mu_0 \simeq \mu_1$. Two FSMDs M_0 and M_1 are said to be computationally equivalent, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.

An FSMD may contain an infinite number of computations. So, it is not feasible to enumerate all possible computations in one FSMD and find their equivalent computations in the other FSMD. To overcome this problem, the following theorem is deduced.

Theorem 1. *For any two FSMDs M_0 and M_1, $M_0 \sqsubseteq M_1$, if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1^0 = \{p_{10}^0,\ p_{11}^0, \ldots,\ p_{1l}^0\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}^0$, $0 \leq i \leq l$.*

Another important notion is as follows.

Definition 3 (Corresponding states). *Let $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$ be two FSMDs having identical input and output sets, I and O, respectively, and $q_{0,i}, q_{0,k} \in Q_0$ and $q_{1,j}, q_{1,l} \in Q_1$.*

1) *The respective reset states $q_{0,0}$ and $q_{1,0}$ are corresponding states.*

2) *If $q_{0,i} \in Q_0$ and $q_{1,j} \in Q_1$ are corresponding states and there exist $q_{0,k} \in Q_0$ and $q_{1,l} \in Q_1$ such that, for some path β from $q_{0,i}$ to $q_{0,k}$ in M_0, there exists a path α from $q_{1,j}$ to $q_{1,l}$ in M_1 such that $\beta \simeq \alpha$, then $q_{0,k}$ and $q_{1,l}$ are corresponding states.*

The method of basic equivalence checking for integer domain data
The basic equivalence checking method is based on the above discussion which consists of the following steps.

1. Construct the set P_0 of paths of M_0 so that P_0 covers M_0. Let $P_0 = \{p_{00}, p_{01}, \cdots, p_{0k}\}$.
2. Show that $\forall p_{0i} \in P_0, there\ exists\ a\ path\ p_{1j}$ of M_1 such that $p_{0i} \simeq p_{1j}$.
3. Repeat steps 1 and 2 with M_0 and M_1 interchanged.

It is important to note that the choice of cut-points is non-unique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cut-points in itself will have the corresponding set of equivalent paths for the other FSMD.

In the second step above a crucial operation is to determine the equivalence of two paths which in turn relies on determining the equivalence of arithmetic expressions. In [33], integer arithmetic expressions have been considered and their equivalence is determined using the normalization mechanism [46], [45]. As a result the equivalence checking method of [33] is biased towards programs that deal with integer arithmetic. While the integer domain offers an elegant representation for a wide variety of problems, in the next section we shall examine issues in determining equivalence over bit-level data and thereby adopt representation and normalization scheme that is better suited to handle bit-level data.

5.3 Issues in determining equivalence over bit-level data

The designs extracted from the experiments conducted through the COLDVL virtual laboratory, have finite data path with various bit-level operations such as arithmetic operations, logical operations, shift operations and some other operations such as concatenation, extraction performed in it. [33], [10] can not handle the finite data path and the aforesaid bit-level operations due to its restriction to the infinite domain. This section briefly describes the way this challenge along with some other issues addressed in the newly developed bit-level equivalence checking method along with an example (shift and add multiplier) presented in Figure 5.1 (reference design) and Figure 5.2 (user or student design) to show the some example scenario where those issues occurs. The two figures shows two versions of shift and add multiplier and some of the operations in them are shown in different colors to indicate different issues in determining the equivalence between the designs.

Handling bit-level operations
Consider two variants of the shift and add multiplication scheme shown in the Figures 5.1 and 5.2. In those designs, we can see some arithmetic operations such as $A[3:0] + M[3:0]$, $Q[3:0] \leftarrow Q[3:0] \gg 1$, $count[3:0] - 1$, $A[3:0] + 0$. These essentially implies that operands of the operations are finite precision data, they have finite but multiple bits and bits are ordered from least significant bits to most significant bits.

On the other hand, other than those designs shown in the Figures 5.1 and 5.2, suppose, between the another two designs, one design contain $A[3:0] - M[3:0])$ and another design contain $A[3:0] + \overline{M[3:0]} + 1)$. These two operations are structurally dissimilar differing in number of operators, type of operators, number and type of operands. however, manually applying the theory of logic design, it is clear that the two operations are the same.

Consider another formula which is as follows.

$$(x - y > 0) \Leftrightarrow (x > y)$$

This is a valid formula when interpreted over integer domain, however this equivalence does not hold good over bit-vectors due to the existence of possible overflow in the subtraction operation. For example, x[3:0]=111, y[3:0]=010, the unsigned & 2's complement value of 111 is 7 & -1 respectively, where as the unsigned & 2's complement value of 010 is 2 for both of them.

From the above scenarios, it is clear that in order to develop an equivalence checker which can establish equivalence between two designs having such operations, the equivalence checker needs to be able to reason over bit-levels. it should also take account for the possible overflow and underflow situations associated with the operations. Structurally dissimilar operations are also required to be represented through a normalized from so that all the expressions are transformed into a standard structure and the equivalence between two non-identical expressions can be established.

To achieve those, the proposed bit-level equivalence checking method models the data path as finite-precision bit-vectors and bit-wise operations performed on them. This enables the equivalence checking method to handle arithmetic, logical, bit-wise and shift operation in bit-level including concatenation operation. The theory of the bit-vectors are equational theory over finite non-empty strings over {0,1} and having a known fixed width. The formulas of this theory are boolean combinations of the equalities over bit-vector expressions. For an n bit bit-vector, the leftmost bit is the least significant bit (LSB) and the rightmost bit is called most significant bit (MSB). The bits are indexed as LSB being 0 and MSB being $n-1$ with an ordering from left to right. Binary decision diagrams (BDD) [15] are used to represent and reason over the bit-vectors efficiently. Following is the bit-vector logic syntax supported by our bit-level equivalence checking method.

$$\begin{array}{ll}
formula: & term \mid atom \mid \sim formula \\
term: & identifier \mid constant \mid term\,[\,constant : constant\,] \mid term\,[\,constant\,] \mid \\
 & \sim term \mid term\; op\; term \mid atom \,?\, term : term \\
atom: & term\; rel\; term \mid term \\
op: & + \mid - \mid \ll \mid \gg \mid \lll \mid \ggg \mid \circ \mid AND \mid OR \mid \\
 & NAND \mid NOR \mid XOR \mid XNOR \\
rel: & == \mid < \mid > \mid \leq \mid \geq
\end{array}$$

Where, $\ll, \gg, \lll, \ggg, \circ$ denotes bit-wise logical left shift, logical right shift, arithmetic left shift, arithmetic right shift and concatenation operation respectively.

Results spanning over variable boundary

In integer domain, result of an operation is stored in a single variable. However, in bit-level operation, often the result is itself build of multiple bits which in turn may be assigned to the bits of other variable with choice. Therefore, the result of a bit-level arithmetic can be distributed in multiple variables.

For example, referring to the Figure 5.1, consider the operation $\{C, A[3:0]\} \leftarrow A[3:0]+M[3:0])$. This operation essentially signifies that the result of the addition operation will be distributed over the single variable C and four bits of variable A. It is to be noted that as the size of the variable C is one, its size is not shown in a range which is shown for A as $[3:0]$ denoting $A[0]$ as the least significant bit (LSB) and the $A[3]$ as the most significant bit (MSB).

Consider another operation $prod[7:0] \leftarrow \{A[3:0], A\_X[3:0]\}$ whose interpretation is as follows. the four data bits residing on A and A_X are assigned to a single variable $prod$ with size equal to the sum of A and A_X in an ordered manner. The right most bit is the LSB, therefore, $A\_X[0]$ will be assigned to the LSB of the result i.e. $prod[0]$. Note that, when a result is distributed over multiple variables, we denote those variable as composed variable.

In such cases, it is necessary for the equivalence checker to track the results, the distribution of the result bits, the boundary of the result and the order of the result bits. The ultimate aim of the equivalence checking is to check whether the corresponding variables (which are present in both the designs) of the two designs holds same output or not. The equivalence checking task performed over integer domain involves only single variable correspondence i.e the correspondence is always between the single variables as the output of any operation is assigned to a single variable. However, due to the variable composition, as mentioned above, the correspondence may range from single variable to multiple variable. The bit-level equivalence checker handles these composed variables along with the issues of tracking result associated with them. In order to identify the variable composition, currently a naming convention is adopted. According to the naming convention, the variable $A\_X[3:0]$ signifies that it is actually the variable X having composition with A.

Evolving size of result

The size of the result may not be fixed. It may be evolving through iterations. Consider the two operations from the Figure 5.1 which is as follows. (*i*) $\{C, A[3:0], A\_X[3:0]\} \leftarrow \{C, A[3:0], A\_X[3:0]\} \gg 1$ (*ii*) $prod[7:0] \leftarrow \{A[3:0], A\_X[3:0]\}$ Where $prod[7:0]$ holds the output of the multiplication which is assigned values from the composed variables $\{A[3:0], A\_X[3:0]\}$ (by naming convention the composition is identified). From the first operation it can be seen that a right shift operation is performed and it is also very important to note that the shifting is performed on these composed variable (along with another variable) iteratively. In this case, the result boundary is growing by one bit with each iteration through shifting operations. There are other computer arithmetic algorithms where data evolves through iteration and in each iteration one or more than one result bits are appended to the boundary of the existing result causing the current boundary of the result to be expanded. For example, in shift and add multiplication and Booth's multiplication, result evolves by one bit per iteration. However, in radix-4 Booths multiplication, the result evolves by two new result bits to the existing result boundary per iteration.

Through several operations the result can grow such as different shifting operations, multiplied by two, etc. Through the above example, it is stated that how a result can evolve through iteration within a design. As the equivalence checking deals with the two design, cases may occur that in both the designs the data is evolving through similar operations such as shifting operations, multiplied by two, etc. or data may grow through dissimilar operations such as mismatched shifting operations, one design consisting of shifting operations and another design having multiplied by two. However, the current scope of the work can handle evolving data through similar or dissimilar shifting operations. dealing equivalence between the operations such as left shifting by one bit and the multiplied by two is beyond the scope of this work. Figures 5.1 and 5.2 consists of both similar and dissimilar shifting operations. The dissimilar operation is shown in magenta color and it depicts that the shift operation $\{C, A[3:0], A\_X[3:0]\} \leftarrow \{C, A[3:0], A\_X[3:0]\} \gg 1$ occurs only in the reference design, not in the user design. However, the operations $Q[3:0] \leftarrow Q[3:0] \gg 1$ and $A\_Q[3:0] \leftarrow A\_Q[3:0] \gg 1$ may appear dissimilar due to their variable name, it may be recalled that due to the naming convention adopted in this work, A_Q is actually the variable Q denoting its association with the variable A. Therefore the two shift operations are similar.

To handle this phenomena of evolving data through iterations, the equivalence checker need to track the boundary of the results and try to obtain the shift characteristic through which data builds. This is accomplished with the help of some observations generated in terms of predicates at different phases of the execution of the bit-level equivalence checking method including the design observations. New observations are deduced from the existing predicates using some predefined rules. Following are some examples of simple predicates used in this work and the other predicates are given in the later sections.

Predicate-1: (SHR(v):n) [Logical right shift of variable v by n bits]

Predicate-2: (SHRA(v):n) [Arithmetic right shift of variable v by n bits]

Predicate-3: (SHL(v):n) [Logical left shift of variable v by n bits]

Again consider the operation, $prod[7:0] \leftarrow \{A[3:0], A\_X[3:0]\}$ in the Figure 5.1. It can be seen that eight bits are assigned to the variable $prod$ from the variables $\{A[3:0]$ and A_X. It has already been noted from the design that $\{A[3:0]$ and A_X are composed variables in which result bit is inserted iteratively implying that after full iteration the all the bits of the composed variables will hold valid data bits before that some bits of the variables. However, with the symbolic execution approach adopted in this bit-level equivalence checking method which does not unroll a loop, can not infer anything about the data bits after a full iteration. To handle the situation, the equivalence checker utilize the shifting characteristic deduced earlier for each path belonging to a loop. From the shifting characteristic of each path of a loop, it is checked whether a loop invariant shifting characteristic can be obtained using the set of predicates (observations) and predefined rules. Once the equivalence checker successfully identifies the loop invariant data shift characteristic, using that characteristic and the loop count (it is assumed that the loop count is finite and known), the equivalence checker then infer the data range after the full iteration through interpolation.

5.4 Methodologies and algorithms

The proposed bit-level equivalence checking method is a path cover based method which decomposes a design into several path segments and can handle bit-level arithmetic, logical, bit-wise, shift operations, conditional branching. For some cases where the result evolves through iteration, the method also does some special analysis to capture the pattern through which the result evolves. A successful extraction of pattern may cause the equivalence checker to perform some other extra analysis to infer the data range after completion of an iteration. The equivalence checker can also determine the equivalence in the presence of operations scheduled in different clock cycles as the related operations may not happen in lock step. The following subsections contain the detail method along with the algorithms for the bit-level equivalence checking method.

5.4.1 Overall equivalence checking method

The overall equivalence checking method is outlined in the Algorithm 19 along with the illustration of terms used in the algorithm in the Table 5.3. Initially the loops are found in the two input designs i.e FSMDs. Then the cut-points are introduced through marking the reset states (*start* and *end* states) and the states having branches in the input FSMDs to compute the path cover. The path cover of an FSMD consists of the paths between the cut-points of that FSMD. The equivalence checker starts from the *start* cut-point in breath-first-search (BFS) manner in both the designs. In this work it is assumed that between the two input FSMDs, a one-to-one correspondence between cut-points exists. Following is the notion of corresponding cut-points.

Definition 4 (Corresponding cut-points). *Let* $fsmd^R = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ *and* $fsmd^U = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$ *be two FSMDs having identical input and output sets,* I *and* O*, respectively, and* $q_{0,i}, q_{0,k} \in Q_0$ *and* $q_{1,j}, q_{1,l} \in Q_1$*. Where* $fsmd^R$ *is the FSMD corresponding to the reference design and* $fsmd^U$ *is the FSMD corresponding to the user design.*

1) *The respective reset states* $q_{0,0}$ *and* $q_{1,0}$ *are corresponding cut-points.*

2) *If* $q_{0,i} \in Q_0$ *and* $q_{1,j} \in Q_1$ *are corresponding cut-points and there exist* $q_{0,k} \in Q_0$ *and* $q_{1,l} \in Q_1$ *such that,* $q_{0,k}$ *and* $q_{1,l}$ *are cut-points and for some path* β *from* $q_{0,i}$ *to* $q_{0,k}$ *in* $fsmd^R$*, there exists a path* α *from* $q_{1,j}$ *to* $q_{1,l}$ *in* $fsmd^U$ *such that* $\beta \simeq \alpha$*, then* $q_{0,k}$ *and* $q_{1,l}$ *are corresponding cut-points.*

In Figure 5.1 and Figure 5.2, the corresponding cut-points are (CP_1, CP_1'), (CP_2, CP_2'), (CP_3, CP_3'), (CP_4, CP_4').

For every path between any two pair of cut-points in one FSMD, the equivalence checker will try to find its equivalent path between the corresponding cut-point (Definition 4) pairs in the another FSMD. In an FSMD, a path contains a condition of execution and a set of data transformation operations and the execution of a path depends upon satisfaction of its condition of execution. Two paths are only considered for checking equivalence if they have same condition of execution. Two designs are declared as equivalent if every path between the cut-points in a design has its equivalent path between the corresponding cut-points in the other design i.e. every path of the path cover of a design must has its equivalent path in the other design.

Path computation

To check equivalence between two paths, the equivalence checker does path computation for both the paths. Before executing a path, all the input variables which are not assigned with specific values are assigned with symbols. The common input variables (which are common in both the FSMDs) are assigned with same symbols.For the path computation we need the following notions.

Definition 5 (Symbolic value vector (SVV)). *Let fsmd* $= \langle Q, q, I, V, O, f, h\rangle$ *is an FSMD and the finite sized variables are denoted by this ordered form* $\langle v_1[m_1 : n1], v_2[m_2 : n_2] \dots, v_k[m_k : n_k]\rangle$ *where* m_k *is the most significant and* n_k *is the least significant bit of the variable* v_k *and* $k \in V$ *then* $SVV = \langle v_1 : \langle b[m_1] \dots, b[n_1]\rangle, v_2 : \langle b[m_2] \dots, b[n_2]\rangle \dots v_k : \langle b[m_k] \dots b[n_k]\rangle\rangle$ *where* $b[i]$ *is the* i^{th} *bit of the variable* $v \in v_1, v_2, \dots, v_k$ *and* $n_j \leqslant i \leqslant m_j$, $1 \leqslant j \leqslant k$. *Every cut-point will hold an* SVV.

Definition 6 (Start symbolic value vector (StartSVV)). *Let fsmd* $= \langle Q, q, I, V, O, f, h\rangle$ *is an FSMD and* $\{p_1, p_2, \dots, p_k\}$ *is the set of paths from the cut-point* CP_x *to* CP_y *where* CP_x *and* $CP_y \in Q$, *then for* $p_i \in \{p_1, p_2, \dots, p_k\}$, $StartSVV_{p_i} \leftarrow SVV_{CP_x}$. $StartSVV_{p_i}$ *is the start value vector used for the symbolic execution of the path* p_i *and* SVV_{CP_x} *is the start value vector stored at the cut-point* CP_x.

In Figure 5.1, for path p_2, $StartSVV = \langle C : \langle c\rangle, A : \langle a_3a_2a_1a_0\rangle, A\_X : \langle x_3x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle q_3q_2q_1q_0\rangle, count : \langle t_2t_1t_0\rangle, prod : \langle \text{-}\rangle\rangle$

Definition 7 (Exit symbolic value vector (ExitSVV)). $ExitSVV$ *is the symbolic value vector generated after the symbolic execution of a path* p *through forward substitution using the values of the variables at bit level.*

For the path p_2 in Figure 5.1, after its symbolic execution, $ExitSVV = \langle C : \langle 0\rangle, A : \langle s_4s_3s_2s_1\rangle, A\_X : \langle s_0x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle 0q_3q_2q_1\rangle, count : \langle t'_2t'_1t'_0\rangle, prod : \langle \text{-}\rangle\rangle$

Computation of a path is accomplished through symbolic execution of the path using the $StartSVV$ for that path and $ExitSVV$ of the path is obtained. The algorithm for finding equivalence between two paths is described in Algorithm 20 with terms defined in Table 5.1. Another useful notion is as follows.

Definition 8 (SVV at cut-point). *Let fsmd* $= \langle Q, q, I, V, O, f, h\rangle$ *is an FSMD and* $\{p_1, p_2, \dots, p_k\}$ *is the set of paths from the cut-point* CP_x *to* CP_y *where* CP_x *and* $CP_y \in Q$, *then for* $p_i \in \{p_1, p_2, \dots, p_k\}$, *if the corresponding equivalent path is found between the corresponding cut-point pair in the another FSMD then,* $SVVCP_y \leftarrow ExitSVV_{p_i}$ *replacing any previous symbolic value vector stored (if any),* In Figure 5.1, at cut-point CP_3, the final SVV will be stored depending on the order of execution of the paths p_2 and p_3. If p_2 is executed at the last then $SVV\ at\ cut-point = \langle C : \langle 0\rangle, A : \langle s_4s_3s_2s_1\rangle, A\_X : \langle s_0x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle 0q_3q_2q_1\rangle, count : \langle t_2't_1't_0'\rangle, prod : \langle\text{-}\rangle\rangle$ If p_3 is executed at the last then $SVV\ at\ cut-point = \langle A : \langle s_4s_3s_2s_1\rangle, A\_Q : \langle s_0q_3q_2q_1\rangle, M : \langle m_3m_2m_1m_0\rangle, count\langle t_2't_1't_0'\rangle, prod : \langle\text{-}\rangle\rangle$

5.4.2 Value match between corresponding variables

After the symbolic execution of the two paths the values of the corresponding bits of the corresponding variables of the two exit symbolic value vectors (one from reference design and another from user design) are then matched. For the corresponding variables having specific values, if the value is matched then the specific value of the variable is replaced with the same symbols in both the designs and the specific values are stored for future analysis (if required). The basic equivalence checker method over integer data presented in the previous section 5.2 terminates if the values of the corresponding variables do not match (from Definition 1). The paths are declared as equivalent if the values of the common variables match. The current bit-level equivalence checker also terminates if the value of the each bit of a corresponding variable does not match with the value of the corresponding bit of the corresponding variable with exception for some special cases. In some complex computer arithmetic algorithms such as multiplication, division, output is iteratively built bit by bit through a loop. In addition with this, the output may reside in more than one variable i.e composed variables (discussed in the previous section) having different initial values, then the equivalence checking task becomes harder. In such case, before the completion of the loop, the variable (single or composed) holding the output may contain data in some bits which is not part of the output yielding some matching bits and some mismatching bits with its corresponding variable during value match. However, the bits having non-output data will eventually be filled up with output or result data after complete iteration. We now introduce some notions related to this.

Definition 9 (Full bit match(mismatch)). *Let fsmd^R = $\langle Q^R, q^{0,R}, I, V^R, O, f^R, h^R\rangle$ and fsmd^U = $\langle Q^U, q^{1,U}, I, V^U, O, f^U, h^U\rangle$ be two FSMDs having identical input and output sets, I and O, respectively. Let*
$ExitSVV^R = \langle v_1^R : \langle b[m_1] \ldots, b[n_1]\rangle, v_2^R : \langle b[m_2] \ldots, b[n_2]\rangle \ldots v_k^R : \langle b[m_k] \ldots b[n_k]\rangle\rangle$ and $ExitSVV^U = \langle v_1^U : \langle b[m_1] \ldots, b[n_1]\rangle, v_2^U : \langle b[m_2] \ldots, b[n_2]\rangle \ldots v_k^U : \langle b[m_k] \ldots b[n_k]\rangle\rangle$
are the exit symbolic value vector belonging to fsmd^R and fsmd^U respectively, whose values are needed to be matched equivalence checking.

- *Full bit match for the corresponding variable v_i^R and v_i^U where $v_i^R \in V^R$ and $v_i^U \in V^U$ is the matching of all of the corresponding bits of those variables.*
- *Full bit mismatch for the corresponding variable v_i^R and v_i^U where $v_i^R \in V^R$ and $v_i^U \in V^U$ is the mismatching of all of the corresponding bits of those variables.*

Definition 10 (Partial bits match(mismatch)). *Let $fsmd^R = \langle Q^R, q^{0,R}, I, V^R, O, f^R, h^R\rangle$ and $fsmd^U = \langle Q^U, q^{1,U}, I, V^U, O, f^U, h^U\rangle$ be two FSMDs representing reference design and user design respectively have identical input and output sets, I and O. Let*
$ExitSVV^R = \langle v_1^R : \langle b[m_1] \ldots, b[n_1]\rangle, v_2^R : \langle b[m_2] \ldots, b[n_2]\rangle \ldots v_k^R : \langle b[m_k] \ldots b[n_k]\rangle\rangle$ *and*
$ExitSVV^U = \langle v_1^U : \langle b[m_1] \ldots, b[n_1]\rangle, v_2^U : \langle b[m_2] \ldots, b[n_2]\rangle \ldots v_k^U : \langle b[m_k] \ldots b[n_k]\rangle\rangle$
are the exit symbolic value vector belonging to $fsmd^R$ and $fsmd^U$ respectively, whose values are needed to be matched equivalence checking.

- *Partial bits match for a corresponding variable v_i^R and v_i^U where $v_i^R \in V^R$ and $v_i^U \in V^U$ is when the value of the bits within the bit ranges $\langle b[k:l], \ldots, b[k':l']\rangle$ of the corresponding variables matches.*

- *Partial bits mismatch for a corresponding variable v_i^R and v_i^U where $v_i^R \in V^R$ and $v_i^U \in V^U$ is when the value of the bits within the bit ranges $\langle b[k:l], \ldots, b[k':l']\rangle$ of the corresponding variables does not match.*

In Figure 5.1 and Figure 5.2, during the value match for paths p_2 and p_2', the corresponding exit value vectors are generated as follows after the symbolic execution of the paths respectively.
ExitSVV (of p_2): $\langle C : \langle 0\rangle, A : \langle s_4s_3s_2s_1\rangle, A\_X : \langle s_0x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle 0q_3q_2q_1\rangle, count : \langle t_2't_1't_0'\rangle, prod : \langle \text{-}\rangle\rangle$
ExitSVV (of p_2'): $\langle A : \langle s_4s_3s_2s_1\rangle, A\_Q : \langle s_0q_3q_2q_1\rangle, M : \langle m_3m_2m_1m_0\rangle, count\langle t_2't_1't_0'\rangle, prod : \langle \text{-}\rangle\rangle$

It is to be noted that according to the naming convention used in this work, A_X denotes that the variables A and X are composed variables and together they will hold some ordered data bits in Figure 5.1. Similarly A and A_Q are composed variables containing some ordered data bits in Figure 5.2. Those two composed variables from the two designs are also corresponding variables, therefore, their value will be matched in a manner that A and A_X of ExitSVV (of p_2) will be matched with A and A_Q of ExitSVV (of p_2') respectively. In order to match values between them, we get a partial match and partial mismatch. The composed variables are of total 8 bits among which $A[3:0]$ matches, $A\_X[3]$ and $A\_Q[3]$ bit matches. However, $A\_X[2:0]$ and $A\_Q[2:0]$ mismatches. The other variables are not mentioned here to keep this example simple.

The value match of the corresponding variables may yield the following situations. (*i*) $full\ bit\ match$ (*ii*) $full\ bit\ mismatch$ (*iii*) $partial\ bits\ match$ and $partial\ bits\ mismatch$ For $full\ bit\ match$, the equivalence checker proceeds, for $full\ bit\ mismatch$ the equivalence checker terminates, the equivalence checking task is not terminated immediately for $partial\ bits\ match$ and $partial\ bits\ mismatch$ if the following conditions hold along with an assumption upon satisfaction of those conditions.

Condition-1: The path (let p) for which the situation occurs is a part of loop

Condition-2: The path p contains shift operations performed on the variables yielding partial bits match and partial bits mismatch.

Assumption: If Condition-1 and Condition-2 satisfies then the output may be building iteratively in a pattern in the designs and eventually the mismatch will be resolved after full iteration of the loop.

With this assumption, the equivalence checker further determines if a mismatch will be dealt with a special analysis or the equivalence checker can not decide further and need to terminate. This procedure is outlined in Algorithm 22. As, the equivalence checker does not unroll any loop during its execution. it relies on several design observations generated during the execution of the equivalence checking in the form of predicates and applying some predefined rules on those observations in order to deduce more observations and to make some decisions and also to extract a pattern through which the output builds. The successful pattern extraction may help the equivalence checker to infer data ranges after completion of a loop without unrolling the loop. The procedure of matching values of the corresponding variables (single or composed) are described in Algorithm 21 with the terms illustrated in the Table 5.2.

5.4.3 Special analysis to check specific value of a symbol

When the mismatch bits of corresponding variables are marked for the special analysis it is then checked if one of the bit contains a symbol and the other one contains a specific value. If both the mismatched bits hold mismatched symbols then this implies that in general case the bits will mismatch i.e. they will always mismatch, therefore there is no need for special analysis and the equivalence checker will terminate. However, if one of the value is a symbol and the other one is a specific value then it is checked whether the bit having a symbol actually takes the same specific value in all possible cases that of the other corresponding bit from the other design. In order to do that, reachability definitions of the path (where the mismatch has occurred, say p_i) are computed. The notion of path composition (in terms of path characteristic) and reachability is as follows.

Definition 11 (Characteristics of a path). *The characteristic formula $\tau_\alpha(\overline{v}, \overline{v}_f, O)$ of a path α is $R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$, where s_α is the data transformation and O_α is the output list in the path α, $\overline{v}$ represents a vector of values of the variables of $I \cup V$, $\overline{v}_f$ represents a vector of values of the variables of V. The formula captures the following: if the condition of execution R_α of the path α is satisfied by the (initial) vector $\overline{v}$ at the beginning of the path, then the path is executed and after execution, the final vector $\overline{v}_f$ of variable values becomes $s_\alpha(\overline{v}_f)$ and the output $O_\alpha(\overline{v})$ is produced.*

Let $\tau_\alpha(\overline{v}, \overline{v}_f, O) : R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$ be the characteristic formula of the path α and $\tau_\beta(\overline{v}, \overline{v}_f, O) : R_\beta(\overline{v}) \wedge (\overline{v}_f = s_\beta(\overline{v})) \wedge (O = O_\beta(\overline{v}))$ be the characteristic formula of the path β. The characteristic formula for the concatenated path $\alpha\beta$ is $\tau_{\alpha\beta}(\overline{v}, \overline{v}_f, O) = \exists \overline{v}_\alpha \exists O_1 \exists O_2 (\tau_\alpha(\overline{v}, \overline{v}_\alpha, O_1) \wedge \tau_\beta(\overline{v}_\alpha, \overline{v}_f, O_2)) = R_\alpha(\overline{v}) \wedge R_\beta(s_\alpha(\overline{v})) \wedge (\overline{v}_f = s_\beta(s_\alpha(\overline{v}))) \wedge (O = O_\alpha(\overline{v}) O_\beta(s_\alpha(\overline{v})))$. O is the concatenated output list of $O_\alpha(\overline{v})$ and $O_\beta(s_\alpha(\overline{v}))$.

Definition 12 (A path reachable from a set of paths ($S_{Rto}^{p_i^{fsmd}}$)). *$fsmd = \langle Q, q, I, V, O, f, h \rangle$ is an FSMD and*

$P = \{p_1, p_2, \ldots, p_k\}$ is its path cover. For any path $p_i \in P$, $S_{Rto}^{p_i^{fsmd}}$ is the set of paths containing the paths or the concatenation of the paths from the path cover P from which p_i can be reached in the fsmd

In the Figure 5.1, for the path p_3, $S_{Rto}^{p_i^{fsmd}} = \{P_1, P_2P_4, P_3P_4\}$.

Definition 13 (A set of paths emerging from a path ($S_{Rfrom}^{p_i^{fsmd}}$)). *$fsmd = \langle Q, q, I, V, O, f, h\rangle$ is an FSMD and*
$P = \{p_1, p_2, \ldots, p_k\}$ is its path cover. For any path $p_i \in P$, $S_{Rfrom}^{p_i^{fsmd}}$ is the set of paths containing the paths or the concatenation of the paths from the path cover P which emerges from p_i in the fsmd

In the Figure 5.1, for the path p_3, $S_{Rfrom}^{p_i^{fsmd}} = \{P_4P_2, P_4P_3, P_5\}$

The value of the symbol is determined from each path belongs to the reachable definitions of the path p_i i.e. $S_{Rto}^{p_i^{fsmd}}$, and substituted in the corresponding bit of the appropriate variable in the current path (p_i). After substitution, the path, p_i is symbolically executed and the value is matched with the corresponding variable in the other FSMD. If we get a new symbol after the symbolic execution instead of getting a specific value to resolve the mismatch occurred earlier, another check is performed recursively on all the paths which do emerges from the path p_i and as well as belongs to the reachable definitions of p_i, i.e. $S_{Rfrom}^{p_i^{fsmd}}$ until a fixed point is reached. The procedure is described in the Algorithm 24 and Algorithm 23 along with the description of the terms used in it in the Table 5.4.

5.4.4 Identifying data building pattern and inferring data range after completion of loop (for special cases)

After the value match is performed for the corresponding variables of the two FSMDs for a pair of paths, the equivalence checker tries to find the shifting pattern through which data builds if it has made the assumptions based on the satisfaction of the conditions mentioned earlier.The shift characteristic is deduced from the design observations generated in the form of predicates at the several phases of the execution using some predefined rules. Some of the predicates are as follows.

Predicate-1: (SHR(v):n) [Logical right shift of v by n bits]

Predicate-2: (SHRA(v):n) [Arithmetic right shift of v by n bits]

ALGORITHM 19: EquivalenceChecker($fsmd^R$, $fsmd^U$)

```
Input  : Reference FSMD ($fsmd^R$) and the user FSMD ($fsmd^U$)
Output: Decision whether the two input FSMDs are equivalent
1 Find loops through DFS back edges and introduce cut-points;
2 $CmpVarSet \leftarrow$ Find variable composition and their ranges (through concatenation);
3 $\{ObvSet^R, ObvSet^U, CmpVarCorrSet\} \leftarrow$ Do live variable analysis and compute
  correspondence;
4 Traverse cut-points in BFS order starting from the start state ;
5 foreach corresponding cut-points $CP_j^R$ and $CP_j^U$ (being j as BFS level) do
6     $EmPathList^R \leftarrow$ FindEmergingPathsToNextCP($CP_j^R$, $CP_{nextOFj}^R$);
7     $EmPathList^U \leftarrow$ FindEmergingPathsToNextCP($CP_j^U$, $CP_{nextOFj}^U$);
8     $SameCondPathList \leftarrow$ FindPathPairWithSameCondition($EmPathList^R$,
      $EmPathList^U$);
9     foreach path pair ($P_i^R$, $P_i^U$) $\in SameCondPathList$ do
10        $\{SCset_{L_i}^R, SCset_{L_i}^U\} \leftarrow$ CheckPathEqv($P_i^R, P_i^U, \overline{V}_{i_s}^R, \overline{V}_{i_s}^U, \overline{Vsp}_i^R, \overline{Vsp}_i^U$);
11        Store ExitValueVector at cut-points $CP_{nextOFj}^R$ and $CP_{nextOFj}^U$;
12        if ($P_i \in$ loop $L_i$ && all paths $\in L_i$ is computed && path contains live shift
          var) then
13            $\{LIS(L_i^D), LIS(L_i^{D'})\}$
              $\leftarrow$ DeduceLoopInvariantShifts($\{SCset_{L_i}^R, SCset_{L_i}^U\}$);
14            if loop invariant shifts matches then
15                DoInterpolate(loopCount, $LIS(L_i^D), LIS(L_i^{D'})$);
16                Update ExitValueVector at cut-points $CP_{nextOFj}^R$ and $CP_{nextOFj}^U$;
17            else
18                Terminate;    /* two designs are not equivalent */
19            end
20        end
21    end
22 end
23 Output successful equivalence decision;
```

ALGORITHM 20: CheckPathEqv($P_i^D, P_i^{D'}, \overline{V}_{i_s}^D, \overline{V}_{i_s}^{D'}, \overline{Vsp_i}^D, \overline{Vsp_i}^{D'}$)

```
Input  : Two path segments of the two FSMDs along with their symbolic start value
         vectors and specific value vector
Output: Decision if the two paths are equivalent, exit symbolic value vector and the
         data shifting pattern in that path
1  ${\{Obv_i^D\}, \overline{V}_{i_E}^D} \leftarrow$ ComputePath($P_i^D, \overline{V}_{i_s}^D$);
2  $ObvSet^D \leftarrow ObvSet^D \cup \{Obv_i^D\}$;
3  ${\{Obv_i^{D'}\}, \overline{V}_{i_E}^{D'}} \leftarrow$ ComputePath($P_i^{D'}, \overline{V}_{i_s}^{D'}$);
4  $ObvSet^{D'} \leftarrow ObvSet^{D'} \cup \{Obv_i^D\}$;
5  $\{ObvSet^D, ObvSet^{D'}, decision\}$
   $\leftarrow$MatchValue($P_i^D, P_i^{D'}, \overline{V}_{i_E}^D, \overline{V}_{i_E}^{D'}, CmpVarCorrSet, ObvSet^D, ObvSet^{D'}$);
6  $\{SC(P_i^D), SC(P_i^{D'})\}\leftarrow$ DeduceShiftChateristics($\{ObvSet^D, ObvSet^{D'}\}$); /* for
   live variables */
7  foreach $P_i^D \in loopL_i$ do
8      $SCset_{L_i}^D \leftarrow SCset_{L_i}^D \cup SC(P_i^D)$;
9      $SCset_{L_i}^{D'} \leftarrow SCset_{L_i}^{D'} \cup SC(P_i^{D'})$;
10 end
```

P_i^D: i^{th} path in design D (if $D = R(reference)$ then $D' = U(user)$)
$\overline{V}_{i_s}^D$: start symbolic value vector in the i^{th} path in design D
$\overline{Vsp_i}^D$: exit specific value vector in the i^{th} path in design D
Obv_i^D: set of observations during computing path P_i in design D
$SC(P_i^D)$: deduced shift characteristics of path P_i^D
$SCset_{L_i}^D$: set of shift characteristics of paths $\in$ loop L_i

Table 5.1: Illustration of terms for Algorithm 20

Predicate-3: (SHL(v):n) [Logical left shift of v by n bits]

Predicate-4: NotUsedWithinLoop(v):(CP_x, CP_y)) [Value of v is not used in a loop within cut-point CP_x and CP_y]

Predicate-5: ValueMatch($v^R[m : n]$, $v^U[m : n]$) [Value matched for the bits in the range $[m : n]$ of the two corresponding variables from the reference and user design respectively]

Predicate-6: IgnoredMismatches($v_i^R[m : n]$, $v_i^U[m : n]$) [mismatches occurred for the bits in the range $[m : n]$ of the two corresponding variables from the reference and user design respectively do not cause termination of the equivalence checker]

Predicate-7: FullBit(v:$size$) [All of the bits of v will be considered]

Predicate-8: IncrementFromMsb(v:n) [Bits of v will be considered incrementally from the most significant bit by n bits]

ALGORITHM 21: MatchValue($P_i^D, P_i^{D'}, \overline{V}_{i_E}^D, \overline{V}_{i_E}^{D'}, CmpVarCorrSet, ObvSet^D, ObvSet^{D'}$)

Input : Two path segments of the two FSMDs along with their symbolic exit value vectors, set of corresponding variables and set of observations

Output: Matches symbolic values of the corresponding variables and allow(or forbid) mismatch

```
1  foreach corresponding single variable $sv^D$ and $sv^{D'}$ do
2      if ($\forall$ bits $b_i \in sv^D$ and $sv^{D'}$ and $val(b_i^D) = val(b_i^{D'})$) then
3          proceed;
4      else
5          terminate;
6      end
7  end
8  foreach pair $(u, v) \in CmpVarCorrSet$ s.t. $P_i \in Range(u, v)$ do
9      if ($\forall$ bits $b_i \in u, v$ and $val(b_i^D) = val(b_i^{D'})$) then
10         Return true;
11     else if $\exists$ bit ranges $\{ b_{[k:l]}, .., b_{[k':l']}\}$ s.t. $val(b_{[k:l]}^D, .., b_{[k':l']}^D) \neq val(b_{[k:l]}^{D'}, .., b_{[k':l']}^{D'})$
       then
12         $\{b_{[i:j]}..b_{[i':j']}\}$
           $\leftarrow$ ForbidMismatchRange($(u, v), (b_{[k:l]}..b_{[k':l']}), P_i, ObvSet^D, ObvSet^{D'}$);
13         foreach $b_{[x:y]} \in \{b_{[i:j]}, .., b_{[i':j']}\}$ do
14             if ($val(b_{[x:y]}^D) = specific$ and $val(b_{[x:y]}^{D'}) = symbolic$) then
15                 CheckAllPossibleSpecVal ($b_{[x:y]}^{D'}, P_i^{D'}, val(b_{[x:y]}^D)$);
16                 UpdateObv($ObvSet^D, ObvSet^{D'}$);
17             else if ($val(b_{[x:y]}^{D'}) = specific$ and $val(b_{[x:y]}^D) = symbolic$) then
18                 CheckAllPossibleSpecVal ($b_{[x:y]}^D, P_i^D, val(b_{[x:y]}^{D'})$);
19                 UpdateObv($ObvSet^D, ObvSet^{D'}$);
20             else
21                 UpdateObv($ObvSet^D, ObvSet^{D'}$) and terminate;
22             end
23         end
24     else
25         UpdateObv($ObvSet^D, ObvSet^{D'}$) and return true;
26     end
27 end
```

ALGORITHM 22: ForbidMismatchRange($(u, v), (b_{[k:l]}, .., b_{[k':l']}), P_i, ObvSet^D, ObvSet^{D'}$)

```
Input  : A path, set of mismatch ranges of bits occurred in the path and set of
         observations of both the input FSMDs
Output: Decision for each mismatch range to allow or forbid
1  if initially $u = v$ then
2      For single mismatched bit ← do not allow;
3  else if initially $u \neq v$ and $\exists$ matched bits then
4      foreach mismatched variable $w \in u$ or $v$ do
5          if $w$ is not live in all the paths originating from $P_i$ then
6              allow mismatch;
7          else if $w$ is live but the original value of $w$ is overwritten then
8              allow mismatch;
9          else if $w$ is live in any path originating from $P_i^D$ then
10             if mismatched bits (of the corresponding composed variable) of $w$
               matches with another corresponding single variable and as single
               variable, the mismatched bits of $w^D$ and $w^{D'}$ have no future use then
11                 allow mismatch;
12             end
13         end
14     end
15 end
```

ALGORITHM 23: CheckAllPossibleSpecVal ($b^T_{[x:y]}, P^T_i, val(b^{T'}_{[x:y]})$)

```
Input  : Mismatched bit ranges and the path in which the specific value of the
         mismatched symbols are to checked in all possible cases
Output: Decision whether the forbidden mismatches resolved in all possible cases
1  $S^{P^T_i}_{Rto}$ ← ReachableTo($P^T_i$);    /* returns set of paths from where
   $P^T_i$ is reachable */
2  foreach path $P^T_k \in S^{P^T_i}_{Rto}$ do
3      if (CheckSpecVal($P^T_k, P^T_i, b^T_{[x:y]}, b^{T'}_{[x:y]}, S^{P^T_i}_{Rto}$)) then
4          Return true;
5      else
6          return false;
7      end
8  end
```

ALGORITHM 24: CheckSpecVal($P_k^T, P_i^T, b_{[x:y]}^T, b_{[x:y]}^{T'}, S_{Rto}^{P_i}$)

```
Input  : A path $P_i^T$ where mismatched bits occurred, one of the path $P_k^T$ belongs to
         set of paths from where $P_i^T$ is reachable, mismatched bits and the set of all
         paths from where $P_i^T$ is reachable
Output: Decision whether the forbidden mismatches resolved in each cases
1  if ($\overline{V}_{k_E}^T$ is not computed) then
2      $\{\{Obv_k^T\}, \overline{V}_{k_E}^T\} \leftarrow$ ComputePath($P_k^T, \overline{V}_{k_S}^T$);
3  end
4  foreach bit $b_i \in b_{[x:y]}^T$ do
5      $Var_{symbolicVal}(b_i)_{\overline{V}_{i_S}^T} \leftarrow Val(Var_{symbolicVal}(b_i)_{\overline{V}_{k_E}^T})$;
6      $\{\{Obv_i^T\}, \overline{V}_{i_E}^T\} \leftarrow$ ComputePath($P_i^T, \overline{V}_{i_S}^T$);
7      MatchValue($P_i^T, P_i^{T'}, \overline{V}_{i_E}^T, \overline{V}_{i_E}^{T'}, CmpVarCorrSet, ObvSet^T, ObvSet^{T'}$);
8      if (value matched) then
9          if ($Val(Var_{symbolicVal}(b_i)_{\overline{V}_{i_E}^T}) \neq Val(Var_{symbolicVal}(b_i)_{\overline{V}_{k_E}^T})$) then
10             $S_{Rfrom}^{P_i^T} \leftarrow$ ReachableFrom($P_i^T$);      /* returns set of paths
               reachable from $P_i^T$ */
11             foreach path $P_l^T \in \{S_{Rto}^{P_i^T} \bigcap S_{Rfrom}^{P_i^T}\}$ do
12                 CheckSpecVal($P_i^T, P_l^T, b_{[x:y]}^T, b_{[x:y]}^{T'}, S_{Rto}^{P_i^T}$);    /* checking only
                   within loops */
13             end
14         else
15             return true;
16         end
17     else
18         terminate;
19     end
20 end
```

sv^D: single variable of design D
(u,v): correspondence between composed variables (in the example, u = A,X and v = A,Q)
ForbidMismatchRange(): given a set of mismatch ranges of corresponding composed variables and set of observations, it deduces which mismatch range can be ignored during value match and returns the set of ranges which can not be ignored
$val(b^D_{[x:y]})$: value of bits $b_{[x:y]}$ in design D

Table 5.2: Illustration of terms for Algorithm 21

$CmpVarSet$: set of composed variables
$ObvSet^R$: set of observations in design R
$CmpVarCorrSet$: set of corresponding composed variables
$SCset^D_{L_i}$: set of shift characteristics of paths $\in$ loop L_i
$LIS(L^D_i)$: loop invariant shifts of loop L_i in design D

Table 5.3: Illustration of terms for Algorithm 19

Apart from these basic predicates, more predicates can be generated using AND operations on these basic predicates which may capture extra observations. Some predefined rules can also be applied on the predicates in order to obtain new predicates. Finally, when the data shifting characteristics of all the paths of a loop are generated in terms of predicates. After a successful generation of data building pattern from each path belonging to a loop, the equivalence checker then checks if their exists any loop invariant data building pattern again using the observations and predefined rules. The set of observations are always kept updated by the equivalence checker. If a loop invariant data building pattern is identified, it is then further used to infer data range after the completion of the loop using interpolation. Following is a form of data building pattern composed of the basic predicates which is identified in the designs presented Figure 5.1 in the path belonging to the loop present in the design.

Data building pattern

ValueMatch((FullBit(A^{Ref}:4) $\wedge$
IncrementFromMsb(A_X^{Ref}:1)), (FullBit(A^{User}:4) $\wedge$ IncrementFromMsb(A_Q^{User}:1)))

This implies the following.
In corresponding composed variables ($\{A^{Ref}, A\_X^{Ref}\}$, $\{A^{User}, A\_Q^{User}\}$), output is incrementally building by adding one bit from MSB of A_X^{Ref} or A_Q^{User} along with the four bits of A^{Ref} or A^{User} respectively.

Using this pattern and the loop count which is four in this example, the equivalence checker now infer the data range residing in the composed variables ($\{A^{Ref}, A\_X^{Ref}\}$,

$\overline{V}^T_{k_E}$: exit symbolic value vector during the computation of path P_k in design T
Obv^T_k: set of observations noted during the computation of path P_k in design T
$Var_{symbolicVal}(b_i)_{\overline{V}^T_{i_S}}$: corresponding variable of the symbolic value present in the bit b_i in the vector $\overline{V}^T_{i_S}$
$Val(Var_{symbolicVal}(b_i)_{\overline{V}^T_{i_S}})$: value of the corresponding variable of the symbolic value present in the bit b_i in the vector $\overline{V}^T_{i_S}$

Table 5.4: Illustration of terms for Algorithm 24

$\{A^{User}, A\_Q^{User}\}$) after the completion of the loop iterations.

Inferencing data range after n iterations

ValueMatch((FullBit(A^{Ref}:4) $\wedge$ MatchRange($A\_X^{Ref}[msb : msb - n + 1]$)), (FullBit($A^{User}$:4) $\wedge$ MatchRange($A\_Q^{User}[msb : msb - n + 1]$)))

This implies the following.
After n iterations of the loop,the variables $A[3 : 0]$ and $A\_X[msb : msb - n + 1]$ in the reference design will match with the variables $A[3 : 0]$ and $A\_Q[msb : msb-n+1]$ in the user design. Where msb is the most significant bit of the variables.

This bit-level equivalence checker assumes that all the corresponding variables (single or composed) must be of same size and the loop counts are fixed and known. The bit-level equivalence checking method along with the generation of predicates, identifying data building pattern, interpolation are explained in detail using a suitable example in the next section.

5.5 Equivalence checking for shift and add multiplier

This section includes an example and a detail analysis that would be performed by the bit-level equivalence checker. The example consists of a reference design and a user design. At first, the designs described along with their design observation and complexities. Then the equivalence checking is shown in each significant path of the designs along with the final equivalence decision.

5.5.1 Reference and user designs

The two designs (Figure 5.1 and Figure 5.2) are very good example to explore the capabilities of the proposed bit-level equivalence checking method. Figure 5.1 is the reference design which consists of a shift and add multiplication (of two 4 bit data) flowchart where result and the multiplier are right shifted by 1 bit at each step. Figure 5.2 is the user design which is a variation of shift and add multiplication of two 4 bit data where the multiplier is right shifted by 1 bit. The size of the variables in the designs having multiple bits are shown in a range such as $[msb : lsb]$ where msb denotes the most significant bit (MSB) and the lsb denotes the least significant bit (LSB). For a variable with single bit, the size is not shown in the range format. The cut-points are shown in yellow colored nodes in the designs. The path cover i.e. the paths between cut-points are shown in dashed and bent arrow line. Each path is assigned a name such as p_1, p_2, p_3 etc. for the ease of explaining the methodologies and the name is shown next to the dashed and bent arrow line.

Design observations

Some of the operations are shown in different colors. The different color signifies different issues in determining the equivalence between the designs.

Mismatched variables and operations: Blue coloring is used to denote the mismatched variables and operations. Concatenated variables are denoted within curly braces $\{\}$ and with comma between two variables for example, $\{C, A[3:0]\}$ is the concatenation of the variable C and A. For example, the variables C, A_X are present in only reference design and the result of the same addition operation $A[3:0] + M[3:0]$ is assigned to $\{C, A[3:0]\}$ in the reference design and to $\{A[3:0], A\_Q[3]\}$ in the user design. It is to be noted that, for the operation $\{C, A[3:0]\}$ $\leftarrow A[3:0] + M[3:0]$, the carry of the addition operation is assigned to C and the sum resides in $A[3:0]$ having the LSB and MSB of the sum at $A[0]$ and $A[3]$ respectively. On the other hand, for the operation $\{A[3:0], A\_Q[3]\} \leftarrow A[3:0] + M[3:0]$, the carry is assigned to $A[3]$ sum resides in $\{A[2:0], A\_Q[3]\}$ having the LSB at $A\_Q[3]$ and MSB at $A[2]$.

Different variable composition: Different variable composition is shown in red color. It is to be noted that in both the designs, the path p_5 contains an 8-bit variable $prod[7:0]$ which hold the multiplication result which actually resides in the composed variables $\{A[3:0], A\_X[3:0]\}$ in the reference design and in $\{A[3:0], A\_Q[3:0]\}$ in the user design. Due the naming convention followed in this work (mentioned earlier in this chapter), the association of the variable X with A is denoted by A_X and the same holds for variable Q. Although there exist correspondence between the composed variable ($\{A[3:0], A\_X[3]\}$-$\{A[3:0], A\_Q[3]\}$), initially they contain different value. The variable A_Q holds the multiplier in the user design and multiplicand is stored in the variable M in both the designs.

Mismatched shifting: The mismatched shift operation is shown in magenta color and which depicts that the shift operation $\{C, A[3:0], A\_X[3:0]\} \leftarrow \{C, A[3:0], A\_X[3:0]\} \gg 1$ occurs only in the reference design, not in the user design. For the operations $Q[3:0] \leftarrow Q[3:0] \gg 1$ and $A\_Q[3:0] \leftarrow A\_Q[3:0] \gg 1$ appears dissimilar due to the variable name, however it is to be noted that A_Q is actually the variable Q denoting its association with the variable A according to the naming convention used in this work. Therefore the two shift operations are same.

Need for interpolation and extraction of loop invariant shift pattern: In the path p_5, in both the designs, eight bits are assigned to the $prod$ variable from the variables including two

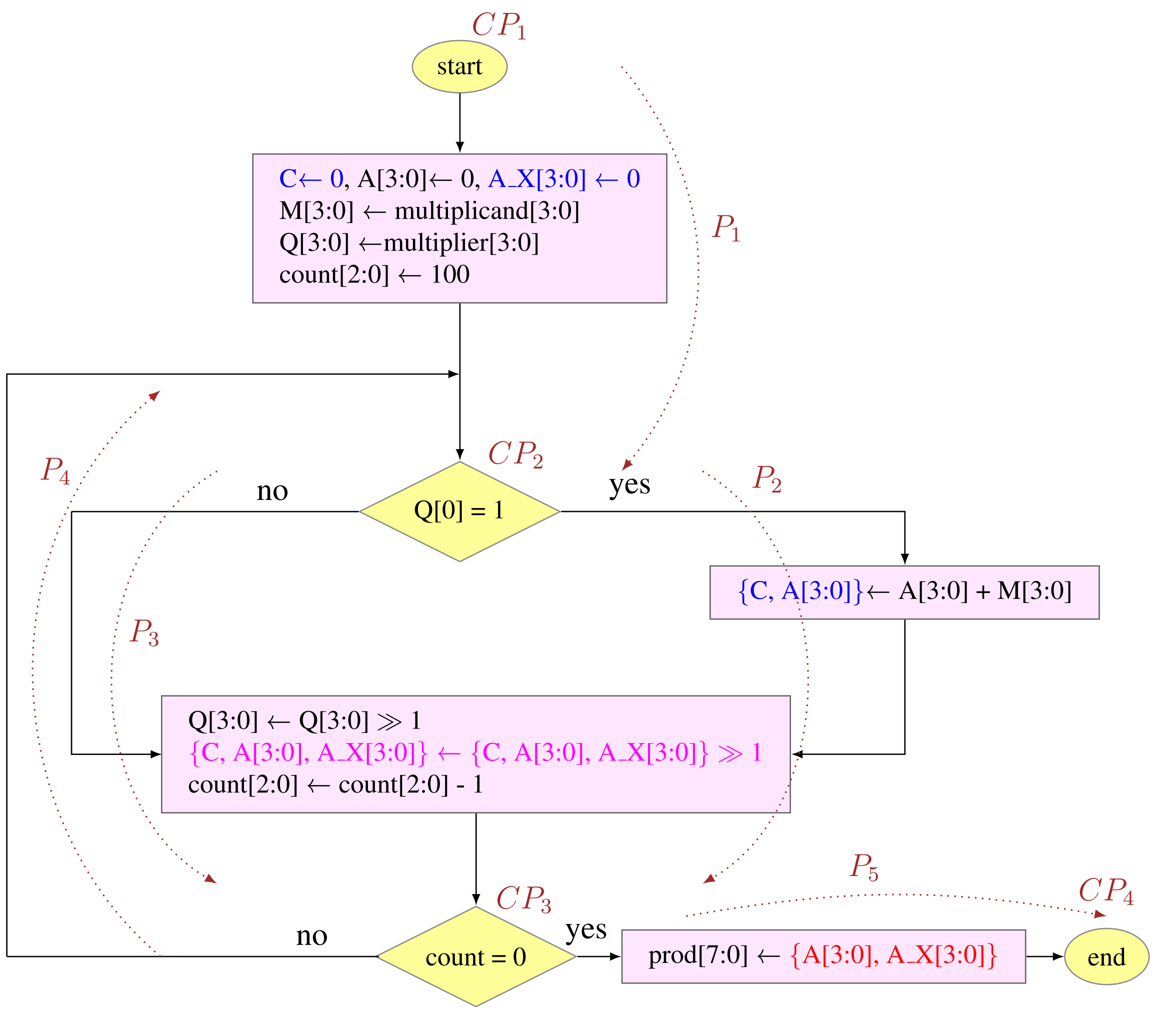
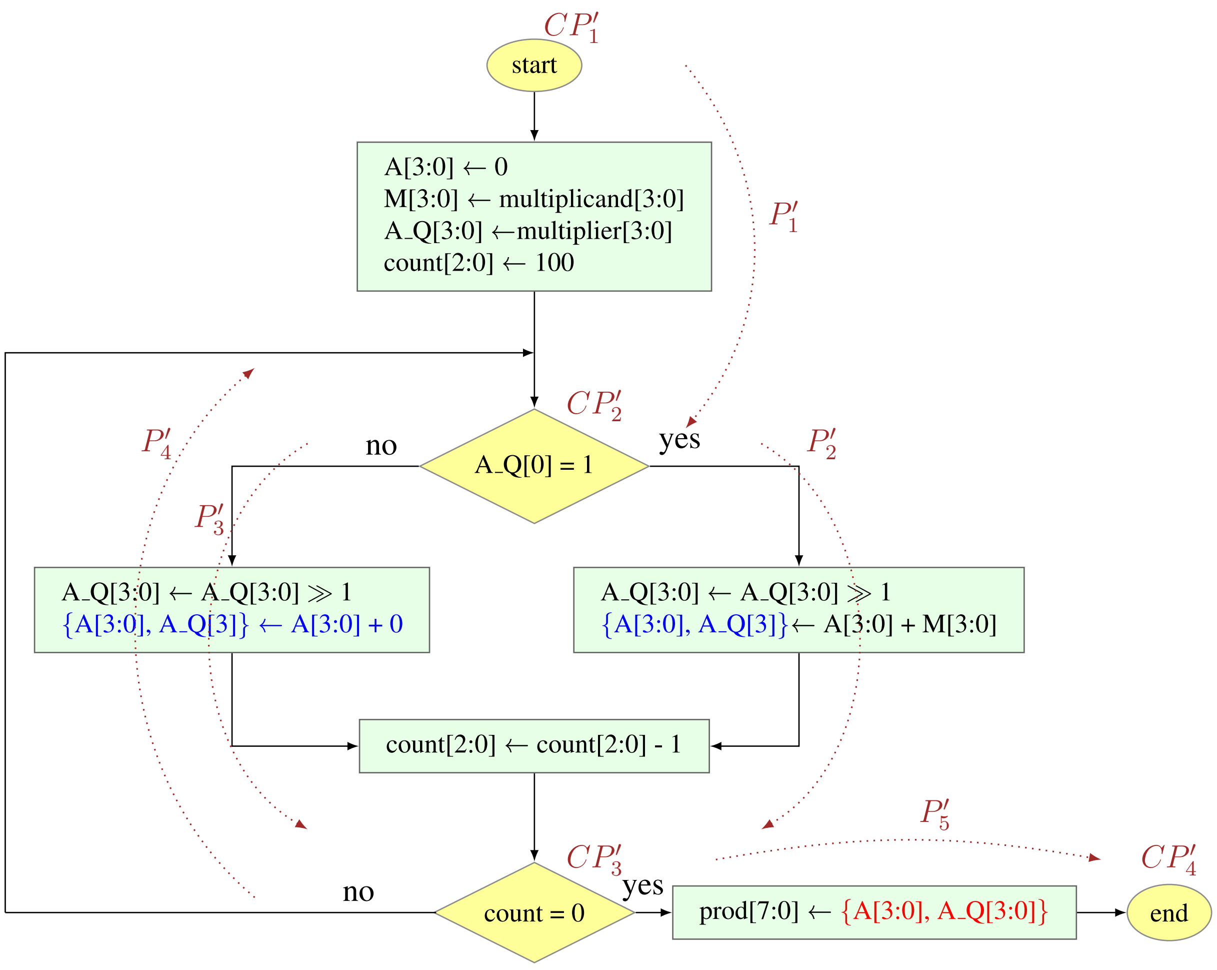

CP_1
start
C← 0, A[3:0]← 0, A_X[3:0] ← 0
M[3:0] ← multiplicand[3:0]
Q[3:0] ←multiplier[3:0]
count[2:0] ← 100
P_1
CP_2
Q[0] = 1
no
yes
P_2
{C, A[3:0]}← A[3:0] + M[3:0]
P_4
P_3
Q[3:0] ← Q[3:0] ≫ 1
{C, A[3:0], A_X[3:0]} ← {C, A[3:0], A_X[3:0]} ≫ 1
count[2:0] ← count[2:0] - 1
CP_3
count = 0
no
yes
P_5
prod[7:0] ← {A[3:0], A_X[3:0]}
CP_4
end

Figure 5.1: Shift-add multiplication (of two 4 bit data) flowchart with right shifting result (C,A,Q) (reference design)

variables which contain different value initially such as A_X and A_Q. In these examples, an incremental bit-by-bit data building is performed through the loop. As the equivalence checking method does not unroll the loop, therefore, it will try to interpolate the data range after the loop from single symbolic execution. In order to do that, it will try to extract the loop invariant shift pattern using some observations and predefined rules and if it is successful to extract such pattern, it will then interpolate the bigger data range using the pattern, some predefined rules and the known, finite loop count.



CP_1'
start
A[3:0] ← 0
M[3:0] ← multiplicand[3:0]
A_Q[3:0] ←multiplier[3:0]
count[2:0] ← 100
P_1'
CP_2'
A_Q[0] = 1
no
yes
P_4'
P_3'
P_2'
A_Q[3:0] ← A_Q[3:0] ≫ 1
{A[3:0], A_Q[3]} ← A[3:0] + 0
A_Q[3:0] ← A_Q[3:0] ≫ 1
{A[3:0], A_Q[3]}← A[3:0] + M[3:0]
count[2:0] ← count[2:0] - 1
P_5'
CP_3'
count = 0
no
yes
prod[7:0] ← {A[3:0], A_Q[3:0]}
CP_4'
end

Figure 5.2: Shift-add multiplication (of two 4 bit data) flowchart with right shifting only Q (user design)

5.5.2 Overview of the equivalence checking steps

At first the equivalence checker introduces the cut-points (shown in yellow colored nodes) and then computes the path cover in both the designs. $\{CP_1, CP_2, CP_3, CP_4, \}$ and $\{CP_1', CP_2', CP_3', CP_4'\}$ are the set of cut-points are as shown in the reference and user design respectively. The path cover consists of the paths between the cut-points. The path covers computed for the reference design is $\{P_1, P_2, P_3, P_4, P_5\}$ and $\{P_1', P_2', P_3', P_4', P_5'\}$ for user design. The equivalence checker will start from the *start* cut-point in breath-first-search (BFS) manner in both the designs. For every path from the path cover between any two pair of cut-points in one design, the equivalence checker will try to find its equivalent path between the corresponding cut-point pairs in the another design. The corresponding cut-points are those having same conditional branching except the *start* and *end* cut-point. In this example the corresponding cut-points are (CP_1, CP_1'), (CP_2, CP_2'), (CP_3, CP_3'), (CP_4, CP_4'). It is noted that the condition of the cut-point $(CP_2,\, CP_2')$ is same as A_Q is actually the variable Q denoting its association with the variable A according to the naming convention. The paths (belonging to the path cover of both the designs) will be checked for equivalence who have same conditions. Therefore, in this example, the equivalence checking will be between (P_1, P_1'), (P_2, P_2'), (P_3, P_3'), (P_4, P_4') and (P_5, P_5'). The following subsections describe the detail equivalence analysis for these paths. As the path pair (P_4, P_4') does not contain any data transformation, its trivial analysis is not discussed here. The paths in both of the designs are symbolically executed using the values (symbolic) of the variables from the start vectors stored at the starting cut-point of the path yielding the exit vectors which will be stored at the end cut-point of that path. For example, path P_2 in the reference design with be symbolically executed with the start vector stored at cut-point CP_2 Then the values of the corresponding bits of the corresponding variables in the exit vector for the paths in the two designs are matched and the equivalence decision is made for the two paths. After checking all the paths of the path cover, the final equivalence decision is made.

5.5.3 Equivalence analysis for path $\mathbf{P_1}$ and $\mathbf{P_1'}$

The *start* cut-point will contain the null start vector. The paths P_1 and P_1' in both of the designs are symbolically executed. The exit vectors generated from the symbolic execution of the two paths are then matched for the values of the corresponding variables. For the corresponding variables having specific values, if the value is matched then the specific value of the variable is replaced with the same symbols in both the designs and the specific values are stored for future analysis (if required). Table 5.5 shows the path segments along with its exit symbolic value vector (ExitSVV) for the reference and the user design. The symbolic execution is performed with the symbolic values in the variables M, Q, A_Q and specific values of the variables C, A, A_X, $count$. The values of the common variables A, A_X, $count$ matched in the corresponding exit vectors, therefore, they are are replaced with same symbols. The matched bits are shown in purple color. For this path, all the common variables' value matched, therefore, the paths P_1 and P_1' are found to be equivalent.

5.5.4 Equivalence analysis for path $\mathbf{P_2}$ and $\mathbf{P_2'}$

Table 5.5 shows the exit vectors after the symbolic execution of the paths P_2 and P_2' using the start vector stored at CP_1 and CP_1' in the reference and the user design respectively. In the exit vector some bits have red color in order to denote the correspondence between the composed variables in the two designs.

Matching values

From the exit vectors of the reference and the user designs, it is seen that the values of the common single variables such as M, $count$ matches. All the bits of the variable A are also matched. However, the MSB bit of the variables A_X and A_Q i.e. $A\_X[3]$ and $A\_Q[3]$ matches and there is a mismatch between the bits $A\_X[2:0]$ and $A\_Q[2:0]$. As these paths are the part of loop and having shift

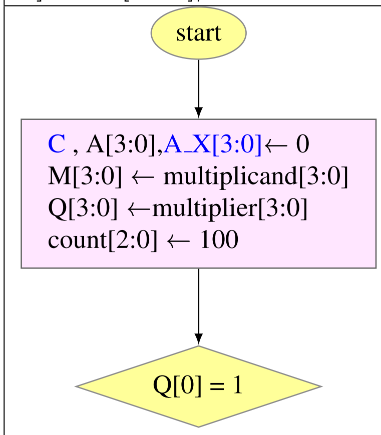
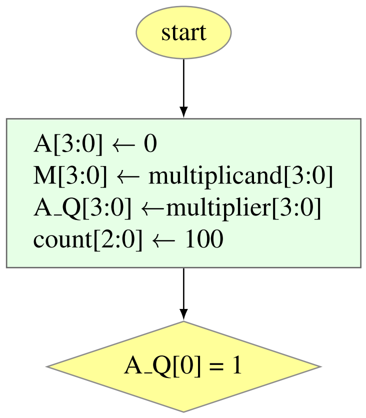
Reference Design	User Design
StartSVV: $\langle C, A[3:0], A\_X[3:0], M[3:0], Q[3:0], count[2:0], prod[7:0]\rangle$	StartSVV: $\langle A[3:0], A\_Q[3:0], M[3:0], count[2:0], prod[7:0\rangle$
 start C , A[3:0],A_X[3:0]← 0 M[3:0] ← multiplicand[3:0] Q[3:0] ←multiplier[3:0] count[2:0] ← 100 Q[0] = 1	 start A[3:0] ← 0 M[3:0] ← multiplicand[3:0] A_Q[3:0] ←multiplier[3:0] count[2:0] ← 100 A_Q[0] = 1
ExitSVV: $\langle C : \langle c\rangle, A : \langle a_3a_2a_1a_0\rangle, A\_X : \langle x_3x_2x_1x_0\rangle M : \langle m_3m_2m_1m_0\rangle, Q : \langle q_3q_2q_1q_0\rangle, count : \langle t_2t_1t_0\rangle prd : \langle\text{-}\rangle\rangle$	ExitSVV: $\langle A : \langle a_3a_2a_1a_0\rangle, A\_Q : \langle q_3q_2q_1q_0\rangle, M : \langle m_3m_2m_1m_0\rangle, count : \langle t_2t_1t_0\rangle, prod : \langle\text{-}\rangle\rangle$
SpecVV: $\langle C : \langle 0\rangle, A : \langle 0000\rangle, A\_X : \langle 0000\rangle, M : \langle -\rangle, Q : \langle -\rangle, count : \langle 100\rangle, prod : \langle\text{-}\rangle\rangle$	SpecVV: $\langle A : \langle 0000\rangle, A\_Q : \langle -\rangle, M : \langle -\rangle, count : \langle 100\rangle, prod : \langle\text{-}\rangle\rangle$

Table 5.5: Equivalence checking of path P_1 and P_1' (StartSVV: start symbolic value vector, ExitSVV: exit symbolic value vector, SpecVV: specific value vector)

operations performed on these composed variables with partial math along with partial mismatch, there is a possibility that some useful data is built incrementally through the shift operation with a pattern. Therefore, the mismatch does not cause the equivalence checking to be failed immediately, instead it perform some extra analysis using some design observations generated in course of the execution of the equivalence checking method and some predefined rules in order to find equivalence between the paths and a data building pattern in the paths.

Design observations

The observations are generated in the form of predicates. Observations those are needed for the analysis to check whether the aforesaid mismatches can be ignored or not, are as follows.

Ob-1: Least significant bits of $\_X^{Ref}$ are discarded by 1 bit and the original bits of $\_X$ are never used within the loop between CP_2 and CP_3
(NOTE: $\_X^{Ref}$ is the variable A_X of the reference design, its association is not shown here for the ease of explanation))

Ob-2: Least significant bits of Q^{Ref} are discarded in every loop iteration by 1 bit

(NOTE: Q^{Ref} is the variable Q of the reference design)

Ob-3: Least significant bits of $\_Q^{Uer}$ are discarded in every loop iteration by 1 bit
(NOTE: $\_Q^{User}$ is the variable Q of the user design and its association is not shown here for the ease of explanation)

Ob-4: $Q^{Ref}[2:0]$ *and* $\_Q^{User}[2:0]$ *have a value match and new value entered in MSB of both the variables are never used with loop*

The predicates corresponding to the observations are shown bellow.

Ob-1: (SHR($\_X^{Ref}$):1 $\wedge$ NotUsedWithinLoop($\_X^{Ref}$):(CP_2, CP_3))

Ob-2: (SHR(Q^{Ref}):1)

Ob-3: (SHR($\_Q^{Uer}$):1)

Ob-4: (ValueMatch($Q^{Ref}[2:0]$, $\_Q^{User}[2:0]$) $\wedge$ NotUsedWithinLoop($Q^{Ref}[3]$):(CP_2, CP_3) $\wedge$ NotUsedWithinLoop($\_Q^{User}[3]$):(CP_2', CP_3'))

With the help of these observations, the equivalence checker performs the analysis for the mismatches of $A\_X[2:0]$ and $A\_Q[2:0]$ and using **Ob-1** the mismatches of $A\_X[2:0]$ is ignored. Application of **Ob-2, Ob-3, Ob-4** yields that the bits $[2:0]$ of the variable A_Q i.e $\_Q^{User}$ matches with the bits $[2:0]$ of the variable Q^{Ref}. The bit $A\_Q[3]$ i.e. $\_Q^{User}[3]$ matches with the bit $A\_X[3]$ i.e. $\_X^{Ref}[3]$. Although there exist a mismatch between $A\_Q[3]$ and the $Q[3]$, they are never used in the loop. Therefore, $A\_X[2:0]$ and $A\_Q[2:0]$ mismatches are ignored by the equivalence checker and it then finds value match for all common variables including all the corresponding single and the composed variables. Finally, the two paths are marked as equivalent.

Finding data building pattern

After the paths are found to be equivalent, the equivalence checker tries to find the data building pattern using the design observations along with the observations generated during the equivalence establishment of the paths. For these paths a pattern can be identified using the following observations and it is to be noted that a variable with superscript Ref or $User$ denotes that the variable belongs to the reference or user design respectively.

Ob-5: Least significant bits of $\_X^{Ref}$ are discarded by 1 bit and a new value from another variable is entered in the MSB of $\_X^{Ref}$

Ob-6: Least significant bits of $\_Q^{User}$ are discarded in every loop iteration by 1 bit and New value is assigned in the MSB of Q^{User}

Ob-7: MSB of $\_X^{Ref}$ and MSB of $\_Q^{User}$ matches (after value match)

Ob-8: Mismatches of $\_X^{Ref}$ and $\_Q^{User}$ are ignored (after value match)

Ob-9: A^{Ref} and A^{User} value matches for all bits (after value match)

The predicates corresponding to these observations generated in due course of the equivalence checking are as follows.

Ob-5: (SHR($\_X^{Ref}$):1 $\wedge$ NewVal(MSB($\_X^{Ref}$)))

Ob-6: (SHR($\_Q^{User}$):1 $\wedge$ NewVal(MSB($\_Q^{User}$)))

Ob-7: ValueMatch(MSB($\_X^{Ref}$), MSB($\_Q^{User}$))

Ob-8: IgnoredMismatches($\_X^{Ref}$, $\_Q^{User}$)

Ob-9: ValueMatch($A^{Ref}[3:0]$, $A^{User}[3:0]$)

From these observations the following shift characteristic is recognized.

Shift characteristic: *In corresponding composed variables ($\{A^{Ref}, A\_X^{Ref}\}$, $\{A^{User}, A\_Q^{User}\}$), output is incrementally building by adding one bit from MSB of A_X^{Ref} or A_Q^{User} along with the four bits of A^{Ref} or A^{User} respectively*

Shift characteristic in the form of predicates: *ValueMatch((FullBit(A^{Ref}:4) $\wedge$ IncrementFromMsb(A_X^{Ref}:1)), (FullBit(A^{User}:4) $\wedge$ IncrementFromMsb(A_Q^{User}:1)))*

5.5.5 Equivalence analysis for path P_3 and P'_3

The paths P_3 and P'_3 along with their start vector and exit vector after symbolically executing those paths are shown in the Table 5.7.

Matching values

From the exit vectors of the reference and the user designs, it is observed that the values of the common single variables such as M, $count$ matches. The corresponding bits $A\_X[3]$ and $A\_Q[3]$ matches and there is a mismatch between the bits $A\_X[2:0]$ and $A\_Q[2:0]$ which is ignored through the analysis given in the previous subsection (5.5.4) for equivalence checking of the paths P_2 and P'_2. For the corresponding variable A^{Ref} and A^{User}, bits $A^{Ref}[2:0]$ and $A^{User}[2:0]$ match, however, $A^{Ref}[3]$ and $A^{User}[3]$ does not match. As $A[3]$ affects determining values in A in both the designs and among the mismatched values, one of the bit contains a symbol and the other one contain a specific value, then the mismatch is handled with a special analysis. If both the mismatched bits would hold mismatched symbols then this would imply that in general case the bits mismatched, therefore there is no need for special analysis and that would cause failure for the equivalence checking. However, in this example, the special analysis checks whether the bit having

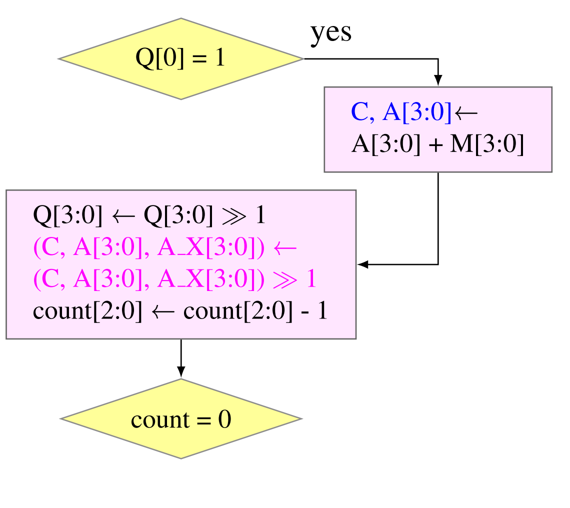
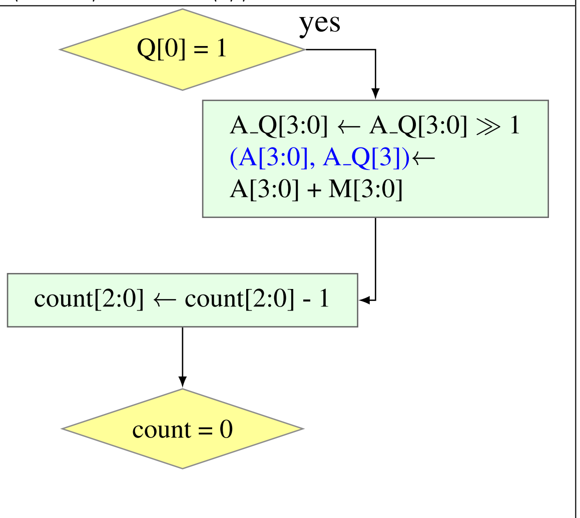
Reference Design	User Design
StartSVV: $\langle C : \langle c\rangle, A : \langle a_3a_2a_1a_0\rangle, A\_X : \langle x_3x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle q_3q_2q_1q_0\rangle, count : \langle t_2t_1t_0\rangle, prod : \langle\text{-}\rangle\rangle$	StartSVV: $\langle A : \langle a_3a_2a_1a_0\rangle, A\_Q : \langle q_3q_2q_1q_0\rangle, M : \langle m_3m_2m_1m_0\rangle, count : \langle t_2t_1t_0\rangle, prod : \langle\text{-}\rangle\rangle$
 Q[0] = 1 yes C, A[3:0]← A[3:0] + M[3:0] Q[3:0] ← Q[3:0] ≫ 1 (C, A[3:0], A_X[3:0]) ← (C, A[3:0], A_X[3:0]) ≫ 1 count[2:0] ← count[2:0] - 1 count = 0	 Q[0] = 1 yes A_Q[3:0] ← A_Q[3:0] ≫ 1 (A[3:0], A_Q[3])← A[3:0] + M[3:0] count[2:0] ← count[2:0] - 1 count = 0
ExitSVV: $\langle C : \langle 0\rangle, A : \langle s_4s_3s_2s_1\rangle, A\_X : \langle s_0x_2x_1x_0\rangle, M : \langle m_3m_2m_1m_0\rangle, Q : \langle 0q_3q_2q_1\rangle, count : \langle t'_2t'_1t'_0\rangle, prod : \langle\text{-}\rangle\rangle$	ExitSVV: $\langle A : \langle s_4s_3s_2s_1\rangle, A\_Q : \langle s_0q_3q_2q_1\rangle, M : \langle m_3m_2m_1m_0\rangle, count\langle t'_2t'_1t'_0\rangle, prod : \langle\text{-}\rangle\rangle$

Table 5.6: Equivalence checking of path P_2 and P'_2 (StartSVV: start symbolic value vector, ExitSVV: exit symbolic value vector)

a symbol actually takes the same specific value in all possible cases that of the other corresponding bit. In the exit vector of path P_3, $A^{Ref}[3]$ contains a symbol c, on the other hand, $A^{User}[3]$ has specific value *0* in the exit vector of the path P'_3. Therefore, it is checked whether c takes value *0* in all possible cases. For that, first the set of paths are computed from where path P_3 is reachable ($S^{P_3}_{Rto}$), which is $\{P_1, P_2P_4, P_3P_4\}$. Now, it is found that for each of the paths in the set $S^{P_3}_{Rto}$, the symbol c takes the specific value *0* after symbolic execution. Therefore, substituting the value of the symbol c in path P_3 for each case, $A\_X[3]$ yields the specific value *0* which is the same as $A\_Q[3]$ which resolves the mismatch. With this resolved mismatch the equivalence checker decides the paths P_3 and P'_3 to be equivalent.

Design observations and data building pattern

The design observations and the data building pattern identified for these paths are same as the paths P_2 and P'_2 which is described in the previous subsection 5.5.4.

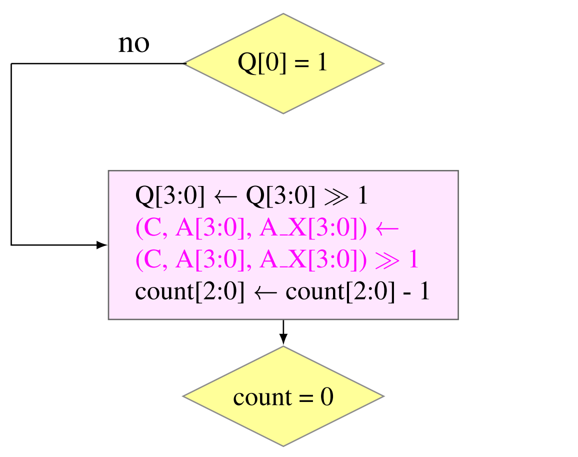
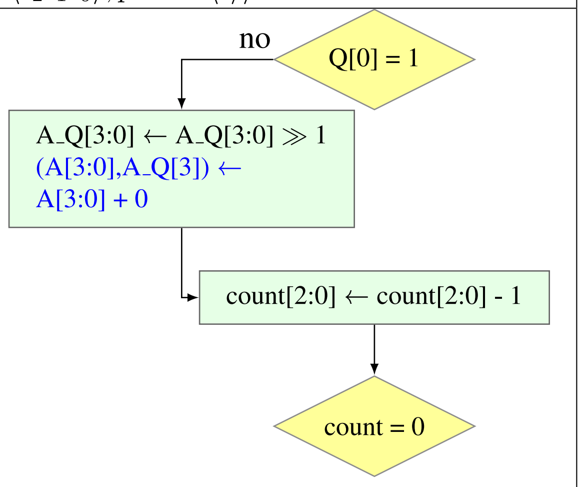
Reference Design	User Design
StartSVV: $\langle C:\langle c\rangle, A:\langle a_3a_2a_1a_0\rangle, A\_X:\langle x_3x_2x_1x_0\rangle, M:\langle m_3m_2m_1m_0\rangle, Q:\langle q_3q_2q_1q_0\rangle, count:\langle t_2t_1t_0\rangle, prod:\langle\text{-}\rangle\rangle$	StartSVV: $\langle A:\langle a_3a_2a_1a_0\rangle, A\_Q:\langle q_3q_2q_1q_0\rangle, M:\langle m_3m_2m_1m_0\rangle, count:\langle t_2t_1t_0\rangle, prod:\langle\text{-}\rangle\rangle$
 no Q[0] = 1 Q[3:0] ← Q[3:0] ≫ 1 (C, A[3:0], A_X[3:0]) ← (C, A[3:0], A_X[3:0]) ≫ 1 count[2:0] ← count[2:0] - 1 count = 0	 no Q[0] = 1 A_Q[3:0] ← A_Q[3:0] ≫ 1 (A[3:0],A_Q[3]) ← A[3:0] + 0 count[2:0] ← count[2:0] - 1 count = 0
ExitSVV: $\langle C:\langle 0\rangle, A:\langle \underline{c}a_3a_2a_1\rangle, A\_X:\langle a_0x_2x_1x_0\rangle, M:\langle m_3m_2m_1m_0\rangle, Q:\langle 0q_3q_2q_1\rangle, count:\langle t'_2t'_1t'_0\rangle, prod:\langle\text{-}\rangle\rangle$	ExitSVV: $\langle A:\langle \underline{0}a_3a_2a_1\rangle, A\_Q:\langle a_0q_3q_2q_1\rangle, M:\langle m_3m_2m_1m_0\rangle, count:\langle t'_2t'_1t'_0\rangle prod:\langle\text{-}\rangle\rangle$

Table 5.7: Equivalence checking of path P_3 and P'_3 (StartSVV: start symbolic value vector, ExitSVV: exit symbolic value vector)

5.5.6 Inferencing data range after loop completion

After the equivalence decision is made for all the paths from CP_2 and CP_3, the equivalence checker yields data building pattern for each path. Now, it is checked whether the identified pattern is same for all the paths.For this example an identical pattern is recognized which is as follows.

Shift characteristic

In corresponding composed variables ($\{A^{Ref}, A\_X^{Ref}\}$*,* $\{A^{User}, A\_Q^{User}\}$*), output is incrementally building by adding one bit from MSB of* A_X^{Ref} *or* A_Q^{User} *along with the four bits of* A^{Ref} *or* A^{User} *respectively*

Shift characteristic in the form of predicates

ValueMatch((FullBit(A^{Ref}*:4)* $\wedge$
IncrementFromMsb(A_X^{Ref}*:1)), (FullBit(*A^{User}*:4)* $\wedge$ *IncrementFromMsb(*A_Q^{User}*:1)))*

Using this pattern and the loop count which is four in this example, the equivalence checker now interpolate the data range residing in the composed variables ($\{A^{Ref}, A\_X^{Ref}\}$, $\{A^{User}, A\_Q^{User}\}$) after all the loop iterations.

Inferencing data range after n iterations
ValueMatch((FullBit(A^{Ref}:4) $\wedge$ MatchRange($A\_X^{Ref}[msb : msb - n + 1]$)), (FullBit($A^{User}$:4) $\wedge$ MatchRange($A\_Q^{User}[msb : msb - n + 1]$)))

Where msb denotes the most significant bit and lsb is the least significant bit of the variable.

Inferencing data range after four iterations
ValueMatch((FullBit(A^{Ref}:4) $\wedge$ MatchRange($A\_X^{Ref}[3 : 0]$)), (FullBit(A^{User}:4) $\wedge$ MatchRange($A\_Q^{User}[3 : 0]$)))

This implies that after four iterations of the loop,the variables $A[3 : 0]$ and $A\_X[3 : 0]$ in the reference design will match with the variables $A[3 : 0]$ and $A\_Q[3 : 0]$ in the user design.

Updated exit vector at cut-point $\mathbf{CP_3}$ and $\mathbf{CP'_3}$
After a successful interpolation, the exit vector is updated at the cut-point CP_3 and CP'_3 in the reference design and the user design respectively. Because as of now, those cut-points contain exit vectors where $A\_X[3]$ of the reference design and $A\_Q[3]$ of the user design matched and the rest of the bits mismatched. However, the interpolation indicates thats all of the will be matched after complete execution of the loop in any order. Therefore, all the bits of those variables in the exit vectors are replaced with the same symbols in both the designs. As the values for the two variables have been interpreted, values of the other values can not be determined after four iterations, therefore other variables contain null in the exit vectors. The updated exit vector at the cut-point CP_3 in the reference design as follows.

$\langle C : \langle - \rangle, A : \langle u_7 u_6 u_5 u_4 \rangle, A\_X : \langle u_3 u_2 u_1 u_0 \rangle, M : \langle - \rangle, Q : \langle - \rangle, count : \langle - \rangle, prod : \langle \text{-} \rangle \rangle$

The updated exit vector at the cut-point CP'_3 in the user design as follows.

$\langle A : \langle u_7 u_6 u_5 u_4 \rangle, A\_Q : \langle u_3 u_2 u_1 u_0 \rangle, M : \langle - \rangle, count : \langle - \rangle, prod : \langle \text{-} \rangle \rangle$

5.5.7 Equivalence analysis for path $\mathbf{P_5}$ and $\mathbf{P'_5}$

Table 5.8 shows the paths P_5 and P'_5 and the associated vectors. The updated exit vectors at the cut-points CP_3 and CP'_3 will be used as start vectors for the symbolic execution of the paths P_5 and P'_5 respectively yielding their corresponding exit vectors. The equivalence checker finds matching values between the common variables in the exit vectors of the two two paths and two paths are marked as equivalent.

Finally the reference design and the user design are found to be equivalent as all the paths of the path cover belonging to one design has its corresponding equivalent path in the another design.

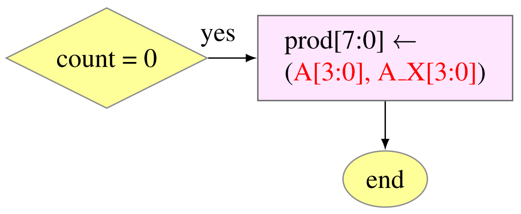
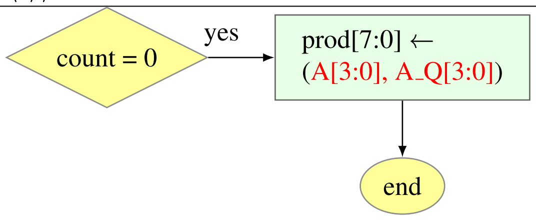
Reference Design	User Design
StartSVV: $\langle C : \langle - \rangle, A : \langle u_7u_6u_5u_4 \rangle, A\_X : \langle u_3u_2u_1u_0 \rangle, M : \langle - \rangle, Q : \langle - \rangle, count : \langle - \rangle, prod : \langle \text{-} \rangle \rangle$	StartSVV: $\langle A : \langle u_7u_6u_5u_4 \rangle, A\_Q : \langle u_3u_2u_1u_0 \rangle, M : \langle - \rangle, count : \langle - \rangle, prod : \langle \text{-} \rangle \rangle$
 count = 0 yes prod[7:0] ← (A[3:0], A_X[3:0]) end	 count = 0 yes prod[7:0] ← (A[3:0], A_Q[3:0]) end
ExitSVV: $\langle C : \langle - \rangle, A : \langle u_7u_6u_5u_4 \rangle, A\_X : \langle u_3u_2u_1u_0 \rangle, M : \langle - \rangle, Q : \langle - \rangle, count : \langle - \rangle, prod : \langle u_7u_6u_5u_4u_3u_2u_1u_0 \rangle \rangle$	ExitSVV: $\langle A : \langle u_7u_6u_5u_4 \rangle, A\_Q : \langle u_3u_2u_1u_0 \rangle, M : \langle - \rangle, count : \langle - \rangle, prod : \langle u_7u_6u_5u_4u_3u_2u_1u_0 \rangle \rangle$

Table 5.8: Equivalence checking of path P_5 and P_5' (StartSVV: start symbolic value vector, ExitSVV: exit symbolic value vector)

5.6 Implementation and results

The basic version of the bit-level equivalence checking is implemented in C which does not include the features of the method to handle special cases such as data interpolation currently. As the equivalence checking mechanism requires extensive symbolic analysis, bit-blasting technique is adopted using the canonical representation of expressions through the efficient BDD data structure. Colorado University Decision Diagram Package (CUDD) [7] is used to implement the bit-blasting through representing each bit as a BDD. One of the drawbacks of bit-blasting is that due to the representation of each bit as individual variable, a register or a bit-vector of size more than one losses its word-level abstraction. This situation is handled in this equivalence checker so that the word-level abstraction preserves and can be used by the equivalence checker when needed. One major limitation of the BDD based approach is its size with the increasing size of the data path. Several optimization can improve the efficiency so that larger data paths can be handled. One such optimization is adopted in this equivalence checker. Although the data transformations of a path are ultimately expressed in BDD form, the conditions of the path are not handled in the same manner but in a similar way it is handled in [10] where the variables are of infinite precision and restricted to the integer domain. Control flow behaviors are not expressed in BDDs instead they are stored as parse trees and in order to math the conditions the parse trees are matched. Few operations are also performed on those parse trees in order to find equivalence between them. The other optimizations can be a good future scope of this work.

The input FSMDs are given input in the form of a text file which is parsed by the lex yacc parser generators. The grammar rules are developed in such a way so that the equivalence checker can

Experiment name	**FSMD states**		**CPU Time (sec.)**			**Datapath operations**
	Ref.	**User**	**User**	**Real**	**Sys**	
GCD Processor 4-bit (GCD)	5	5	0.004	0.096	0.016	Const. assignment, reg to reg assignment, add, sub, compliment
Traffic Light Controller 4-bit (TLC)	13	13	0.016	0.449	0.060	Const. assignment, reg to reg assignment
BARCODE Reader 4-bit (BARCODE)	32	32	0.048	2.001	0.292	Const. assignment, reg to reg assignment, add
Sum of N Fibonacci Numbers 4-bit (FIBSUM)	5	5	0.008	0.118	0.024	Const. assignment, reg to reg assignment, add
Radix-4 Booth's Multiplier 4-bit (BOOTH)	5	5	0.004	0.171	0.024	Const. assignment, reg to reg assignment, add, sub, compliment, shift

Table 5.9: Verification results of equivalent test benches on the current implementation.

handle the following operations.

(*i*) Addition with constant, (*ii*) addition of variables, (*iii*) subtraction with constant operand and variable operand, (*iv*) complement, (*v*) concatenation, (*vi*) logical shift right, (*vii*) logical shift left, (*viii*) arithmetic shift right, (*ix*) arithmetic shift left, (*x*) different types of assignment including variable assignment, (*xi*) constant assignment,

Table 5.9 shows the CPU time during the execution of the equivalent test benches. The execution time is measured with the *time* command where *user* gives total CPU time in second that the process spent in user mode, *real* gives the elapsed real time between invocation and termination of the process and *sys* gives total CPU time in second that the process spent in kernel mode. The test benches correspond to the common experiments given in the laboratory course as assignments executed on the current implementation on a machine with the following configuration. Intel® Core™ i5-3210M CPU @ 2.50GHz×4 processor, 64-bit ubuntu 12.04 LTS operating system and 3.5 GiB memory. The test benches are BB-based i.e. the reference design and the user design behavior differs in basic block operations. Among the test benches, GCD, FIBSUM, BOOTH are arithmetic operation, logical operation intensive including shift operation. ON the other hand, the TLC, BARCODE they are control intensive with the heavy use of assignment operations. The operations which are handled are listed in the short form where reg. denotes register which is a variable, const. denotes constant value, add and sub denotes addition and subtraction operations respectively.

The Table 5.9 shows the verification results the test benches whose reference design and the corresponding user design are actually equivalent in spite of having structurally different but actually equivalent operations. However, the equivalence checker can also detect errors due to which equivalence checker fails and produce the result showing where the error is detected. In such case, before terminating, the equivalence checker shows the bit blasted registers i.e. variables so that it can be identified which corresponding bit(s) of the variable(s) mismatched.

The Tables 5.10, 5.11, 5.12, 5.13, 5.14 show the CPU time for the execution of the equivalence checker run with the test benches GCD, BOOTH, TLC, FIBSUM and BARCODE respectively with the user designs having non identical data transformations which causes the equivalence checker to fail. The execution time is measured with the *time* command where *user* gives total CPU time in second that the process spent in user mode, *real* gives the elapsed real time between invocation and termination of the process and *sys* gives total CPU time in second that the process spent in kernel mode. Four different user designs are generated through inducing different types of errors in the initial basic version of the user design which is being used in Table 5.9. Then each modified version of the user design with error introduced in them are checked for equivalence with their corresponding reference design. The machine configuration is same as mentioned aforesaid in this section. The errors mentioned in the table are self descriptive. Following are the brief description of some of the errors.

The initialization error is introduced in the common variable(s)

The initialization error is introduced in the common variable(s) (which is present in both the reference design and the user design) where the variable(s) in the paths emerging from the reset state or the start state are assigned with specific binary values.
Example: $A[3:0] = \{1,0,1,1\}$ and $A[3:0] = \{0,0,1,1\}$

Wrong variable assignment operation

The content of a mismatched variable is assigned to a common variable. This error may occur in any path.
Example: $A[3:0] = X[3:0]$ and $A[3:0] = Y[3:0]$

Arithmetic operation with wrong operator

The operands of an arithmetic operation remain similar including the variable to which the result is assigned, however, the operations become different. Note that in the Example2, due to the missing complement operator associated with $Y[3:0]$, the two operations become non-equivalent.
Example1: $A[3:0] = X[3:0] + Y[3:0]$ and $A[3:0] = X[3:0] - Y[3:0]$
Example2: $A[3:0] = X[3:0] - Y[3:0]$ and $A[3:0] = X[3:0] + (Y[3:0] + 1)$

Arithmetic operation with wrong operand

The arithmetic operator remains same, however, the operands are different although they are being

Sl no.	FSMD states		CPU Time (sec.)			Data transformation errors
	Ref.	User	User	Real	Sys	
1.	5	5	0.004	0.070	0.012	Initialization error in common variable
2.	5	5	0.000	0.045	0.012	Wrong variable assignment operation
3.	5	5	0.000	0.068	0.016	Arithmetic operation with wrong operator
4.	5	5	0.004	0.043	0.012	Arithmetic operation with wrong operand

Table 5.10: Failure of equivalence checker due to errors in GCD test bench.

assigned to the same common variable.
Example: $A[3:0] = X[3:0] + Y[3:0]$ and $A[3:0] = X[3:0] + Z[3:0]$

Wrong concatenation operation

The concatenation operation is used create composed variable. In this error the single variable which is used to create the composed variable differs in both the designs.
Example: $P[7:0] \leftarrow \{A[3:0], X[3:0]\}$ and $P[7:0] \leftarrow \{A[3:0], Q[3:0]\}$

Wrong shift operation

Different types of shift operations sometimes may introduce serious design error such as use of logical shift vs. arithmetic shift, right shift vs. left shift, etc.
Example1: $A[7:0] \leftarrow A[7:0] \gg 1$ and $A[7:0] \leftarrow A[7:0] \ggg 1$
Example2: $A[7:0] \leftarrow A[7:0] \gg 1$ and $A[7:0] \leftarrow A[7:0] \ll 1$

Some other errors include a path P^{Ref} in the reference design without any data transformation where as the user design contains data transformations in the path P^{User} with same condition. In such case, the path P^{Ref} of the reference design does not have any equivalent path in the user design.

Sl no.	**FSMD states**		**CPU Time (sec.)**			**Data transformation errors**
	Ref.	**User**	**User**	**Real**	**Sys**	
1.	5	5	0.000	0.101	0.020	Initialization error in common variable
2.	5	5	0.004	0.133	0.024	Wrong concatenation operation
3.	5	5	0.008	0.131	0.020	Absence of complement operator and wrong assignment
4.	5	5	0.004	0.141	0.024	Wrong shift operation

Table 5.11: Failure of equivalence checker due to errors in BOOTH test bench.

Sl no.	**FSMD states**		**CPU Time (sec.)**			**Data transformation errors**
	Ref.	**User**	**User**	**Real**	**Sys**	
1.	13	13	0.016	0.377	0.052	Initialization error in multiple common variable
2.	13	13	0.000	0.410	0.060	Wrong constant assignment operation
3.	13	13	0.004	0.358	0.044	Wrong constant and variable assignment operation
4.	13	13	0.012	0.413	0.052	Introduction of non-equivalent uncommon arithmetic operation

Table 5.12: Failure of equivalence checker due to errors in TLC test bench.

Sl no.	**FSMD states**		**CPU Time (sec.)**			**Data transformation errors**
	Ref.	**User**	**User**	**Real**	**Sys**	
1.	5	5	0.004	0.082	0.024	Initialization error in common variable
2.	5	5	0.004	0.117	0.024	Wrong variable assignment operation
3.	5	5	0.004	0.134	0.028	Arithmetic operation with wrong operator
4.	5	5	0.008	0.111	0.016	Arithmetic operation with wrong operand

Table 5.13: Failure of equivalence checker due to errors in FIBSUM test bench.

Sl no.	**FSMD states**		**CPU Time (sec.)**			**Data transformation errors**
	Ref.	**User**	**User**	**Real**	**Sys**	
1.	32	32	0.012	0.692	0.076	Initialization error in multiple common variables
2.	32	32	0.012	0.713	0.084	Wrong constant assignment and uncommon variable initialization
3.	32	32	0.020	0.886	0.100	Introduction of non-equivalent shift operation
4.	32	32	0.036	0.908	0.120	Introduction of non-equivalent data transformation

Table 5.14: Failure of equivalence checker due to errors in BARCODE test bench.

Chapter 6

Conclusion

In this work a virtual laboratory has been developed to support teaching of logic design and computer organization along with a newly developed tool which has a graphical interface at its front end to supports designing of experiments and an efficient logic simulator to simulate the experiments built so far. A bit-level equivalence checker has also been developed for automatic evaluation of design correctness.

6.1 Summary of work done

This work presents some techniques and algorithms to support teaching of logic design and computer organization through developing a web based virtual laboratory (COLDVL) and a formal verification method of bit-level equivalence checking for automatic evaluation of student designs. The virtual laboratory contains a newly developed the COLDVL tool equipped with a circuit drawing and experimentation interface as a front end, a logic simulator as the back end with features to provide real laboratory like learning experience and a set of pre-designed guided experiments with the facility to add new experiments. The COLDVL tool provides a generic simulation platform for performing experimentation pertaining to logic design and computer organization. Some pedagogic considerations are assimilated in the design of the COLDVL through designing the web interface of the virtual laboratory, a set of pre-designed experiments based on a concept hierarchy (derived from various text books), a sequence of learning activities based on some commonly accepted pedagogic principles. Satisfactory user feedback has been gathered and analyzed through deploying the COLDVL in several colleges and institutes. COLDVL, having much more useful features and facilities than the virtual lab presented in the literature survey of this thesis, is also been used to conduct laboratory courses in undergraduate and postgraduate level laboratory course at IIT Kharagpur. In addition with the experimentation in a laboratory course, the evaluation of student designs is also an important aspect to track the effective learning of the students. The manual evaluation is man power intensive and its availability is limited through out the year. On the other hand, simulation based evaluations are some what restricted in uncovering various cases. These limitations of the manual and simulation based evaluation leads to an automatic evaluation technique using formal methods.

Front end of the COLDVL tool

The COLDVL tool is capable of simulating both combinational and sequential circuits. The front end of the tool contains several building blocks and reference designs for each experiment (with opaque internal details), input-output components, several combinational components (various gates, multiplexers, adders, etc), sequential components including major types of flip flops (both behavioral and structural) and other components such as RAMs with editable cells and an controller chart based control unit generator. The generated control unit can be used in association of a data path to build circuits to implement simple and as well as complex computer arithmetic algorithms such as multiplication, division etc. The tool also has facilities for generating the structural verilog netlist of user designed circuits which can be independently used with other electronic CAD tools, creating and reusing encapsulated user modules and saving circuits with user identification. A five valued logic is employed in the tool to support tri-state logic, wired AND operation for a bus based design and to set unknown value as default initial value for all logic signals indicating an unknown logic value of either 0 or 1.

Back end of the COLDVL tool

For better simulation performance, the back end of the tool, the logic simulator uses an efficient simulation technique for the sequence circuits conforming to the Huffman model which is a combination of topologically ordered levelized simulation (which is also known as levelized simulation in the literature) and event driven simulation. Levelized simulation is used for combinational circuits and the standard event driven simulation is used for sequential circuits which do not conform to the Huffman model. In order to use the combined simulation technique, a partition technique (in a gate level circuit) along with a Huffman structure detection technique has been developed. The simulator detects the existence of possible race around condition in a circuit prior to simulation. These features of the simulator help students learning a safe circuit design. Due to the use of unknown initial value for a logic signal, the standard event driven simulation technique may not produce desirable definite output for a given determinate input in some structural gate level storage elements which would give determinate output when performed in bread boards. For a novice student this discrepancy is circuit outputs may appear as confusing, therefore, to handle the situation, the simulation technique performs a case based analysis of unknown values of some nets of the loops in the gate level memory elements to determine whether or not a definite output value is assumed irrespective of the cases considered. The simulator also incorporates an optional feature of obtaining a workable initialization by way of simulation while reconstructing a circuit in an ordered manner for the structural memory elements where the case analysis mechanism fails to resolve initial unknown uncertainty.

Bit-level equivalence checking method

For evaluation of student designs, the design correctness is checked between the reference designs provided by the course instructor and the implemented student design i.e. user design. The proposed path based bit-level equivalence checking is performed on the two designs whose behaviors are modeled as finite state machines with data path (FSMD). An FSMD models the condition of execution and the data transformations of a circuit. As the data path of a circuit is finite, the data path elements are modeled as bit-vectors and a reasoning over the bit-vectors are used in order to handle the arithmetic, logical, bit wise and shift operations performed on them. The reasoning over bit-vectors are accomplished with the help of bit-blasting technique where each bit of a bit-vector is represented as a variable. In this work we represent each bit as binary decision diagram (BDD) which transforms all the expressions in their canonical form. Apart from handling the several types of data transformation operations, the proposed method can also handle a special situation where different shift patterns produces identical outputs after a finite iteration and the output may reside in multiple bit-vectors. Several approaches has been introduced in the current bit-level equivalence checking method in order to handle the situation such as generation of design observations in the form of predicates, determination of loop invariant shift pattern using the design observations and some predefined rules, interpolation of the bigger data ranges using loop invariant data building pattern.

6.2 Future scope of this work

The overall design and development of the COLDVL package along with the design of the front end and the back end of its simulator have been carried out in order to support the learning experience achieved in a real laboratory like virtual environment. This work can be further extended as follows.

- A new feature can be added in the COLDVL tool which will extract the FSMD model from a user circuit.
- Integration of the COLDVL tool with the bit-level equivalence checker where the course guide will feed the FSMD corresponding to the reference design and the COLDVL tool will input the extracted FSMD from the user design i.e. the designs built by the students. Finally the equivalence checker will deliver the evaluation results for the design correctness.
- The simulation can be made interactive which will allow users to pause simulation and resume later.
- An optimized approach can be adopted to build user defined encapsulated modules to support larger hierarchical designs instead of the current approach.
- Currently the feature for bundle connection, rotation of a component, the graphical editor with multiple tab are not supported in the front end of the COLDVL tool which may be included in future.

- A technique can also be integrated with the COLDVL to run the COLDVL tool on the server side for better accessibility through ubiquitous but relatively less capable mobile computing platforms

The current path based bit-level equivalence checker uses BDD based approach. The data transformations of a path are expressed as BDDs. The BDD based approach has some limitations for larger data paths with size more than 16 bits. Following are some future scope of the current bit-level equivalence checker.

- Currently all the data transformations of a computational path are represented as BDDs prior to the execution of the equivalence checker. This approach can be further optimized by building BDDs for the data transformations which are effectively required for the computation of the path in order to handle larger data paths efficiently.
- A naming convention is used to identify the variable composition which may be determined automatically through a backward analysis.

Appendix A

FSMDs of test bench designs

A.1 Equivalent designs

Following are the FSMDs of the test benches presented in the Table 5.9.

GCD reference FSMD

```
 "GcdBasicRef.org"
q00 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0} q01;
q01 2 X[3:0]>0 | - q02 X[3:0]<=0 | Z[3:0]=Y[3:0] q0e;
q02 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q03 X[3:0]>Y[3:0] | - q03;
q03 1 - | X[3:0]=X[3:0]-Y[3:0] q01;
q0e 0;\\
```

GCD user FSMD

```
 "GcdBasicUser.org"
q10 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0} q11;
q11 2 X[3:0]>0 | - q12 X[3:0]<=0 | Z[3:0]=Y[3:0] q1e;
q12 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q13 X[3:0]>Y[3:0] | - q13;
q13 1 - | X[3:0]=(~Y[2:0]+1)+X[2:0] q11;
q1e 0;
```

TLC reference FSMD

```
 "TrafficLightControllerRef.org"
q000 2 currentState[3:0]==0 | newHL[3:0]={0,1,0,0},newFL[3:0]={0,1,1,0} q001 currentState[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
        | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 currentState[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]={0,1,1,0} q003 currentState[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 currentState[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 currentState[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,0,0,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 currentState[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 currentState[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 currentState[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 currentState[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0 ;
```

TLC user FSMD

```
 "TrafficLightControllerUser.org"
q000 2 currentState[3:0]==0 | newHL[3:0]={0,1,0,0},newFL[3:0]={0,1,1,0} q001 currentState[3:0]!=0 | − q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
      | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 currentState[3:0]==4 | newHL[3:0]=2,newFL[3:0]={0,1,1,0} q003 currentState[3:0]!=4 | − q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
     newST[3:0]={0,0,0,0} q004;
q004 2 currentState[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 currentState[3:0]!=2 | − q008;
q005 4 timeoutL[3:0]==1 | − q006 cars[3:0]==0 | − q006 timeoutL[3:0]!=1 | − q007 cars[3:0]!=0 | − q007;
q006 1 − | newstate[3:0]={0,1,1,0},newST[3:0]={0,0,0,1} q008;
q007 1 − | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 currentState[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 currentState[3:0]!=6 | − q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
     newST[3:0]={0,0,0,0} q010;
q010 2 currentState[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
     q011 currentState[3:0]!=7 | − q011;
q011 1 − | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0;
```

BARCODE reference FSMD

```
 "BARCODEref.org"
q000 1 − | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={0,0,0,0},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
     [3:0]={0,0,0,0},eop[3:0]={0,0,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | − q031 ;
q002 2 breakflag[3:0]!=1 | − q003 breakflag[3:0]==1 | − q006 ;
q003 2 clk[3:0]!=1 | − q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | − q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | − q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | − q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | −
      q001 ;
q008 2 breakflag[3:0]!=1 | − q009 breakflag[3:0]!=1 | − q012 ;
q009 2 clk[3:0]!=1 | − q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | − q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | − q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
     [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | − q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | − q014 white[3:0]!=255 | − q014 black[3:0]!=255&&breakflag[3:0]==0 | − q026
     white[3:0]!=255 | − q026 ;
q014 2 video[3:0]==wh[3:0] | − q015 video[3:0]!=wh[3:0] | − q021 ;
q015 2 clk[3:0]!=1 | − q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | − q017 ;
q017 1 − | white[3:0]=white[3:0]+1 q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0]+1,memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
     q019 ;
q019 1 − | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | − q013 ;
q021 2 clk[3:0]!=1 | − q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | − q023 ;
q023 1 − | black[3:0]=black[3:0]+1 q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0]+1,memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
     q025 ;
q025 1 − | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | − q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | − q028 start[3:0]!=0&&breakflag[3:0]==0 | − q007 ;
q028 2 clk[3:0]!=1 | − q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | − q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | − q027 ;
q031 0 ;
```

BARCODE user FSMD

```
 "BARCODEuser.org"
q000 1 − | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={0,0,0,0},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
     [3:0]={0,0,0,0},eop[3:0]={0,0,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | − q031 ;
q002 2 breakflag[3:0]!=1 | − q003 breakflag[3:0]==1 | − q006 ;
q003 2 clk[3:0]!=1 | − q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | − q005 ;
```

```
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | - q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | - q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | -
     q001 ;
q008 2 breakflag[3:0]!=1 | - q009 breakflag[3:0]!=1 | - q012 ;
q009 2 clk[3:0]!=1 | - q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | - q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | - q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
     [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | - q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | - q014 white[3:0]!=255 | - q014 black[3:0]!=255&&breakflag[3:0]==0 | - q026
     white[3:0]!=255 | - q026 ;
q014 2 video[3:0]==wh[3:0] | - q015 video[3:0]!=wh[3:0] | - q021 ;
q015 2 clk[3:0]!=1 | - q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | - q017 ;
q017 1 - | white[3:0]=white[3:0]+1 q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0]+1,memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
     q019 ;
q019 1 - | black[3:0]={0,0,0,0},data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0],flag[3:0]=wh[3:0] q013 eop[3:0]!=0 | flag[3:0]=wh[3:0] q013 ;
q021 2 clk[3:0]!=1 | - q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | - q023 ;
q023 1 - | - q024 ;
q024 2 flag[3:0]==wh[3:0] | black[3:0]=black[3:0]+1,actnum[3:0]=actnum[3:0]+1,memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh
     [3:0] | black[3:0]=black[3:0]+1,memw[3:0]={0,0,0,1} q025 ;
q025 1 - | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | - q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | - q028 start[3:0]!=0&&breakflag[3:0]==0 | - q007 ;
q028 2 clk[3:0]!=1 | - q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | - q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | - q027 ;
q031 0 ;
```

FIBSUM reference FSMD

```
 "FibBasicRef.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q01;
q0e 0;
```

FIBSUM user FSMD

```
 "FibBasicUser1.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0] q04;
q04 1 - | Y[3:0]=Z[3:0] q01;
q0e 0;
```

BOOTH reference FSMD

```
 "BoothBasicRef1.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0},X[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=A[3:0]-M[3:0] q03 Q[0]==1&&Qprev
     [0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1,Cnt[2:0]=Cnt[2:0]-1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],X[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

BOOTH user FSMD

```
 "BoothBasicUser1.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=(~M[2:0]+1)+A[2:0] q03 Q[0]==1&&
     Qprev[0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1,Cnt[2:0]=Cnt[2:0]-1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],X[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

A.2 Non-equivalent designs for test bench GCD

Following are the FSMDs of the test benches presented in the Table 5.10 related to test bench GCD. As the correctness of the user design is to be matched with the reference design, different types of errors are introduced in the user design and the reference design remains same.

GCD reference FSMD (Sl no. 1), (Sl no. 2), (Sl no. 3) and (Sl no. 4)

```
 "GcdBasicRef.org"
q00 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0} q01;
q01 2 X[3:0]>0 | - q02 X[3:0]<=0 | Z[3:0]=Y[3:0] q0e;
q02 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q03 X[3:0]>Y[3:0] | - q03;
q03 1 - | X[3:0]=X[3:0]-Y[3:0] q01;
q0e 0;
```

GCD user FSMD (Sl no. 1)

```
 "GcdBasicUserC1.org"
q10 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,1} q11;
q11 2 X[3:0]>0 | - q12 X[3:0]<=0 | Z[3:0]=Y[3:0] q1e;
q12 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q13 X[3:0]>Y[3:0] | - q13;
q13 1 - | X[3:0]=(~Y[2:0]+1)+X[2:0] q11;
q1e 0;
```

GCD user FSMD (Sl no. 2)

```
 "GcdBasicUserC4.org"
q10 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,1} q11;
q11 2 X[3:0]>0 | - q12 X[3:0]<=0 | Z[3:0]=Y[3:0] q1e;
q12 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q13 X[3:0]>Y[3:0] | - q13;
q13 1 - | X[3:0]=(~Y[2:0]+1)+Y[2:0] q11;
q1e 0;
```

GCD user FSMD (Sl no. 3)

```
 "GcdBasicUserC3.org"
q10 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,1} q11;
q11 2 X[3:0]>0 | - q12 X[3:0]<=0 | Z[3:0]=Y[3:0] q1e;
q12 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q13 X[3:0]>Y[3:0] | - q13;
q13 1 - | X[3:0]=(Y[2:0]+1)+X[2:0] q11;
q1e 0;
```

GCD user FSMD (Sl no. 4)

```
 "GcdBasicUserC4.org"
q10 1 - | X[3:0]={0,0,0,0},Y[3:0]={0,0,0,1} q11;
q11 2 X[3:0]>0 | - q12 X[3:0]<=0 | Z[3:0]=Y[3:0] q1e;
q12 2 X[3:0]<=Y[3:0] | tmp[3:0]=Y[3:0],Y[3:0]=X[3:0],X[3:0]=tmp[3:0] q13 X[3:0]>Y[3:0] | - q13;
q13 1 - | X[3:0]=(~Y[2:0]+1)+Y[2:0] q11;
q1e 0;
```

A.3 Non-equivalent designs for test bench BOOTH

Following are the FSMDs of the test benches presented in the Table 5.11 related to test bench BOOTH. As the correctness of the user design is to be matched with the reference design, different types of errors are introduced in the user design and the reference design remains same.

BOOTH reference FSMD (Sl no. 1), (Sl no. 2), (Sl no. 3) and (Sl no. 4)

```
 "BoothBasicRef.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0},X[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=A[3:0]-M[3:0] q03 Q[0]==1&&Qprev
      [0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1,Cnt[2:0]=Cnt[2:0]-1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],X[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

BOOTH user FSMD (Sl no. 1)

```
 "BoothBasicUserC1.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,1,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=(~M[2:0]+1)+A[2:0] q03 Q[0]==1&&
      Qprev[0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],X[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

BOOTH user FSMD (Sl no. 2)

```
 "BoothBasicUserC2.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=(~M[2:0]+1)+A[2:0] q03 Q[0]==1&&
      Qprev[0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],Q[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

BOOTH user FSMD (Sl no. 3)

```
 "BoothBasicUserC3.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=(M[2:0]+1)+A[2:0] q03 Q[0]==1&&Qprev
      [0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],Q[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

BOOTH user FSMD (Sl no. 4)

```
 "BoothBasicUserC5.org"
q00 1 - | Qprev[0:0]={0},A[3:0]={0,0,0,0},cnt[2:0]={1,0,0},M[3:0]={0,0,0,0},Q[3:0]={0,0,0,0} q01;
q01 4 Q[0]==0&&Qprev[0]==1 | A[3:0]=A[3:0]+M[3:0] q03 Q[0]==1&&Qprev[0]==0 | A[3:0]=(~M[2:0]+1)+A[2:0] q03 Q[0]==1&&
      Qprev[0]==1 | - q03 Q[0]==0&&Qprev[0]==0 | - q03;
q03 1 - | {A[3:0],X[3:0]}={A[3:0],X[3:0]}>>>1,{Q[3:0],Qprev[0:0]}={Q[3:0],Qprev[0:0]}>>>1 q04;
q04 2 Cnt[2:0]==0 | prod[7:0]={A[3:0],X[3:0]} q0e Cnt[2:0]!=0 | - q01 ;
q0e 0;
```

A.4 Non-equivalent designs for test bench TLC

Following are the FSMDs of the test benches presented in the Table 5.12 related to test bench TLC. As the correctness of the user design is to be matched with the reference design, different types of errors are introduced in the user design and the reference design remains same.

TLC reference FSMD (Sl no. 1), (Sl no. 2), (Sl no. 3) and (Sl no. 4)

```
 "TrafficLightControllerRef.org"
q000 2 current_state[3:0]==0 | newHL[3:0]={0,1,0,0},newFL[3:0]={0,1,1,0} q001 current_state[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
       | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 current_state[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]={0,1,1,0} q003 current_state[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 current_state[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 current_state[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,0,0,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 current_state[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 current_state[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 current_state[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 current_state[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0 ;
```

TLC user FSMD (Sl no. 1)

```
 "TrafficLightControllerUserC1.org"
q000 2 current_state[3:0]==0 | newHL[3:0]={0,1,1,1},newFL[3:0]={1,1,1,0} q001 current_state[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
       | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 current_state[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]={0,1,1,0} q003 current_state[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 current_state[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 current_state[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,0,0,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 current_state[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 current_state[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 current_state[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 current_state[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0;
```

TLC user FSMD (Sl no. 2)

```
 "TrafficLightControllerUserC2.org"
q000 2 current_state[3:0]==0 | newHL[3:0]={0,1,0,0},newFL[3:0]={0,1,1,0} q001 current_state[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
       | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 current_state[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]={0,1,1,0} q003 current_state[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 current_state[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 current_state[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,1,1,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 current_state[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 current_state[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 current_state[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 current_state[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0;
```

TLC user FSMD (Sl no. 3)

```
 "TrafficLightControllerUserC3.org"
q000 2 current_state[3:0]==0 | newHL[3:0]={0,1,0,1},newFL[3:0]=newHL[3:0] q001 current_state[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
       | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
```

```
q002 2 current_state[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]={0,1,1,0} q003 current_state[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 current_state[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 current_state[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,1,1,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 current_state[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]=newstate[3:0] q009 current_state[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 current_state[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 current_state[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0;
```

TLC user FSMD (Sl no. 4)

```
 "TrafficLightControllerUserC4.org"
q000 2 current_state[3:0]==0 | newHL[3:0]={0,1,0,0},newFL[3:0]={0,1,1,0} q001 current_state[3:0]!=0 | - q002;
q001 2 timeoutL[3:0]==1&&cars[3:0]==1 | newstate[3:0]={0,1,0,0},newST[3:0]={0,0,0,1} q002 timeoutL[3:0]!=1&&cars[3:0]==1
       | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0} q002;
q002 2 current_state[3:0]==4 | newHL[3:0]={0,0,1,0},newFL[3:0]=newHL[3:0]+1 q003 current_state[3:0]!=4 | - q004;
q003 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,1} q004 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q004;
q004 2 current_state[3:0]==2 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,1,0,0} q005 current_state[3:0]!=2 | - q008;
q005 4 timeoutL[3:0]==1 | - q006 cars[3:0]==0 | - q006 timeoutL[3:0]!=1 | - q007 cars[3:0]!=0 | - q007;
q006 1 - | newstate[3:0]={0,1,1,0},newST[3:0]={0,0,0,1} q008;
q007 1 - | newstate[3:0]={0,0,1,0},newST[3:0]={0,0,0,0} q008;
q008 2 current_state[3:0]==6 | newHL[3:0]={0,1,1,0},newFL[3:0]={0,0,1,0} q009 current_state[3:0]!=6 | - q010;
q009 2 timeoutS[3:0]==1 | newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,1} q010 timeoutS[3:0]!=1 | newstate[3:0]={0,1,1,0},
      newST[3:0]={0,0,0,0} q010;
q010 2 current_state[3:0]==7 | newHL[3:0]={0,0,0,0},newFL[3:0]={0,0,0,0},newstate[3:0]={0,0,0,0},newST[3:0]={0,0,0,0}
      q011 current_state[3:0]!=7 | - q011;
q011 1 - | state[3:0]=newstate[3:0],Hiway[3:0]=newHL[3:0],FarmL[3:0]=newST[3:0],startTimer[3:0]=newST[3:0] q0e;
q0e 0;
```

A.5 Non-equivalent designs for test bench FIBSUM

Following are the FSMDs of the test benches presented in the Table 5.13 related to test bench FIBSUM. As the correctness of the user design is to be matched with the reference design, different types of errors are introduced in the user design and the reference design remains same.

FIBSUM reference FSMD (Sl no. 1), (Sl no. 2), (Sl no. 3) and (Sl no. 4)

```
 "FibBasicRef.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q01;
q0e 0;
```

FIBSUM user FSMD (Sl no. 1)

```
 "FibBasicUserC1.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={1,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q0e;
q0e 0;
```

FIBSUM user FSMD (Sl no. 2)

```
 "FibBasicUserC2.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=X[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q0e;
q0e 0;
```

FIBSUM user FSMD (Sl no. 3)

```
 "FibBasicUserC3.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]-X[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q0e;
q0e 0;
```

FIBSUM user FSMD (Sl no. 4)

```
 "FibBasicUserC4.org"
q00 1 - | num[3:0]={0,1,1,0},X[3:0]={0,0,0,0},Y[3:0]={0,0,0,0},sum[3:0]={0,0,0,1} q01;
q01 2 num[3:0]>=X[3:0] | Z[3:0]=X[3:0]+Y[3:0] q02 num[3:0]<X[3:0] | out[3:0]=sum[3:0] q0e;
q02 1 - | sum[3:0]=sum[3:0]+Y[3:0] q03;
q03 1 - | X[3:0]=Y[3:0],Y[3:0]=Z[3:0] q0e;
q0e 0;
```

A.6 Non-equivalent designs for test bench BARCODE

Following are the FSMDs of the test benches presented in the Table 5.14 related to test bench BARCODE. As the correctness of the user design is to be matched with the reference design, different types of errors are introduced in the user design and the reference design remains same.

BARCODE reference FSMD (Sl no. 1), (Sl no. 2), (Sl no. 3) and (Sl no. 4)

```
 "BarcodeReaderRef.org"
q000 1 - | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={0,0,0,0},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
     [3:0]={0,0,0,0},eop[3:0]={0,0,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | - q031 ;
q002 2 breakflag[3:0]!=1 | - q003 breakflag[3:0]==1 | - q006 ;
q003 2 clk[3:0]!=1 | - q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | - q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | - q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | - q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | -
      q001 ;
q008 2 breakflag[3:0]!=1 | - q009 breakflag[3:0]!=1 | - q012 ;
q009 2 clk[3:0]!=1 | - q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | - q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | - q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
     [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | - q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | - q014 white[3:0]!=255 | - q014 black[3:0]!=255&&breakflag[3:0]==0 | - q026
     white[3:0]!=255 | - q026 ;
q014 2 video[3:0]==wh[3:0] | - q015 video[3:0]!=wh[3:0] | - q021 ;
q015 2 clk[3:0]!=1 | - q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | - q017 ;
q017 1 - | white[3:0]=white[3:0] q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
     q019 ;
q019 1 - | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | - q013 ;
```

```
q021 2 clk[3:0]!=1 | – q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | – q023 ;
q023 1 – | black[3:0]=black[3:0] q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
      q025 ;
q025 1 – | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | – q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | – q028 start[3:0]!=0&&breakflag[3:0]==0 | – q007 ;
q028 2 clk[3:0]!=1 | – q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | – q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | – q027 ;
q031 0 ;
```

BARCODE user FSMD (Sl no. 1)

```
 "BarcodeReaderUserC1.org"
q000 1 – | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={1,1,0,1},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
      [3:0]={0,0,0,0},eop[3:0]={0,0,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | – q031 ;
q002 2 breakflag[3:0]!=1 | – q003 breakflag[3:0]==1 | – q006 ;
q003 2 clk[3:0]!=1 | – q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | – q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | – q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | – q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | –
      q001 ;
q008 2 breakflag[3:0]!=1 | – q009 breakflag[3:0]!=1 | – q012 ;
q009 2 clk[3:0]!=1 | – q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | – q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | – q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
      [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | – q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | – q014 white[3:0]!=255 | – q014 black[3:0]!=255&&breakflag[3:0]==0 | – q026
      white[3:0]!=255 | – q026 ;
q014 2 video[3:0]==wh[3:0] | – q015 video[3:0]!=wh[3:0] | – q021 ;
q015 2 clk[3:0]!=1 | – q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | – q017 ;
q017 1 – | white[3:0]=white[3:0] q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
      q019 ;
q019 1 – | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | – q013 ;
q021 2 clk[3:0]!=1 | – q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | – q023 ;
q023 1 – | black[3:0]=black[3:0] q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
      q025 ;
q025 1 – | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | – q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | – q028 start[3:0]!=0&&breakflag[3:0]==0 | – q007 ;
q028 2 clk[3:0]!=1 | – q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | – q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | – q027 ;
q031 0 ;
```

BARCODE user FSMD (Sl no. 2)

```
 "BarcodeReaderUserC2.org"
q000 1 – | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={1,1,0,1},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
      [3:0]={0,0,0,0},eop[3:0]={0,0,0,0}, uncommon1[3:0]={1,1,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | – q031 ;
q002 2 breakflag[3:0]!=1 | – q003 breakflag[3:0]==1 | – q006 ;
q003 2 clk[3:0]!=1 | – q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | – q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | – q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | – q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | –
      q001 ;
q008 2 breakflag[3:0]!=1 | – q009 breakflag[3:0]!=1 | – q012 ;
q009 2 clk[3:0]!=1 | – q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | – q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | – q008 ;
```

```
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
      [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | - q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | - q014 white[3:0]!=255 | - q014 black[3:0]!=255&&breakflag[3:0]==0 | - q026
      white[3:0]!=255 | - q026 ;
q014 2 video[3:0]==wh[3:0] | - q015 video[3:0]!=wh[3:0] | - q021 ;
q015 2 clk[3:0]!=1 | - q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,1,1} q017 reset[3:0]!=1 | - q017 ;
q017 1 - | white[3:0]=white[3:0] q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
      q019 ;
q019 1 - | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | - q013 ;
q021 2 clk[3:0]!=1 | - q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | - q023 ;
q023 1 - | black[3:0]=black[3:0] q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
      q025 ;
q025 1 - | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | - q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | - q028 start[3:0]!=0&&breakflag[3:0]==0 | - q007 ;
q028 2 clk[3:0]!=1 | - q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | - q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | - q027 ;
q031 0 ;
```

BARCODE user FSMD (Sl no. 3)

```
 "BarcodeReaderUserC3.org"
q000 1 - | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={0,0,0,0},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
      [3:0]={0,0,0,0},eop[3:0]={0,0,0,0},uncommon1[3:0]={1,1,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | - q031 ;
q002 2 breakflag[3:0]!=1 | - q003 breakflag[3:0]==1 | - q006 ;
q003 2 clk[3:0]!=1 | - q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | - q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | - q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | - q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | -
       q001 ;
q008 2 breakflag[3:0]!=1 | - q009 breakflag[3:0]!=1 | - q012 ;
q009 2 clk[3:0]!=1 | - q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | - q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | - q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
      [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | - q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | - q014 white[3:0]!=255 | - q014 black[3:0]!=255&&breakflag[3:0]==0 | - q026
      white[3:0]!=255 | - q026 ;
q014 2 video[3:0]==wh[3:0] | - q015 video[3:0]!=wh[3:0] | - q021 ;
q015 2 clk[3:0]!=1 | - q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | - q017 ;
q017 1 - | white[3:0]=white[3:0] q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
      q019 ;
q019 1 - | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | - q013 ;
q021 2 clk[3:0]!=1 | - q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | - q023 ;
q023 1 - | black[3:0]=black[3:0] q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
      q025 ;
q025 1 - | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0]>>1 q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | - q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | - q028 start[3:0]!=0&&breakflag[3:0]==0 | - q007 ;
q028 2 clk[3:0]!=1 | - q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | - q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | - q027 ;
q031 0 ;
```

BARCODE user FSMD (Sl no. 4)

```
 "BarcodeReaderUserC4.org"
q000 1 - | wh[3:0]={0,0,0,0},bl[3:0]={0,0,0,1},eoc[3:0]={0,0,0,0},memw[3:0]={0,0,0,0},data[3:0]={0,0,0,0},addr
      [3:0]={0,0,0,0},eop[3:0]={0,0,0,0} q001 ;
q001 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q002 eop[3:0]!=0 | - q031 ;
```

```
q002 2 breakflag[3:0]!=1 | - q003 breakflag[3:0]==1 | - q006 ;
q003 2 clk[3:0]!=1 | - q003 clk[3:0]==1 | eop[3:0]={0,0,0,0} q004 ;
q004 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q005 reset[3:0]!=1 | - q005 ;
q005 2 start[3:0]==1 | breakflag[3:0]={0,0,0,1} q002 start[3:0]!=1 | - q002 ;
q006 2 eop[3:0]==0 | breakflag[3:0]={0,0,0,0} q007 eop[3:0]!=0 | - q001 ;
q007 2 actnum[3:0]!=num[3:0]&&white[3:0]!=255 | breakflag[3:0]={0,0,0,0} q008 actnum[3:0]==num[3:0]&&white[3:0]==255 | -
       q001 ;
q008 2 breakflag[3:0]!=1 | - q009 breakflag[3:0]!=1 | - q012 ;
q009 2 clk[3:0]!=1 | - q009 clk[3:0]!=1 | eop[3:0]={0,0,0,0} q010 ;
q010 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q011 reset[3:0]!=1 | - q011 ;
q011 2 scan[3:0]==1 | breakflag[3:0]={0,0,0,1} q008 scan[3:0]!=1 | - q008 ;
q012 2 eop[3:0]==0 | flag[3:0]=wh[3:0],actnum[3:0]={0,0,0,0},white[3:0]={0,0,0,0},black[3:0]={0,0,0,0},eoc
      [3:0]={0,0,0,0},breakflag[3:0]={0,0,0,0} q013 eop[3:0]!=0 | - q007 ;
q013 4 black[3:0]!=255&&breakflag[3:0]==0 | - q014 white[3:0]!=255 | - q014 black[3:0]!=255&&breakflag[3:0]==0 | - q026
      white[3:0]!=255 | - q026 ;
q014 2 video[3:0]==wh[3:0] | - q015 video[3:0]!=wh[3:0] | - q021 ;
q015 2 clk[3:0]!=1 | - q015 clk[3:0]==1 | eop[3:0]={0,0,0,0} q016 ;
q016 2 reset[3:0]==1 | eop[3:0]=1,breakflag[3:0]={0,0,0,1} q017 reset[3:0]!=1 | - q017 ;
q017 1 - | white[3:0]=white[3:0] q018 ;
q018 2 flag[3:0]==bl[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q019 flag[3:0]!=bl[3:0] | memw[3:0]={0,0,0,1}
      q019 ;
q019 1 - | black[3:0]={0,0,0,0},flag[3:0]=wh[3:0],data[3:0]=white[3:0] q020 ;
q020 2 eop[3:0]==0 | addr[3:0]=actnum[3:0] q013 eop[3:0]!=0 | - q013 ;
q021 2 clk[3:0]!=1 | - q021 clk[3:0]==1 | eop[3:0]={0,0,0,0} q022 ;
q022 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q023 reset[3:0]!=1 | - q023 ;
q023 1 - | black[3:0]=black[3:0] q024 ;
q024 2 flag[3:0]==wh[3:0] | actnum[3:0]=actnum[3:0],memw[3:0]={0,0,0,0} q025 flag[3:0]!=wh[3:0] | memw[3:0]={0,0,0,1}
      q025 ;
q025 1 - | flag[3:0]=bl[3:0],white[3:0]={0,0,0,0},data[3:0]=black[3:0] q020 ;
q026 2 eop[3:0]==0 | memw[3:0]={0,0,0,0},eoc[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,0} q027 eop[3:0]!=0 | - q007 ;
q027 2 start[3:0]!=0&&breakflag[3:0]==0 | - q028 start[3:0]!=0&&breakflag[3:0]==0 | - q007 ;
q028 2 clk[3:0]!=1 | - q028 clk[3:0]==1 | eop[3:0]={0,0,0,0} q029 ;
q029 2 reset[3:0]==1 | eop[3:0]={0,0,0,1},breakflag[3:0]={0,0,0,1} q030 reset[3:0]!=1 | - q030 ;
q030 2 start[3:0]==0 | breakflag[3:0]={0,0,0,1} q027 start[3:0]!=0 | breakflag[3:0]={1,1,1,1} q027 ;
q031 0 ;
```

Appendix B

Case studies of deployment

B.1 Deployment of COLDVL

B.1.1 Semester laboratory course

COLDVL has been used for conducting laboratory sessions in the first year M.Tech course ‘Computing systems lab (IT69101)’ for laboratory experimentation in Autumn session, 2012, 2013, and 2014. Students were given assignments on logic design and computer organization, after building their circuit and saving with their identification they uploaded the file to the *Web-Based Course Management system* [36] of IIT Kharagpur over Internet, for manual evaluation. Following are some assignments statements which were accomplished by the students.

1. Implementation, simulation and analysis of adders

```
You are required to implement, simulate and analyse the following two types of adders.

    1. Ripple carry adder (RCA), for four bits
    2. Carry lookahead adder (CLA), for four bits, generating carry generate and carry propagate functions.
    3. Block carry lookahead adder (BCLA), to add two 16-bit numbers, using 4-bit CLAs and 4-bit BCLA units

You should do a comparison of these two types of adders for cost and speed.

You are advised to use the COA virtual lab facilty at the SIT web site for doing this experiment.

After completing the experiment, you should save the design in a "logic" file for subsequent submission. You will also
       have to create a report in plain text.

The design of the 16-bit BCLA is to be done hierarchically. The CLA and BCLA units (and possible the full adder unit)
       are to be designed as components and also saved as "logic" files.
Submissible items

    1. Logic file for 4-bit RCA
    2. Logic file for 4-bit CLA
    3. Logic file for 4-bit BCLA unit
    4. Overall 16-bit BCLA

Marking guidelines

Assignment marking is to be done only after the deadline expires, as submissions gets blocked after the assignment is
       marked.

Correct working of RCA building blocks  5
Overall correct operation of RCA, including generation of carryout and overflow indication       5
Correct working of CLA, with carry generate and carry propagate functions       5
Correct working of BCLA unit, with carry bit generation, carry generate and carry propagate functions     5
Overall correct operation of BCLA, including generation of carryout and overflow indication      5
Cost and speed analysis of the two adders       5
Determination of cost and speed of RCA  5
Determination of cost and speed of CLA  5
Determination of cost and speed of BCLA unit     5
Determination of overall cost and speed of BCLA 5
Total Marks     50
Assignment submission
```

Use electronic submission via the WBCM link

You should keep submitting your incomplete assignment from time to time after making some progress, as you can submit any number of times before the deadline expires.

2. Implementation, simulation and analysis of multipliers

You are required to implement, simulate and analyse the shift and add multiplier, doing the implementation for eight bits 2's complement numbers. Ensure that the sign necessary for arithmetic shifting is properly generated using a suitable sign generation logic.

You are permitted to use the ALU, multiplexers, registers and other discrete components.

Your design should be modular and should have two top-level components, viz samDP and samCtrl. Ensure that your modules are well labelled (pins should be labelled to indicate their functionality).

The data path should have all the data processing elements, appropriate data path control signal inputs and appropriate status signal outputs.

The controller should be designed as a finite state machine, using the status signals of the data path as its inputs and generating outputs which are used to drive the data path control signals.

You are required to document the operation of the multiplier.

You are advised to use the COA virtual lab facilty at the SIT web site for doing this experiment. Make sure you enter your name and roll number in the design you create.

After completing the experiment, you should save the design in a "logic" file for subsequent submission. You will also have to create a report in plain text.
Marking guidelines

Assignment marking is to be done only after the deadline expires, as submissions gets blocked after the assignment is marked.

Proper operation and documentation of SAM datapath modules	10
Proper operation and documentation of SAM controller	10
Proper operation and documentation of SAM datapaths	10
Discussion on the multiplication of negative numbers using SAM	5
Overall documentation and description of the design	5
Total Marks	40

Assignment submission

Use electronic submission via the WBCM link

Definitely upload the top-level design and the detailed report (in plain text). Upload as many of the other components as possi ble. Ask you TA to conduct an evaluation in the lab, apart from the usual online evaluation.

You should keep submitting your incomplete assignment from time to time after making some progress, as you can submit any number of times before the deadline expires.

3. Restoring and non-restoring division

You are required to implement, simulate and analyse the following two types of dividers, doing the implementation for five bits of divisor and ten bits of dividend.

Restoring divider (RDiv)
Non-restoring divider (NRDiv)

You should do a comparison of these two types of dividers for cost and speed. You should also discuss division methods for signed operands.

You are advised to use the COA virtual lab facilty at the SIT web site for doing this experiment. Make sure you enter your name and roll number in the design you create.

After completing the experiment, you should save the design in a "logic" file for subsequent submission. You will also have to create a report in plain text.
Marking guidelines

Assignment marking is to be done only after the deadline expires, as submissions gets blocked after the assignment is marked.

Proper operation and documentation of RDiv datapaths	10

Proper operation and documentation of RDiv controller 10
Proper operation and documentation of NRDiv datapaths 10
Proper operation and documentation of NRDiv controller 10
Discussion on the division of negative numbers using 10
Total Marks 50
Assignment submission

Submissible items:

1. Logic file of restoring division divider (logic.gz)
2. Logic file of non−restoring division divider (logic.gz)
3. Report on the dividers covering items of discussion

Use electronic submission via the WBCM link

You should keep submitting your incomplete assignment from time to time after making some progress, as you can submit any number of times before the deadline expires.

4. Simulation of Caches

The proportion of accesses that result in a cache hit is known as the hit rate, and can be a measure of the effectiveness of the cache for a given program or algorithm.

Memory access of your program is modelled assuming that your program has:

a set of 16 variables which are located at consecutive locations and accessed at random.
an array of 256 elements which are accessed in an arithmetic progressing of the index values with the step varying between 1 to 4.

When non−trivial replacement of a cache item is needed, assumed that the least recently accessed entry is replaced.

You are required to model memory accesses and simulate the performance of the following caches where a cache line has l words.

1. Direct mapped cache with N lines
2. Fully associative cache with N lines
3. K−way set associative cache with M sets, so that KM=N

It is common for the parameters used above to be powers of 2.

You should simulate the hit rate for various chosen values of the cache parameters and identify optimal values of those parameters for the given program model. These results should be suitably depicted using simple graphs (avoid creating large files). In your report, you should do a cost/performance comparison of these caches.

This experiment has to be done by way of writing a program to carry out the simulation of the cache.

You need to submit your program file, the graphs and a report, as a text file.
Marking guidelines

Assignment marking is to be done only after the deadline expires, as submissions gets blocked after the assignment is marked.

Proper modelling of memory access of the program 10
Proper modelling of a direct mapped cache 5
Proper modelling of a fully associative cache 10
Proper modelling of a K−way associative cache 10
Proper modelling of the replacement policy and idenitfication of its applicability 10
Identifying optimal values of the cache parameters for the direct mapped cache 5
Identifying optimal values of the cache parameters for the fully associative cache 5
Identifying optimal values of the cache parameters for the K−way associative cache 5
Discussion on the cost/performance analysis 5
Total Marks 65
Assignment submission

Use electronic submission via the WBCM link

You should keep submitting your incomplete assignment from time to time after making some progress, as you can submit any number of times before the deadline expires.

B.1.2 Workshops

Workshops having duration of three hours were organized in engineering colleges of three different universities of West Bengal, India, which includes Jadavpur University, Bengal Engineering and Science University and West Bengal University of Technology. In the workshops, a brief demonstration of the virtual laboratory (COLDVL) was given. After that participants were asked to perform experiments followed by a feedback session. Workshops were organized for both faculties and students.

Workshop at IIT Kharagpur
A workshop was conducted in association with a short term teacher training course, 25-30 June, 2012, at IIT Kharagpur for the AICTE faculty members where 33 faculty members from different engineering colleges all over India had participated.

Workshop at colleges under West Bengal University of Technology (WBUT)
CIT and MCKV are the two AICTE approved engineering colleges under WBUT. A workshop had been organized in CIT and MCKV on 25th February and 27th February of 2013 respectively for the students from Information Technology, Computer Science, Electronics, Electrical branch as all of them have digital logic and computer organization as there core courses. In CIT,there were total 38 participants and among them 26 submitted their feedback in hard copy and rest of them submitted online. From the feedback, 8 feedback were canceled for those who had not paid sufficient attention during the workshop. In the workshop of the MCKV college, the number of participants was 34 among which 25 valid feedback were obtained.

Workshop at Jadavpur University
On 1st April, 2013 a workshop was conducted in the Jadavpur University. 26 students from the computer science department participated and 23 valid feedback were collected.

Workshop at Bengal Engineering and Science University
On 12th April, 2013 a workshop was conducted in the Bengal Engineering and Science University. 12 students from the computer science department participated and no invalid feedback were gathered.

B.1.3 Independent participation

COLDVL virtual laboratory was used and evaluated independently (without the benefit of workshop with live demonstration) at NIT Jalandhar, early in 2013. This participation is of special significance as students are expected to be able to use the COLDVL to conduct experiments, as it is available in the Internet without any special guidance from the developers. The major motivation was to know the feedback of students who were not given practical demonstration through workshops. In a span of few days, 46 feedback were submitted online along with the saved circuits performed by the students.

B.2 Feedback questions for COLDVL

Following is the feedback questions used to evaluate COLDVL through workshops and independent participation.

```
1. Name of the Experiment:

Ripple Carry Adder
Carry-look-ahead adder
Synthesis of flip-flops
Registers and Counters
Wallace Tree Adder
Combinational Multipliers
Arithmatic Logic Unit
Memory Design
Direct Mapped cache Design
Associative cache
CPU design
Other:

2. How do you rate the online performance of the experiment?
Excellent
Very Good
Good
Average
Poor

3. Was the procedure and manual found to be helpful?
Yes
No
Improvement Required

4. To what degree was the actual lab environment simulated?
Complete
Partial
Improvement Required

5. To what extent did you have control over the interactions?
Excellent
Very Good
Good
Average
Poor

6. Rate the quality of graphics of the simulator
Excellent
Very Good
Good
Average
Poor

7. Do you have a clear understanding of the experiment and related topics?
Yes
```

No
Partial

8. Did you experience any problems? (Yes/No) If ”Yes” give the reasons

9. How much do you like the way of analyzing results and the value propagation in your circuit
through different wire colors?
Excellent
Very Good
Good
Average
Poor

10. Rate the usability of the interface of the simulator
Excelent
Very Good
Good
Average
Poor

11. Is there anything (specific/generic) feedback with respect to the experiments that you would
like to share?

12. Do you think doing experiments through virtual lab gives scope for more innovative and
creative work?
Yes
No
To some extenct

13. Please suggest some points to improve the virtual lab(if any)

14. Do you find this simulator and the associated experiments will motivate students for
self−learning?
Yes
To Some Extent
No

15. Do you think that this kind of virtual lab with experiments and theory will really enhance
student learning?

16. From a teacher’s perspective do you feel that this kind of e−learning along with teaching and
real lab experiments will inspire students for future research?

Bibliography

[1] Hase, hierarchical computer architecture design and simulation environment, school of informatics, university of edinburgh,. http://www.icsa.inf.ed.ac.uk/research/groups/hase/.

[2] Jhdl overview brigham young univ. http://www.jhdl.org/overview.html.

[3] Syllabus for all courses at CSE, IIT Kharagpur. http://cse.iitkgp.ac.in/.

[4] Architexa, 2010. http://www.architexa.com/labs/.

[5] Eclipse gef framework, 2010. http://www.eclipse.org/gef/.

[6] Ims-ld learning design specifications, 2010. http://www.imsglobal.org.

[7] Cudd: Cu decision diagram package, 2012. http://vlsi.colorado.edu/ fabio/CUDD/.

[8] Aura Poulsen, Khoa Lam, Sarah Cisneros, and Torrey Trust. Arcs model of motivational design. EDTEC 544.

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[10] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. A value propagation based equivalence checking method for verification of code motion techniques. In *Electronic System Design (ISED), 2012 International Symposium on*, pages 67–71, 2012.

[11] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. Verification of code motion techniques using value propagation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(8):1180–1193, Aug 2014.

[12] J. Bransford, National Research Council (U.S.). Committee on Developments in the Science of Learning, National Research Council (U.S.). Committee on Learning Research, and Educational Practice. *How people learn: brain, mind, experience, and school*. National Academy Press, 2000.

[13] M. Breuer and A. Friedman. *Diagnosis and reliable design of digital systems*. Digital system design series. Computer Science Press, 1976.

[14] S. Britain. A review of learning design: Concepts, specifications and tools, May. A Report for JISC E-Learning Pedagogy Programme.

[15] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[16] C. Burch. Logisim: A graphical system for logic circuit design and simulation. *J. Educ. Resour. Comput.*, 2(1):5–16, Mar. 2002.

[17] J. Byrne, C. Heavey, and P. Byrne. A review of web-based simulation and supporting tools. *Simulation Modelling Practice and Theory*, 18(3):253 – 276, 2010.

[18] Duschl, R.A., H. Schweingruber, A. Shouse, K. t. E. G. Committee on Science Learning, National Research Council (U.S.). Board on Science Education, and National Research Council (U.S.). *Taking Science to School: Learning and Teaching Science in Grades K-8*. National Academies Press, 2007.

[19] R. W. Floyd. Assigning meaning to programs. In *Proceedings the 19th Symposium on Applied Mathematics*, pages 19–32, 1967.

[20] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[21] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Des. Test*, 11(4):44–54, Oct. 1994.

[22] S. Grossberg. The link between brain learning, attention, and consciousness. *Consciousness and Cognition*, 8(1):1 – 44, 1999.

[23] S. Grossberg. Chapter 107 - linking attention to learning, expectation, competition, and consciousness. In L. Itti, G. Rees, and J. K. Tsotsos, editors, *Neurobiology of Attention*, pages 652 – 662. Academic Press, Burlington, 2005.

[24] M. H. Gunes, M. A. Thornton, F. Kocan, and S. A. Szygenda. A survey and comparison of digital logic simulators. In *48th Midwest Symposium on Circuits and Systems*, pages 744–749 Vol. 1, 2005.

[25] D. Hammer and A. Elby. Tapping Epistemological Resources for Learning Physics. *The Journal of the Learning Sciences*, 12(1), 2003.

[26] J. P. Hayes. *Computer Architecture and Organization; (2Nd Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1988.

[27] W. Houghton. *Engineering Subject Centre Guide : learning and teaching theory for engineering academics*. Higher Education Academy Engineering Subject Centre, Loughborough University, 2004.

[28] C. Huandong, W. Shulei, S. Chunhui, and C. Mingrui. Research on the learning theory of e-learning. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 1185–1187, 2009.

[29] D. A. Huffman. The Design and Use of Hazard-[f]ree Switching Networks. *J. ACM*, 4(1):47–62, Jan. 1957.

[30] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[31] C. M. Jenkins, A. D. Voss, and D. Furcy. An effective educational module for booth's multiplication algorithm. *J. Comput. Sci. Coll.*, 27(4):54–62, Apr. 2012.

[32] C. Karfa, C. Mandal, and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30, 2012.

[33] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(3):556–569, 2008.

[34] J. M. Keller. Development and use of the arcs model of motivational design. *Journal of Instructional Development*, 10(3), 1987.

[35] K. Kotovsky, J. Hayes, and H. Simon. Why are some problems hard? evidence from tower of hanoi. *Cognitive Psychology*, 17(2):248 – 294, 1985.

[36] C. Mandal, V. L. Sinha, and C. M. P. Reade. Web-based course management and web services. *Electronic Journal on e-Learning*, 2(1):135–144, Feb. 2004.

[37] Z. Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974.

[38] M. M. Mano. *Digital Logic and Computer Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1979.

[39] M. M. Mano. *Computer System Architecture (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[40] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic. A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *Education, IEEE Transactions on*, 52(4):449 –458, nov. 2009.

[41] L. Pascual, A. Torrentí, J. Sahuquillo, and J. Flich. Understanding cache hierarchy interactions with a program-driven simulator. In *Proceedings of the 2007 workshop on Computer architecture education*, WCAE '07, pages 30–35, New York, NY, USA, 2007. ACM.

[42] D. A. Poplawski. A pedagogically targeted logic design and simulation tool. In *Proceedings of the 2007 Workshop on Computer Architecture Education*, WCAE '07, pages 1–7, New York, NY, USA, 2007. ACM.

[43] M. I. Posner. Cumulative development of attentional theory. *American Psychologist*, 37:168–179, 1982.

[44] Z. Radivojevic, M. Cvetanovic, and J. Dordevic. Design of the simulator for teaching computer architecture and organization. In *Engineering of Computer Based Systems (ECBS-EERC), 2011 2nd Eastern European Regional Conference on the*, pages 124 –130, sept. 2011.

[45] D. Sarkar and S. C. De Sarkar. A set of inference rules for quantified formula handling and array handling in verification of programs over integers. *IEEE Trans. Softw. Eng.*, 15(11):1368–1381, Nov. 1989.

[46] D. Sarkar and S. C. De Sarkar. Some inference rules for integer arithmetic for verification of flowchart programs on integers. *IEEE Trans. Softw. Eng.*, 15(1):1–9, Jan. 1989.

[47] W. Schneider and R. M. Shiffrin. Controlled and automatic human information processing: I. detection, search, and attention. *Psychological Review*, 84(1):1–66, 1977.

[48] C. Schunn and E. Silk. Learning theories for engineering and technology education. In M. Barak and M. Hacker, editors, *Fostering Human Development Through Engineering and Technology Education*, volume 6 of *International Technology Education Studies*, pages 3–18. SensePublishers, 2011.

[49] M. Serra, E. Wang, and J. Muzio. A multimedia virtual lab for digital logic design. In *Microelectronic Systems Education, 1999. MSE'99. IEEE International Conference on*, pages 39–40, 1999.

[50] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of polynomial datapaths using ideal membership testing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1320–1330, July 2007.

[51] R. M. Shiffrin and W. Schneider. Controlled and automatic human information processing: Ii. perceptual learning, automatic attending and a general theory. *Psychological Review*, 84(2):127–190, 1977.

[52] W. Stallings. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[53] W. Stallings. *Computer Organization and Architecture - Designing for Performance (7. ed.)*. Prentice Hall, 2006.

[54] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(5):586–597, May 2004.

[55] J. Sweller. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, 4(4):295 – 312, 1994.

[56] B. Talon, M. Sagar, and C. Kolski. Developing competence in interactive systems: The grasp tool for the design or redesign of pedagogical ict devices. *Trans. Comput. Educ.*, 12(3):9:1–9:43, 2012.

[57] The Joint Task Force on Computing Curricula, I. Computer Society, and Association for Computing Machinery. Computing curricula 2001, computer science. Final Report.

[58] The Joint Task Force on Computing Curricula, I. Computer Society, and Association for Computing Machinery. Curriculum guidelines for undergraduate degree programs in computer engineering. A Report in the Computing Curricula Series.

[59] P. A. von Kaenel. Designing and testing a control unit. *J. Comput. Sci. Coll.*, 19(5):228–237, May 2004.

[60] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. SSIM: a software levelized compiled-code simulator. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pages 2–8, New York, NY, USA, 1987. ACM.

[61] C. Wickens and J. McCarley. *Applied Attention Theory*. CRC Press, 2008.

[62] C. D. Wickens. Multiple resources and mental workload. *Human Factors*, 50(3):449–455, 2008.