

T.R.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING

**C TO ALGORITHMIC STATE MACHINE
CONVERTER 2 (CONTINUATION PROJECT)**

MUHAMMET FIKRET ATAR

**SUPERVISOR
DR. ALP ARSLAN BAYRAKCI**

**GEBZE
2022**

T.R.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

**C TO ALGORITHMIC STATE MACHINE
CONVERTER 2 (CONTINUATION
PROJECT)**

MUHAMMET FIKRET ATAR

SUPERVISOR
DR. ALP ARSLAN BAYRAKCI

2022
GEBZE

 <p>GEBZE TECHNICAL UNIVERSITY</p>	<p>GRADUATION PROJECT JURY APPROVAL FORM</p>
--	--

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 03/03/2022 by the following jury.

JURY

Member

(Supervisor) : Dr. Alp Arslan BAYRAKCI

Member : Prof. Dr. Hasari Çelebi

ABSTRACT

In this project, it is aimed to create a tool in the QT environment. It converts C code into algorithmic state machine and also displays the resulting ASM and converts this ASM to Verilog code. The tool also provides an environment where the user can create his own state diagram. In addition, there is an environment where you can create a state diagram from scratch, and the diagram you create can be converted to verilog code without c code. The state diagram is designed to be zoomable and editable. The tool aims to get the closest results to a hand drawn Asm and executable verilog code.

The focus of the project is the world of embedded and real-time systems. Verilog and state machine are indispensable parts of this world. However, manually designed state diagrams require a long process. The project plans to eliminate these difficulties. Projects designed simply in C will be instantly transformed into state diagram and verilog code. Thanks to this project, it will make the use of Asm and Verilog more effective in the world of embedded and real-time systems.

ÖZET

Bu projede QT ortamında bir araç oluşturulması hedeflenmiştir. C kodunu algoritmik durum makinesine dönüştürür ve ayrıca elde edilen ASM'yi gösterir ve bu ASM'yi Verilog koduna dönüştürür. Araç, kullanıcının kendi durum diyagramını oluşturabileceği bir ortam da sağlar. Ayrıca sıfırdan bir durum diyagramı oluşturabileceğiniz bir ortam bulunmakta , c codu olmadan da oluşturduğunuz diagram verilog koduna dönüştürülebilmektedir. Durum diyagramı yakınlaştırılabilir ve düzenlenebilir şekilde dizayn edildi. Araç, elle çizilmiş bir Asm ve yürütülebilir verilog koduna en yakın sonuçları almayı amaçlar.

Projenin odak noktası, gömülü ve gerçek zamanlı sistemler dünyasıdır. Verilog ve durum makinesi bu dünyanın vazgeçilmez parçalarıdır. Ancak manuel olarak tasarlanmış durum diyagramları uzun bir süreç gerektirir. Proje bu zorlukları ortadan kaldırmayı planlıyor. Basitçe C ile tasarlanan projeler, anında durum diyagramına ve verilog koduna dönüştürülecektir. Bu proje sayesinde gömülü ve gerçek zamanlı sistemler dünyasında Asm ve Verilog kullanımı daha efektif hale gelecektir.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to everyone who gave me the opportunity to complete this report. Special thanks go to my graduation project supervisor, Dr.Alp Arslan BAYRAKCI, who contributed to encourage and encourage suggestions and especially helped me coordinate my project while writing this report. I would also like to thank Prof. Dr. Hasari Çelebi, a member of the jury. I would also like to express my sincere respect and thanks to Gebze Technical University for supporting this study. In addition, I would like to express my respect and love to my family who supported me in every way throughout my education life, and to all my teachers who set an example for me with their lives.

Muhammet Fikret ATAR

LIST OF SYMBOLS AND ABBREVIATIONS

Symbol or

Abbreviation : Explanation

ABC : The first three letters in the English alphabet

μ : Small mu

ASM : Algorithmic State Machine

HDL : Hardware Description Language

GCD : Greatest common divisor

HCF : Highest common factor

CONTENTS

Abstract	iv
Özet	v
Acknowledgement	vi
List of Symbols and Abbreviations	vii
Contents	xi
List of Figures	xiii

1 Project General Information	1
1.1 Project Scheme and Description	1
1.2 Literature Research	2
1.2.1 C Programming Language	2
1.2.2 Verilog	2
1.2.3 What is the Difference Between Verilog and C	2
1.2.3.1 File Extensions	2
1.2.3.2 Usage	2
1.2.3.3 Conclusion	3
1.2.4 Algorithmic State Machine	3
1.2.4.1 State	3
1.2.4.2 Decision	4
1.2.4.3 Conditional Output	4
1.2.4.4 Datapath Derivation	5
1.2.4.5 Control Unit Derivation	5
1.2.4.6 Example	5
1.3 Project Design Plan	6
1.3.0.1 C Code	6
1.3.0.2 Parser	7
1.3.0.3 ASM	7
1.3.0.4 Verilog Code	7
1.3.0.5 Design plan in qt	7
1.4 Project Goal	8

1.5	Project Requirement and Libraries	8
1.5.1	Project Requirement	8
1.5.2	Libraries	9
1.6	Timeline of Project	9
2	Implementation	11
2.1	Design Gui	11
2.1.1	Qt Code Classes	11
2.1.1.1	Class Fsd	11
2.1.1.2	Class JsonToVlog	11
2.1.1.3	Class Mainwindow	11
2.1.1.4	Class Properties	12
2.1.1.5	Class State	12
2.1.1.6	Class State Diagram	12
2.1.1.7	Class Transition	12
2.2	Analyze C Code	12
2.2.1	Analysis Functions	13
2.2.1.1	FindInt	13
2.2.1.2	WithoutBracket	13
2.2.1.3	IsHaveBracket	13
2.2.2	Convert Verilog	14
2.2.2.1	ConvertAndPrint	14
2.2.2.2	Convert()	14
2.3	Draw State Diagram	14
2.3.1	Draw Functions	14
2.3.1.1	Draw Line	14
2.3.1.2	Draw line arrow	15
2.3.1.3	Draw Text	15
2.3.1.4	Draw State Num	15
2.3.1.5	Rectangel Print	15
2.3.1.6	Calculate Points	15
3	Qt Creator	16
3.0.1	Qt Creator	16
3.0.2	C to Asm and Verilog Editor	17
3.0.2.1	Documentation of Editing	17
3.0.2.1.1	Add a state	17
3.0.2.1.2	Add a transition	17
3.0.2.1.3	Add a self transition	18
3.0.2.1.4	Add an initial transition	18

3.0.2.1.5	Delete a state or a transition	18
3.0.2.1.6	Move a state	18
3.0.2.1.7	Edit a state or a transition	19
3.0.2.2	Saving and loading	19
3.0.2.3	C to Verilog	19
4	Test and Result	21
4.0.1	To find the factorial of a number	21
4.0.1.1	C code	21
4.0.1.2	State Diagram	22
4.0.1.3	Verilog Code	23
4.0.1.4	Running the verilog code generated by the tool . . .	24
4.0.2	Greatest Common Divisor	25
4.0.2.1	C code	25
4.0.2.2	State Diagram	26
4.0.2.3	Verilog Code	27
4.0.2.4	Running the verilog code generated by the tool . . .	28
4.0.3	Largest element in array	28
4.0.3.1	C code	29
4.0.3.2	State Diagram	30
4.0.3.3	Verilog Code	31
4.0.3.4	Running the verilog code generated by the tool . . .	31
4.0.4	Power	32
4.0.4.1	C code	32
4.0.4.2	State Diagram	33
4.0.4.3	Verilog Code	34
4.0.4.4	Running the verilog code generated by the tool . . .	35
4.0.5	IsPrime	36
4.0.5.1	C code	36
4.0.5.2	State Diagram	37
4.0.5.3	Verilog Code	38
4.0.5.4	Running the verilog code generated by the tool . . .	39
4.0.6	GUI Test	41
4.0.6.1	Open C Code	42
4.0.6.2	State Diagram in GUI	43
4.0.6.3	Verilog Code in GUI	44
4.0.6.4	Edit State Diagram in GUI	45
	Restrictions	46

Work Done By Other People	47
Success Criteria	48
Conclusions	49
Bibliography	50

LIST OF FIGURES

1.1	Scheme of Project	1
1.2	State	3
1.3	State	4
1.4	State	4
1.5	State	6
1.6	Design Plan	6
1.7	C to Json format	8
1.8	Timeline of Project	10
1.9	Plan	10
3.1	Qt Cpp Icon	16
3.2	Qt Cpp GUI	17
3.3	Add State Icon	17
3.4	Add a transition Icon	18
3.5	Add a self transition Icon	18
3.6	Add an initial transition Icon	18
3.7	Delete a state or a transition Icon	18
3.8	Move a state Icon	19
3.9	Save and Open Icon	19
3.10	C To Verilog Action Icon	20
4.1	Factorial C Code	21
4.2	State diagram for fact created by the tool	22
4.3	Fact verilog code created by the tool	23
4.4	Verilog code simulation	24
4.5	GCD c code	25
4.6	GCD state diagram created by the tool	26
4.7	GCD verilog code created by the tool	27
4.8	GCD test code	28
4.9	GCD test result	28
4.10	Largest element C Code	29
4.11	State diagram for largest created by the tool	30
4.12	Verilog code for largest element created by the tool	31
4.13	Verilog code simulation	31
4.14	Power C Code	32

4.15	State diagram for power created by the tool	33
4.16	Verilog code for power created by the tool	34
4.17	Verilog code for power simulation	35
4.18	Is Prime C Code	36
4.19	State diagram for isprime created by the tool	37
4.20	Verilog code for isprime created by the tool	38
4.21	If number is not prime verilog code simulation	39
4.22	If number is prime verilog code simulation	40
4.23	GUI displaying	41
4.24	Open C code file	42
4.25	Display state diagram	43
4.26	Create Verilog Code	44
4.27	Edit State Diagram (Delete Some State)	45
4.28	Sample code	46

1. PROJECT GENERAL INFORMATION

The project facilitates the world of the embedded and real-time system by taking advantage of the similarities between c code and Verilog. You will be able to do long-term and tiring projects in a much simpler and more comfortable way with the language closest to the machine language, c.

1.1. Project Scheme and Description

If we are to schematize the project, the code analysis is completed, and then the state diagram with the obtained tokens appears as close to hand drawn. With the diagram data formed after this part, the Verilog code becomes the closest to working in a way that acts like a state machine.

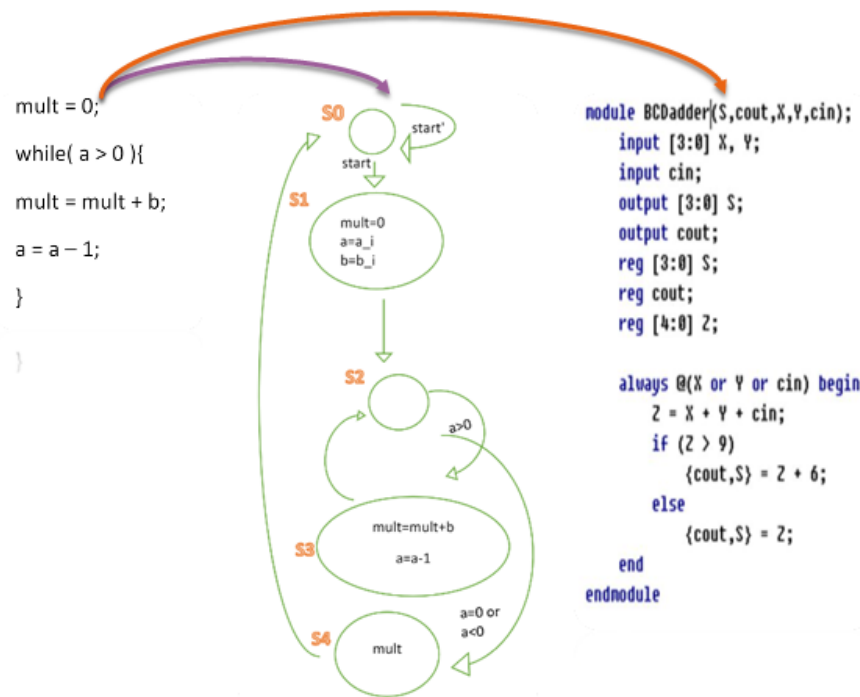


Figure 1.1: Scheme of Project

1.2. Literature Research

1.2.1. C Programming Language

C (/si/, as in the letter c) is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, with a static type system. By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language. Such applications include operating systems and various application software for computer architectures that range from supercomputers to PLCs and embedded systems.

1.2.2. Verilog

Verilog is a Hardware Description Language . It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flipflop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

1.2.3. What is the Difference Between Verilog and C

Verilog is a Hardware Description Language (HDL) used to model electronic systems whereas C is a general-purpose programming language that allows structured programming. Thus, this is the main difference between Verilog and C.

1.2.3.1. File Extensions

File extensions is another difference between Verilog and C. Verilog files have .v or.vh file extension whereas C files have .c file extension.

1.2.3.2. Usage

Verilog helps to design and describe digital systems while C helps to build operating systems, databases, compilers, interpreters, network drivers, etc.

1.2.3.3. Conclusion

The main difference between Verilog and C is that the Verilog is a Hardware Description Language while the C is a high level, general-purpose programming language. In brief, Verilog is based on C.

1.2.4. Algorithmic State Machine

An Algorithmic State Machine (ASM) is a graphical notation similar to a flow-chart, the main difference being that an ASM also includes timing information. This notation can be used to specify the operation of both the datapath and the control unit.

Essentially, the ASM is a state diagram where the states have internal structure described by a simple flow-chart like notation. The symbols used in the notation are states, decisions, and conditional outputs.

1.2.4.1. State

The entry-point of a state is represented graphically by a rectangle. The name of the state and possibly the binary code assigned to the state are written outside the rectangle. Inside the rectangle register transfer operations and outputs are given. Register transfers will happen at the next positive clock edge (assuming positive edge flip-flops are used). Outputs will be immediately set to 1. Any output not mentioned in a state will be set to 0.1.2.

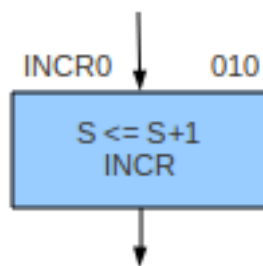


Figure 1.2: State

The actual state comprises all of the symbols that are to be found on all paths coming from the rectangle up to the next rectangle on such a path.

In the example below we have a state called INCR0 with binary code 010. There is a single register transfer which increments S and there is a single output called INCR.

1.2.4.2. Decision

A decision is represented graphically by a diamond (rhombus). The inside of the diamond contains the condition upon which the decision rests. Possible conditions could be checking the status of a signal, comparing two registers for equality, etc. Any such condition can have only one of two outcomes: 0 or 1. As the decision is made immediately, any register updated in the same state as the decision won't be seen by that decision during the current clock cycle.

In the example below we have a decision which checks the signal ADD. What happens if ADD is 0 or 1 isn't shown.

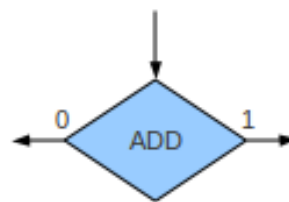


Figure 1.3: State

1.2.4.3. Conditional Output

Conditional output is represented graphically by an oval. The inside of the oval contains the outputs to be set and register transfers to be made. The oval can only occur immediately after a decision. Note that if any outputs occur in an ASM oval, then that ASM describes a Mealy model. If no ovals exist, or no oval has an output, then we have a Moore model.

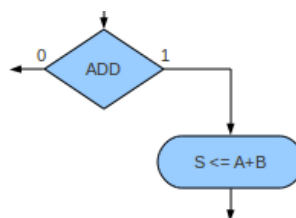


Figure 1.4: State

In the example above, we transfer the sum of A and B to S only when the signal ADD is 1.

1.2.4.4. Datapath Derivation

A well developed ASM contains all necessary detail to derive an appropriate datapath. We can see that any destination of register transfer will need to be a register, so we can analyse the ASM and build the set of registers needed. The register transfer operations performed will give us the combinational logic circuits needed, e.g., adder/subtractors, shifters, etc. It may be the case that the operations dictate that some registers will be best implemented as counters or shift registers. We also need to analyse the decisions found in the ASM as these may need circuits such as comparators.

1.2.4.5. Control Unit Derivation

As mentioned earlier, the ASM can be seen as a type of state diagram. We also know that a state diagram can be transformed into a state table, and that a state table can be used to derive a sequential logic circuit. To build a complete state table, we need all the input and output signals. We have all external input and output signals explicitly given in the ASM, and, after deriving the datapath we have can find the set of internal control signals and how they will need to be manipulated in each state, and this gives us what we need to complete state table.

1.2.4.6. Example

As an example of an ASM, consider the Euclidean greatest common divisor (gcd) which is one of the oldest algorithms we know of. Mathematically, the gcd of two positive integers is defined as follows:

$$\text{gcd}(a,a) = a$$

$$\text{gcd}(a,b) = \text{gcd}(a-b, b) \text{ if } a > b,$$

$$\text{gcd}(a,b) = \text{gcd}(a, b-a) \text{ otherwise.}$$

The above ASM was created to implement the gcd. It comprises three states (IDLE, COMP, and DONE), two registers (A and B), three inputs (X, Y, and GO), and two outputs (DONE and either A or B — let's choose A). If the initial state is IDLE and the arguments of the gcd are X and Y, then gcd(X,Y) will eventually be loaded into A. The input GO starts the whole process, and the output DONE is pulsed (0 to 1) when A contains the result.

[1]–[6].

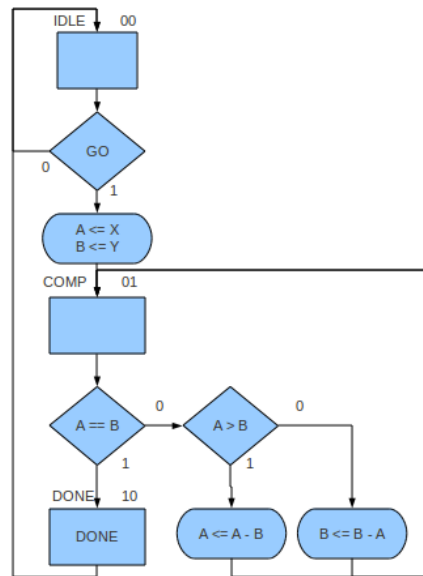


Figure 1.5: State

1.3. Project Design Plan

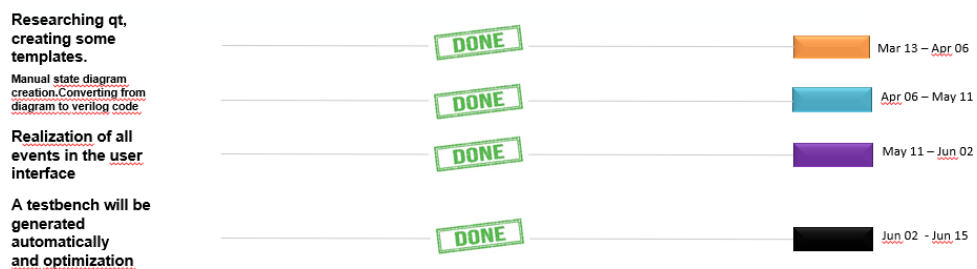


Figure 1.6: Design Plan

I will do the lexical analysis to understand whether the code to be translated is suitable for translation, and then I will analyze the code with the parsing process and try to reveal the asm and verilog code. All processes will run inside the program written in C++.

1.3.0.1. C Code

Rules and restrictions warnings will be shown to the user for the codes to be retrieved in the Tool. Then a code that complies with the rules will be requested, and the c code will be requested from the user. When the code that does not comply with

the rules is entered, the user will be informed of the errors.

1.3.0.2. Parser

Firstly tool aims to declare function which given input and name. After that tool aims to find how many step do we have. In each step I need to take note what should I write in that steps. So I am using map library for that. I am checking keywords and brackets to give shape my verilog code. In each step I am checking to end of step which I created so after that I am incrementing current step stage. So that I can fill all the steps with that.

1.3.0.3. ASM

In this part, the rectangle and line functions from the openCV library will be used. States and arrows will be placed in the map in accordance with the appropriate coordinates on an empty map.

1.3.0.4. Verilog Code

The parsed tokens are placed in the verilog code in accordance with the state machine theme, converted into text and presented to the user.

1.3.0.5. Design plan in qt

The tool takes the c code from the relevant path and converts the state diagram to a json format. Then the tool reads the json file and prints it properly on the screen in the interface. After doing all this, it outputs the verilog code from the json file. If editing is done on the state diagram formed later, the user can update the verilog file by pressing the convert button. fig: 1.7

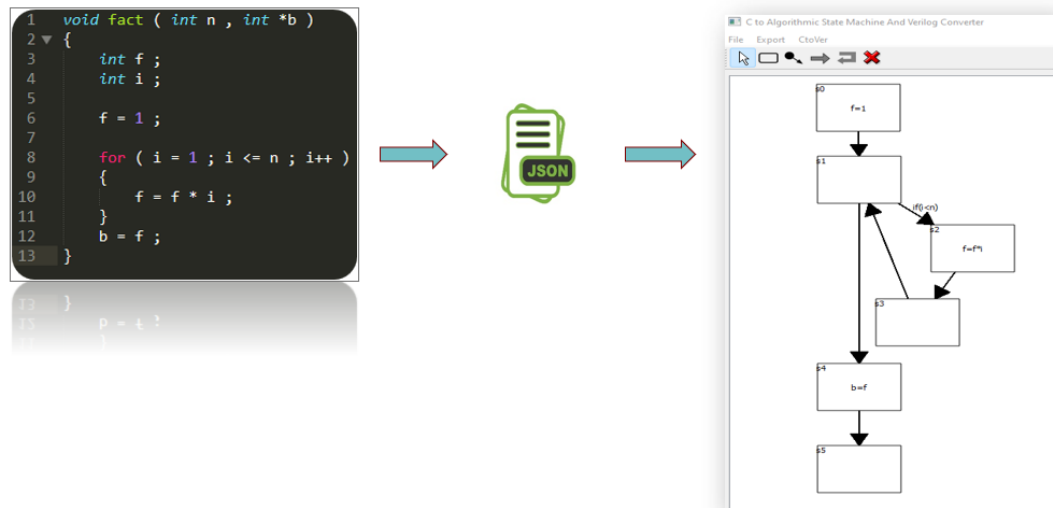


Figure 1.7: C to Json format

1.4. Project Goal

The focus of the project is the world of embedded and real-time systems. Verilog and state machine are indispensable parts of this world. However, manually designed state diagrams require a long process. The project plans to eliminate these difficulties. Projects designed simply in C will be instantly transformed into state diagram and verilog code.

Thanks to this project, it will make the use of Asm and Verilog more effective in the world of embedded and real-time systems.

Some advantages are;

Faster to develop

Easier to change

Easier to debug

1.5. Project Requirement and Libraries

1.5.1. Project Requirement

Verilog has some C-like syntax, but it's not an imperative programming language. C and Verilog work with different underlying realities. This project isn't practical for

arbitrary C code, or even big C programs. Algorithmic simple and tight c codes can work in this tool. (e.g, Matrix multiply, greatest common divisor , factorial ,largest element of array , check prime number).There must be requirements for these;

->It will be necessary to impose some restrictions on the c code. Illegal codes will not be converted.

->C code consisting of a single function with assignment , loop and branch statements should be take from the user.

->The generated Verilog code must be Behavioral Modeling Style. Behavioral modeling contains procedural statements that control the simulation and manipulate the data types of the variables involved.

->The arrows in the diagram should contain conditional statements.

The project should have following kinds of technology:

- 1)The ability to parse C code accurately
- 2)The ability to analyze the C code to determine its control flow and data flow in precise detail
- 3)The ability to built with a state diagram that is dynamic structure.
- 4)A tool that can walk the control and dataflows and synthesize Verilog fragments

1.5.2. Libraries

I had to use the qt general canvas library for the state diagram.Also I used cpp libraries for code analysis. <QComboBox>, <QFrame>, <QGraphicsItem>, <QGroupBox>, <QLineEdit>, <QListView>, <QString>, <QStringListModel>, <QMainWindow>, <iostream>, <cstring>, <fstream>, <string>, <vector>, <cctype>, <algorithm>, <regex>, <sstream>, <stdlib.h>, <QString>, <QFrame>.

1.6. Timeline of Project

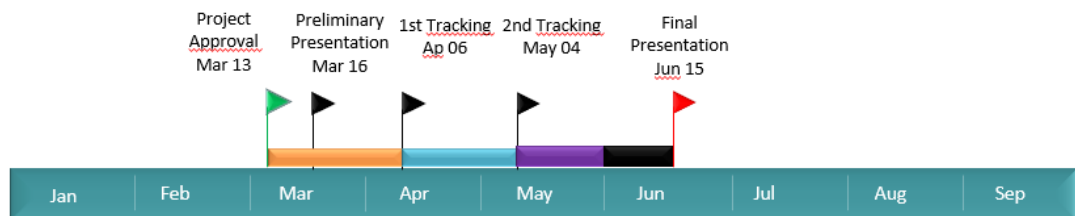


Figure 1.8: Timeline of Project

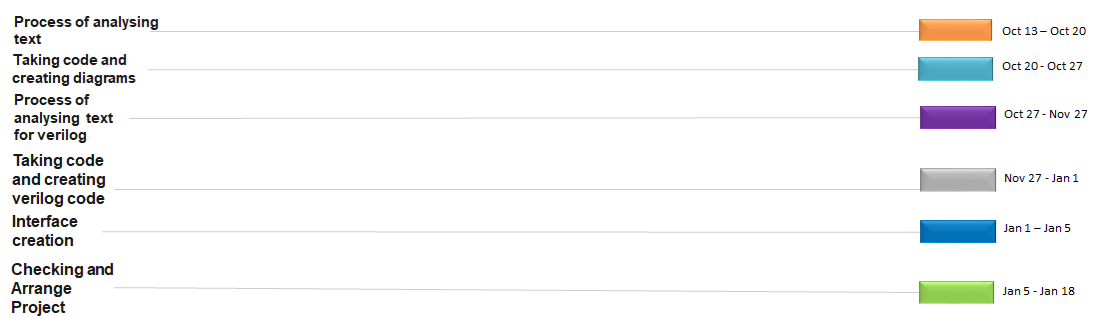


Figure 1.9: Plan

2. IMPLEMENTATION

In this project I mainly take the point from last year lesson which is Programming languages. I checked parsing styles and I choose the best for this parsing thing.

2.1. Design Gui

2.1.1. Qt Code Classes

2.1.1.1. Class Fsd

Functions of all buttons created in the menu bar are defined in this class. If we list them in enum Mode, as follows;

```
enum Mode InsertState, InsertPseudoState, InsertTransition, InsertLoopTransition, SelectItem, DeleteItem, ConvertJtoVer ;
```

2.1.1.2. Class JsonToVlog

In this section, there are definitions of headers and subtitles for the json file that the program will use. The functions in this class are as follows;

Readfile(): to read the json file

Getvars(): collects init headers for verilog code if it comes from c code.

ToVlog(): converts json file to verilog code and prints it.

2.1.1.3. Class Mainwindow

In the mainwindow class, we have ready-made classes that we use, and we can easily do all our actions on the canvas with these classes.

class QAction; class QToolBox; class QSpinBox; class QComboBox; class QFontComboBox; class QButtonGroup; class QLineEdit; class QGraphicsTextItem; class QFont; class QToolButton; class QAbstractButton; class QGraphicsView; .In these classes, all the actions to be performed on the canvas are available.

Apart from these, all the actions of the menu part are defined in this part. Here are the functions I wrote for them: void stateSelected(State *); void transitionSelected(Transition *); void fsdModified(); void save(); void saveAs(); void openFile();


```
void newDiagram(); void quit(); void about(); void exportDot(); void CtoAsmtoVerilog();
```

2.1.1.4. Class Properties

In this section, there are selected and change methods to make changes on the selected state or transitions. Using these methods, we can add explain and enumerate states and arrows.

2.1.1.5. Class State

In this class, the visual properties of the states were determined. In addition, all the methods to be made on the states are also available in this class.

2.1.1.6. Class State Diagram

In this section, there are methods to create json file for analysis and state diagram from c code.

2.1.1.7. Class Transition

All properties and methods of arrows are available in this class.

2.2. Analyze C Code

Firstly my project aims to declare function which given input and name. After that my tool aims to find how many step do we have. In each step I need to take note what should I write in that steps. So I am using map library for that. I am checking keywords and brackets to give shape my verilog code. In each step I am checking to end of step which I created so after that I am incrementing current step stage. So that I can fill all the steps with that.

Lets evaluate for (i = 0 ; i < 5 ; i++)

As you see we have to have space after all statements. Our program will check all words like brut force.

In this example after we see for keyword we will jump 1 string because we don't need (that

After that we check our incremented value Up one step because of ; After that

we got our if statement for our for loop Up one step because of ; Now we will check incrementer's increment value.

We put all this thing in the same state. We put state map if statement and if state address after that we put else state and else address So we can know where should we go.

After that statement we go until see curly braces that keyword to check for loop states. And put this states in to for's state +1.for's state +2 just return state to for's state.

But we have to check if we have another for loop while loop or if statement in this for loop. So we have stack for that and after we see for loop we put return address in it so if we see another for loop we change 1.for loop else address so that our program works fine.

In every statement we must put our return address to stack . After edit our address in the other loops we have to remove.

Another important point is if we don't see anything like for while if .. in themselves we have to update stack after see curly braces this keyword and after all statement states we have to increment current state. And total state value so our states looks fine and works fine.

2.2.1. Analysis Functions

These functions synthesize the code.

2.2.1.1. FindInt

Used to detect digits in function

2.2.1.2. WithoutBracket

Identifying non-bracket status

2.2.1.3. IsHaveBracket

Function that looks for brackets inside incoming text

2.2.2. Convert Verilog

2.2.2.1. ConvertAndPrint

With the keyword parsed and filled into the stack in the Convert function, the verilog code is printed to the file. The code also decides in which state to place the increment parts of the conditional statements and loop parts.

2.2.2.2. Convert()

Convert code examines loop, statement, condition cases. And then it places them in the 3 template state diagram states that I have defined before. For example, the code is analyzed and the number of states is decided in this function according to the number of loops. Then, the previously found keywords are correctly placed in the verilog state machine theme to convert it to verilog. In this way, the closest verilog code to work is created thanks to this function. [7].

2.3. Draw State Diagram

After all this statement now we have to print all statements. I put all states keywords in their index map so we start print with one for loop And put this general state:0 and start end things.

After start keyword we write our map values which in that index. Than print end keyword.

Last state means exit state I wanted to show it in chart but this situation gives some error so I just deleted it.

2.3.1. Draw Functions

These functions make the diagram appear.

2.3.1.1. Draw Line

This function takes the current coordinates and draws a line from one state to another.

2.3.1.2. Draw line arrow

This function is using the line function. It directs the arrows that will carry the condition statements to the right places.

2.3.1.3. Draw Text

Function used to place appropriate words from keywords into states.

2.3.1.4. Draw State Num

After the code analysis is finished, the calculated state number is written next to the states on the map.

2.3.1.5. Rectangel Print

This function has a dynamic structure and allows us to draw states. The number of lines is checked while drawing the states. If there is an overflow, state in size i is enlarged. As a result, a dynamic structure is established.

2.3.1.6. Calculate Points

This function is needed to accurately determine the location of the arrows and states while drawing the diagram. It works to determine the current positions and to direct the arrows correctly.

3. QT CREATOR

In this section, how a gui is designed in the qt environment and what kind of environment is presented to the user will be mentioned.

3.0.1. Qt Creator

Qt Creator is a cross-platform C++, JavaScript and QML integrated development environment (IDE) which simplifies GUI application development. It is part of the SDK for the Qt GUI application development framework and uses the Qt API, which encapsulates host OS GUI function calls. It includes a visual debugger and an integrated WYSIWYG GUI layout and forms designer. The editor has features such as syntax highlighting and autocompletion. Qt Creator uses the C++ compiler from the GNU Compiler Collection on Linux. On Windows it can use MinGW or MSVC with the default install and can also use Microsoft Console Debugger when compiled from source code.[8].



Figure 3.1: Qt Cpp Icon

3.0.2. C to Asm and Verilog Editor

The tool is developed on the qt platform. Qt 5.0.2 (community) has been chosen as the qt version. The GUI of the tool is designed in this way. A suitable environment has been created for the user to draw their own state diagram.[9].

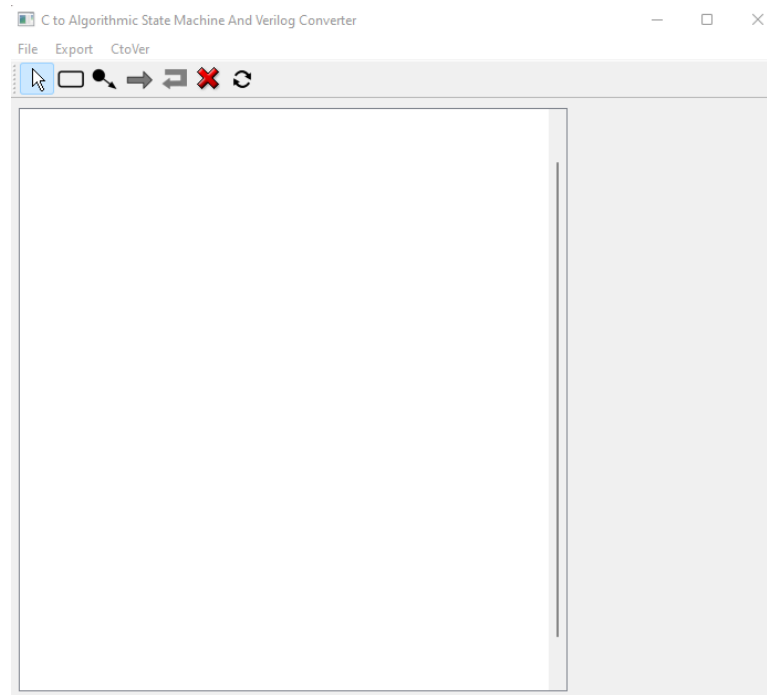


Figure 3.2: Qt Cpp GUI

3.0.2.1. Documentation of Editing

3.0.2.1.1 Add a state To add a state, select the fig: 3.3 button in the toolbar and click on the canvas.



Figure 3.3: Add State Icon

3.0.2.1.2 Add a transition To add a transition, select the button, fig: 3.4 click on the start state and, keeping the mouse button pressed, go the end state and release mouse button.



Figure 3.4: Add a transition Icon

3.0.2.1.3 Add a self transition To add a self transition (from a state to itself) , select the fig: 3.5 button and click on start state (the location of the click will decide on that of the transition).



Figure 3.5: Add a self transition Icon

3.0.2.1.4 Add an initial transition To add an initial transition, select the fig: 3.6 button and click on initial state .



Figure 3.6: Add an initial transition Icon

3.0.2.1.5 Delete a state or a transition To delete a state or a transition, select the fig: 3.7 button and click on the state or transition (deleting a state will also delete all incoming and outgoing transitions).



Figure 3.7: Delete a state or a transition Icon

3.0.2.1.6 Move a state To move a state, select the fig: 3.8 button and drag the state.



Figure 3.8: Move a state Icon

3.0.2.1.7 Edit a state or a transition To edit a state or a transition, select the fig: 3.8 button, click on the corresponding item and update the property panel on the right.

3.0.2.2. Saving and loading

The current diagram can be saved by invoking the ‘Save’ or ‘Save As’ action in the ‘File’ menu. A saved diagram can be reloaded by invoking the ‘Open’ action in the ‘File’ menu. fig: 3.9 The ‘New’ action in the ‘File’ menu clears the diagram

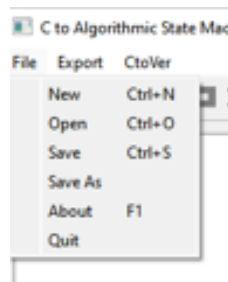


Figure 3.9: Save and Open Icon

3.0.2.3. C to Verilog

In order for the selected c code to be converted to ASM and Verilog code, action is taken from the CtoVer button in the menu. The C code is taken from the required path, the ASM is drawn on the canvas, and then converted to Verilog code, the converted verilog code is saved to the desired path. fig: 3.10

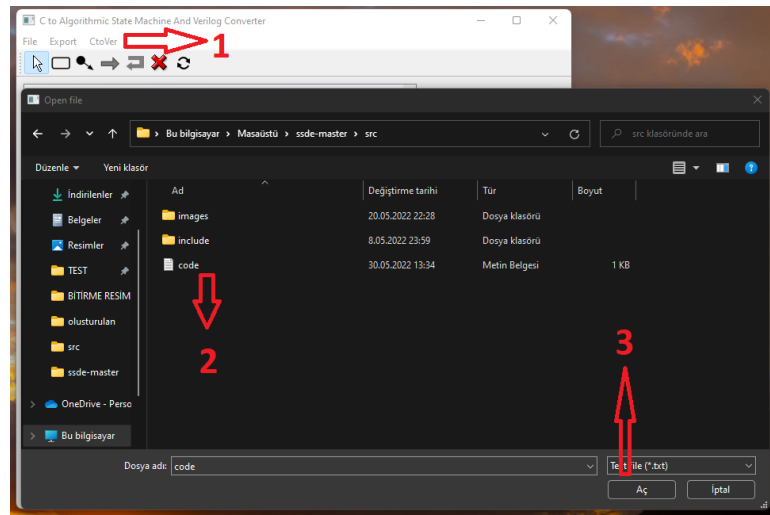


Figure 3.10: C To Verilog Action Icon

4. TEST AND RESULT

In this part, some simple c codes are made in accordance with the rules and run in the tool. [10].

4.0.1. To find the factorial of a number

For example, 4 factorial, that is, 4! can be written as: $4! = 4 \times 3 \times 2 \times 1 = 24$.

Observe the numbers and their factorial values given in the following table. To find the factorial of a number, multiply the number with the factorial value of the previous number. For example, to know the value of 6! multiply 120 (the factorial of 5) by 6, and get 720. For 7! multiply 720 (the factorial value of 6) by 7, to get 5040.

4.0.1.1. C code

A suitable c code is given to the tool.

```
void fact ( int n , int *Result )  
{  
    int f ;  
    int i ;  
  
    f = 1 ;  
  
    for ( i = 1 ; i <= n ; i++ )  
    {  
        f = f * i ;  
    }  
    Result = f ;  
}
```

Figure 4.1: Factorial C Code

4.0.1.2. State Diagram

The state diagram consisting of 5 states is as follows

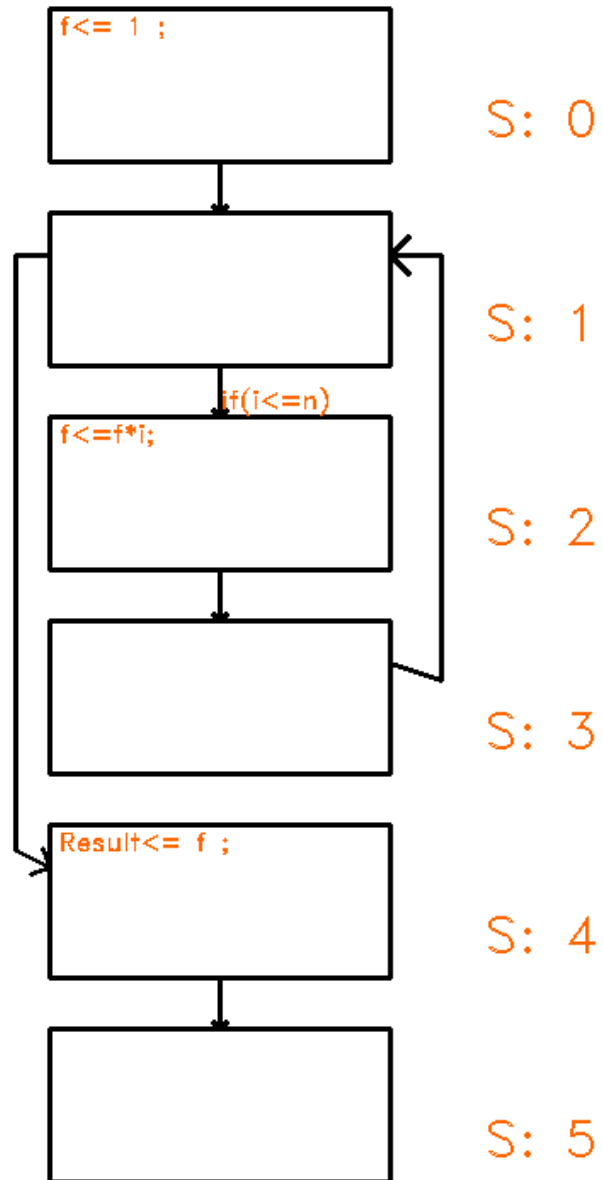


Figure 4.2: State diagram for fact created by the tool

4.0.1.3. Verilog Code

Verilog code closest to executing for factorial function

```
1  module fact(n,Result,clk);
2  input clk;
3  reg[31:0]state;
4  input[31:0]n;
5  output[31:0]Result;
6  reg[31:0]Result;
7
8  localparam
9      S0 = 0,
10     S1 = 1,
11     S2 = 2,
12     S3 = 3,
13     S4 = 4,
14     S5 = 5;
15
16  reg [31:0] f;
17  reg [31:0] i;
18  initial begin
19      state <= 0;
20      i <= 1;
21  end
22
23  always @(posedge clk)
24  case(state)
25
26  S0:
27  begin
28      f<= 1 ; state <= S1;
29  end
30  S1:
31  begin
32      if(i<=n)
33      begin
34          state <= S2;
35      end
36      else
37      begin
38          state <= S4;
39      end
40  end
41  S2:
42  begin
43      f<=f*i;
44      i <= i + 1;
45
46      state <= S3;
47  end
48  S3:
49  begin
50
51      state <= S1;
52  end
53  S4:
54  begin
55      Result<= f ;    state <= S5;
56  end
57  S5:
58  begin
59
60  end
61  endcase
62  endmodule
63
```

Figure 4.3: Fact verilog code created by the tool

4.0.1.4. Running the verilog code generated by the tool

The running of verilog code in Quartus 2 64 bit is shown step by step.

```
# vsim work.fact_test
# Loading work.fact_test
# Loading work.fact
add wave -position insertpoint
sim:/fact_test/n \
sim:/fact_test/f \
sim:/fact_test/clk \
sim:/fact_test/Result
VSIM6> step -current
# f = x
# ,n= 6
# ,state= 0
# ,i= 1
# ,Result= x
#
# f = 1
# ,n= 6
# ,state= 1
# ,i= 1
# ,Result= x
#
# f = 1
# ,n= 6
# ,state= 2
# ,i= 1
# ,Result= x
#
# f = 1
# ,n= 6
# ,state= 3
# ,i= 2
# ,Result= x
#
# f = 1
# ,n= 6
# ,state= 1
# ,i= 2
# ,Result= x
#
# f = 2
# ,n= 6
# ,state= 3
# ,i= 3
# ,Result= x
#
# f = 2
# ,n= 6
# ,state= 1
# ,i= 3
# ,Result= x
#
# f = 2
# ,n= 6
# ,state= 2
# ,i= 3
# ,Result= x
#
# f = 6
# ,n= 6
# ,state= 3
# ,i= 4
# ,Result= x
#
# f = 6
# ,n= 6
# ,state= 2
# ,i= 4
# ,Result= x
#
# f = 24
# ,n= 6
# ,state= 3
# ,i= 5
# ,Result= x
#
# f = 24
# ,n= 6
# ,state= 1
# ,i= 5
# ,Result= x
#
# f = 720
# ,n= 6
# ,state= 5
# ,i= 7
# ,Result=720
#
# f = 120
# ,n= 6
# ,state= 3
# ,i= 6
# ,Result= x
#
# f = 120
# ,n= 6
# ,state= 1
# ,i= 6
# ,Result= x
#
# f = 120
# ,n= 6
# ,state= 2
# ,i= 6
# ,Result= x
#
# f = 720
# ,n= 6
# ,state= 3
# ,i= 7
# ,Result= x
#
```

Figure 4.4: Verilog code simulation

4.0.2. Greatest Common Divisor

The HCF or GCD of two integers is the largest integer that can exactly divide both numbers (without a remainder).

4.0.2.1. C code

A suitable c code is given to the tool.

```
void gcd ( int a , int b , int *result )
{
    int hcf;
    int i ;
    for ( i = 0 ; i<=a && i<=b ; i++ )
    {
        if ( a%i==0 && b%i==0 )
        {
            hcf = i ;
        }
        else {
            hcf = hcf + 0 ;
        }
    }
    result = hcf ;
}
```

Figure 4.5: GCD c code

4.0.2.2. State Diagram

The state diagram consisting of 6 states is as follows

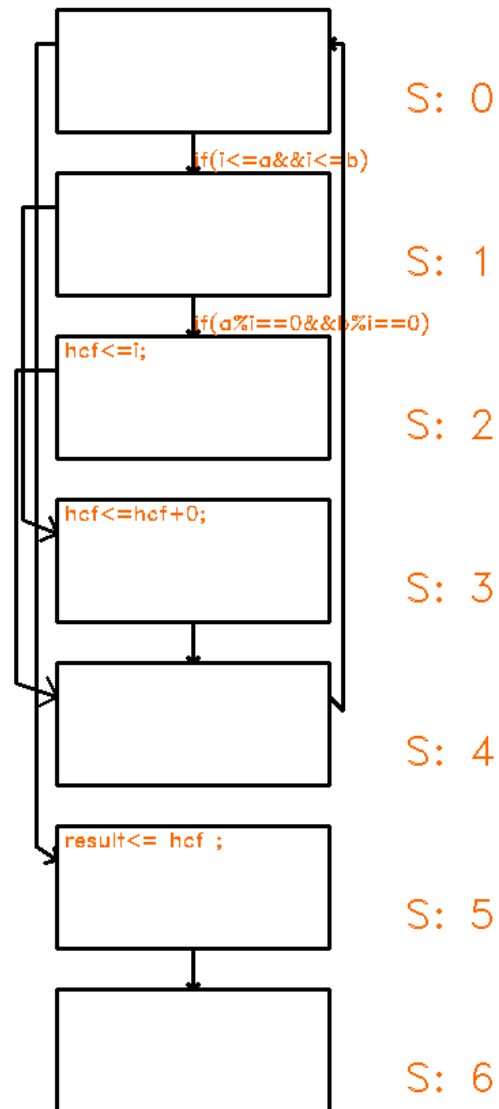


Figure 4.6: GCD state diagram created by the tool

4.0.2.3. Verilog Code

Verilog code closest to executing for GCD function

```
1  module gcd(a,b,result,clk);
2  input clk;
3  reg[31:0]state;
4  input[31:0]a;
5  input[31:0]b;
6  output[31:0]result;
7
8  reg[31:0]result;
9
10 ▼ localparam
11     S0 = 0,
12     S1 = 1,
13     S2 = 2,
14     S3 = 3,
15     S4 = 4,
16     S5 = 5,
17     S6 = 6;
18
19  reg [31:0] hcf;
20  reg [31:0] i;
21  initial begin
22  state <= 0 ;
23  hcf <= 0;
24  i <= 0;
25  end
26
27  always @(posedge clk)
28  case(state)
29
30  S0:
31  begin
32  i <= i + 1;
33  if(i<=a&& i<=b)
34  begin
35      state <= S1;
36  end
37  else
38  ▼ begin
39      state <= S5;
40  end
41
42  S1:
43  ▼ begin
44      if(a%i==0&&b%i==0)
45      begin
46          state <= S2;
47      end
48  else
49  ▼ begin
50      state <= S3;
51  end
52
53
54
55  end
56  S2:
57  ▼ begin
58      hcf<=i;
59      state <= S4;
60  end
61  S3:
62  ▼ begin
63      hcf<=hcf+0;
64      state <= S4;
65  end
66  S4:
67  ▼ begin
68      state <= S0;
69  end
70
71  S5:
72  begin
73  result<= hcf ; state <= S6;
74  end
75  S6:
76  begin
77
78  ▼ end
79  endcase
80  endmodule
```

Figure 4.7: GCD verilog code created by the tool

4.0.2.4. Running the verilog code generated by the tool

The running of verilog code in Quartus 2 64 bit is shown step by step.

```
module gcd_test;
reg  clk;
reg  [31:0] a;
reg  [31:0] b;

wire [31:0] result;

gcd commondivide(a,b,result,clk);

initial begin
    clk = 0;
    b=32'd12;
    a=32'd20;
    #2 clk = 1;
    #2 clk = 0;
```

Figure 4.8: GCD test code

```
# ,hcf= 4
# ,a=20
# ,state= 6
# ,b=12
# ,result= 4
#
```

Figure 4.9: GCD test result

4.0.3. Largest element in array

In this example, you will see to display the largest element entered by the user in an array.

4.0.3.1. C code

A suitable c code is given to the tool.

```
void largest ( int *b )
{
    int arr[3] ;
    int i ;
    int max ;
    max = 0 ;
    arr[0] = 2 ;
    arr[1] = 4 ;
    arr[2] = 6 ;

    for ( i = 0 ; i < 3 ; i++ ) {

        if ( arr[i] > max ) {
            max = arr[i] ;
        }
        else {

        }

    }

    b = max ;
}
```

Figure 4.10: Largest element C Code

4.0.3.2. State Diagram

The state diagram consisting of 7 states is as follows

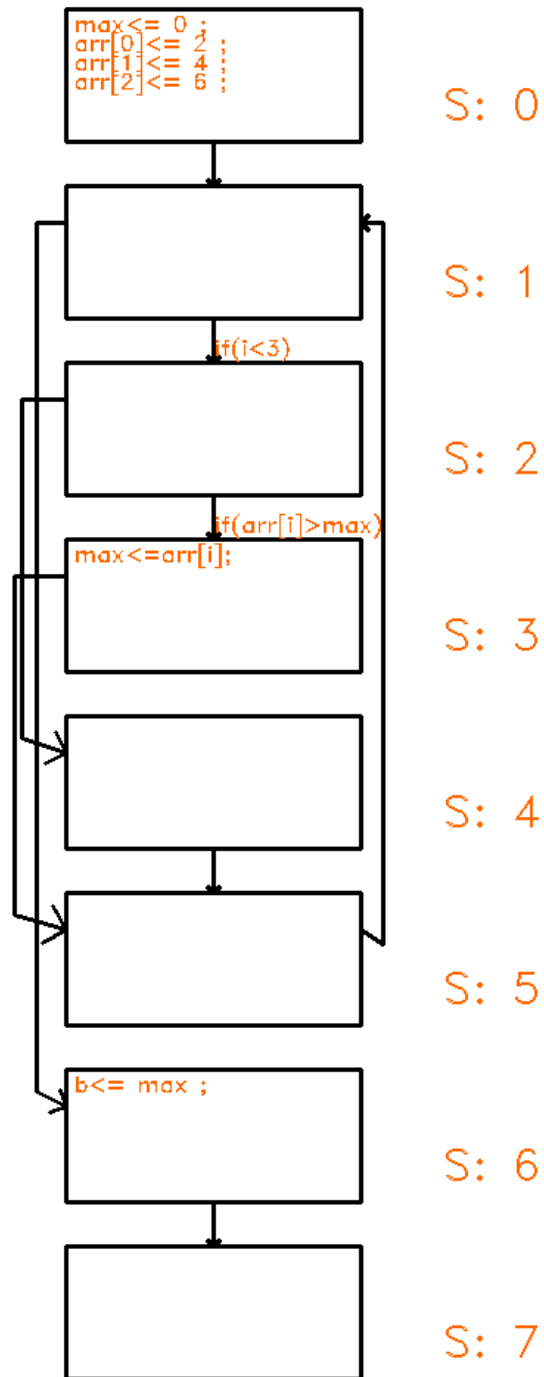


Figure 4.11: State diagram for largest created by the tool

4.0.3.3. Verilog Code

Verilog code closest to executing for largest element function

```
1 module largest(b,clk);
2   input clk;
3   reg[31:0]state;
4   output[31:0]b;
5   reg[31:0]b;
6
7   localparam
8     S0 = 0,
9     S1 = 1,
10    S2 = 2,
11    S3 = 3,
12    S4 = 4,
13    S5 = 5,
14    S6 = 6,
15    S7 = 7;
16
17   reg [31:0] arr[0:2];
18   reg [31:0] i;
19   reg [31:0] max;
20   initial begin
21     state <= 0 ;
22     i <= 0;
23     max <= 0;
24   end
25
26   always @(posedge clk)
27     case(state)
28
29     S0:
30       begin
31         max<= 0 ;   state <= S1;arr[0]<= 2 ;arr[1]<= 4 ;arr[2]<= 6 ;
32       end
33     S1:
34       begin
35         i <= i + 1;
36
37         if(i<3)
38           begin
39             state <= S2;
40           end
41         else
42           begin
43             state <= S6 ;
44           end
45         end
46       S2:
47         begin
48           if(arr[i]>max)
49             begin
50               state <= S3;
51             end
52           else
53             begin
54               state <= S4;
55             end
56         end
57       S3:
58         begin
59           max<=arr[i];
60           state <= S5;
61         end
62       S4:
63         begin
64           state <= S5;
65         end
66       S5:
67         begin
68           state <= S1;
69         end
70       S6:
71         begin
72           b<= max ;   state <= S7;
73         end
74       S7:
75         begin
76
77         end
78       end
79     endcase
80   endmodule
81
82
```

Figure 4.12: Verilog code for largest element created by the tool

4.0.3.4. Running the verilog code generated by the tool

The running of verilog code in Quartus 2 64 bit is shown.

```
# i = 4
# ,state= 7
# ,max= 6
# ,b= 6
#
```

Figure 4.13: Verilog code simulation

4.0.4. Power

The ComputePower() function takes two arguments (base value and power value) and, returns the power raised to the base number.

4.0.4.1. C code

A suitable c code is given to the tool.

```
void ComputePower ( int exponent , int base , int *b )
{
    int result ;

    result = 1 ;

    while ( exponent != 0 ) {
        result = base * base ;

        exponent = exponent - 1 ;
    }

    b = result ;
}
```

Figure 4.14: Power C Code

4.0.4.2. State Diagram

The state diagram consisting of 5 states is as follows

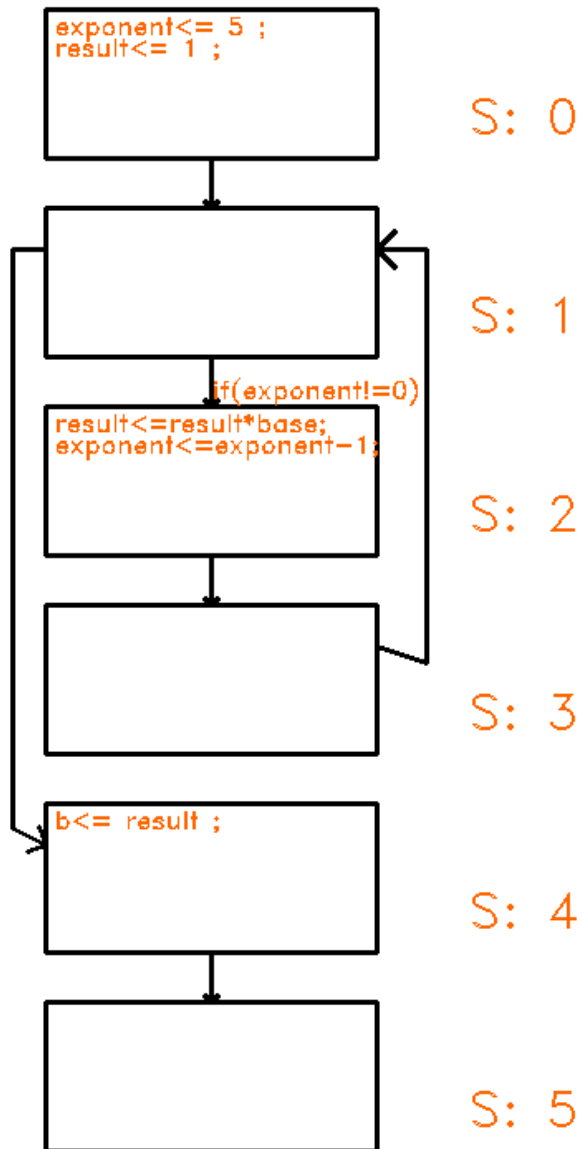


Figure 4.15: State diagram for power created by the tool

4.0.4.3. Verilog Code

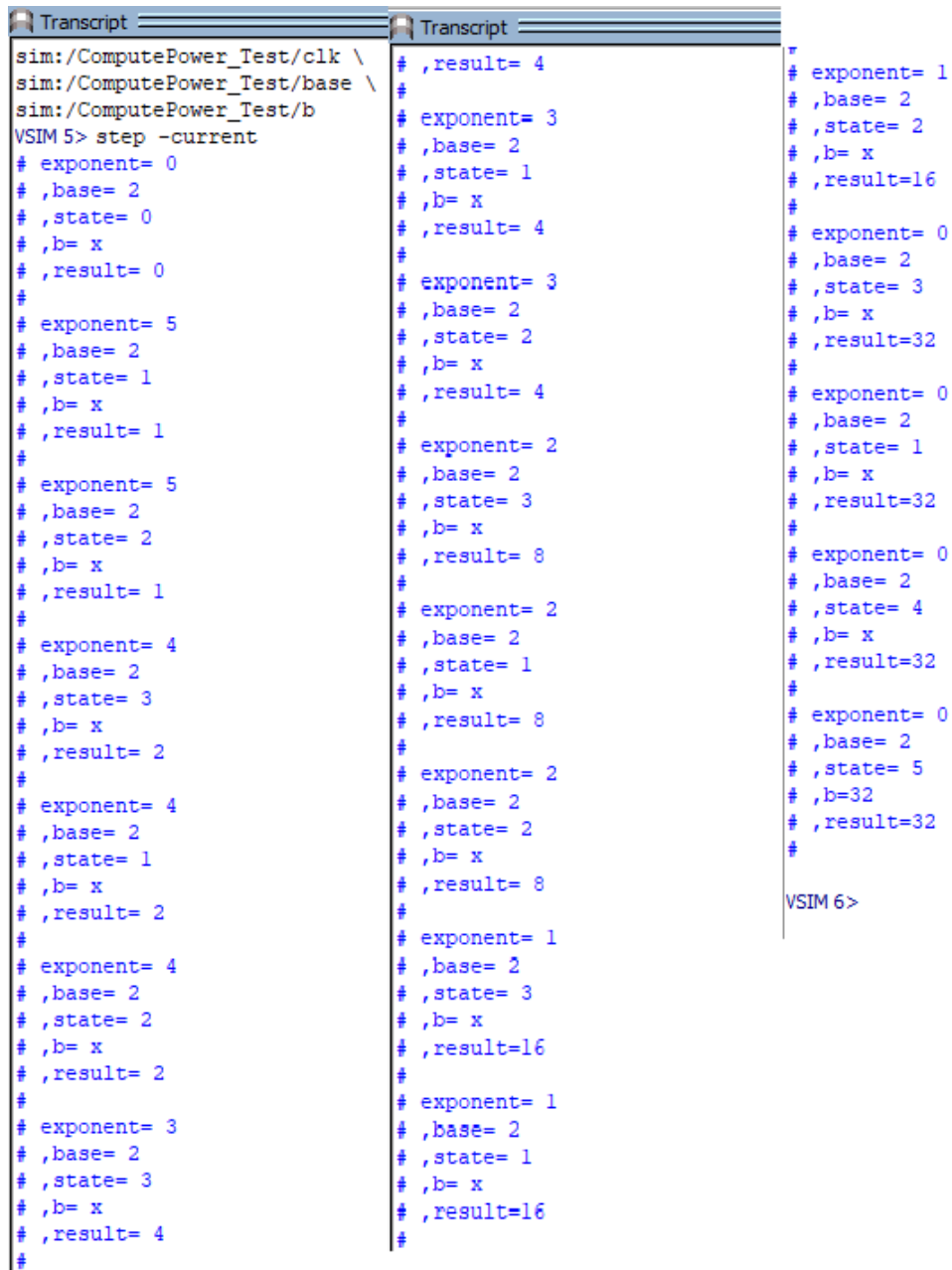
Verilog code closest to executing for power function

```
1  module ComputePower(base,b,clk);
2  input clk;
3  reg[31:0]state;
4  input[31:0]base;
5  output[31:0]b;
6  reg[31:0]b;
7
8  localparam
9      S0 = 0,
10     S1 = 1,
11     S2 = 2,
12     S3 = 3,
13     S4 = 4,
14     S5 = 5;
15
16  reg [31:0] result;
17  reg [31:0] exponent;
18  initial begin
19      state <= 0 ;
20      result <= 0;
21      exponent <= 0;
22  end
23
24  always @(posedge clk)
25  case(state)
26
27  S0:
28  begin
29      exponent<= 5 ; state <= S1;result<= 1 ;
30  end
31  S1:
32  begin
33      if(exponent!=0)
34      begin
35          state <= S2;
36      end
37      else
38      begin
39          state <= S4;
40      end
41  end
42  S2:
43  begin
44      result<=result*base;
45      exponent<=exponent-1;
46
47      state <= S3;
48  end
49  S3:
50  begin
51
52      state <= S1;
53  end
54  S4:
55  begin
56      b<= result ; state <= S5;
57  end
58  S5:
59  begin
60
61  end
62  endcase
63  endmodule
```

Figure 4.16: Verilog code for power created by the tool

4.0.4.4. Running the verilog code generated by the tool

The running of verilog code in Quartus 2 64 bit is shown.



The image shows three side-by-side 'Transcript' windows from the Quartus 2 64 bit IDE, displaying the output of a Verilog simulation. The left window shows the initial simulation steps, including setting the exponent to 0 and base to 2, and calculating the result as 0. The middle window shows the simulation steps for exponent 3, base 2, state 1, and result 4. The right window shows the simulation steps for exponent 1, base 2, state 2, and result 16. The simulation is controlled by the 'VSIM' command, and the results are displayed in a structured format with comments and variable assignments.

```

sim:/ComputePower_Test/clk \
sim:/ComputePower_Test/base \
sim:/ComputePower_Test/b
VSIM 5> step -current
# exponent= 0
# ,base= 2
# ,state= 0
# ,b= x
# ,result= 0
#
# exponent= 5
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 1
#
# exponent= 5
# ,base= 2
# ,state= 2
# ,b= x
# ,result= 1
#
# exponent= 4
# ,base= 2
# ,state= 3
# ,b= x
# ,result= 2
#
# exponent= 4
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 2
#
# exponent= 4
# ,base= 2
# ,state= 2
# ,b= x
# ,result= 2
#
# exponent= 3
# ,base= 2
# ,state= 3
# ,b= x
# ,result= 4
#
# ,result= 4
#
# exponent= 3
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 4
#
# exponent= 3
# ,base= 2
# ,state= 2
# ,b= x
# ,result= 4
#
# exponent= 2
# ,base= 2
# ,state= 3
# ,b= x
# ,result= 8
#
# exponent= 2
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 8
#
# exponent= 2
# ,base= 2
# ,state= 2
# ,b= x
# ,result= 8
#
# exponent= 1
# ,base= 2
# ,state= 3
# ,b= x
# ,result= 16
#
# exponent= 1
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 16
#
# exponent= 1
# ,base= 2
# ,state= 2
# ,b= x
# ,result= 16
#
# exponent= 0
# ,base= 2
# ,state= 3
# ,b= x
# ,result= 32
#
# exponent= 0
# ,base= 2
# ,state= 1
# ,b= x
# ,result= 32
#
# exponent= 0
# ,base= 2
# ,state= 4
# ,b= x
# ,result= 32
#
# exponent= 0
# ,base= 2
# ,state= 5
# ,b= 32
# ,result= 32
#
VSIM 6>

```

Figure 4.17: Verilog code for power simulation

4.0.5. IsPrime

In this example, we check whether an integer entered by the user is a prime number or not.

4.0.5.1. C code

A suitable c code is given to the tool.

```
1 void isPrime ( int number , int *result ) {  
2  
3     int i ;  
4     int k ;  
5     k = 0 ;  
6     for ( i = 2 ; i <= number ; i++ ) {  
7  
8         if ( number%i == 0 ) {  
9             k = k + 1 ;  
10        }  
11        else {  
12            k = k + 0 ;  
13        }  
14    }  
15  
16    if ( k >= 2 ) {  
17        result = 0 ;  
18    }  
19    else {  
20        result = 1 ;  
21    }  
22  
23 }
```

Figure 4.18: Is Prime C Code

4.0.5.2. State Diagram

The state diagram consisting of 9 states is as follows

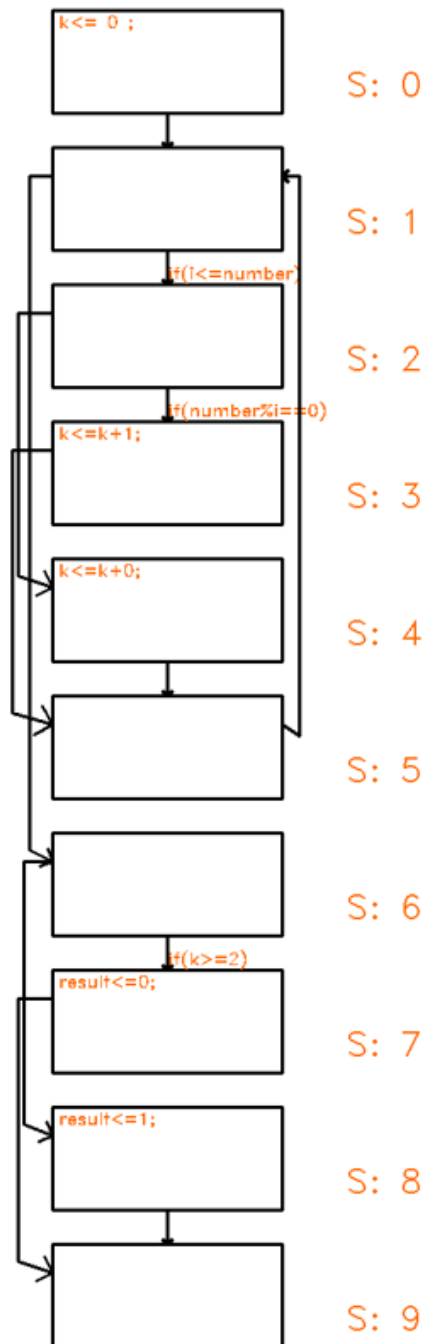


Figure 4.19: State diagram for isprime created by the tool

4.0.5.3. Verilog Code

Verilog code closest to executing for isprime function

```
1  module isPrime(number,result,clk);
2  input clk;
3  reg[31:0]state;
4  input[31:0]number;
5  output[31:0]result;
6  reg[31:0]result;
7
8  localparam
9      S0 = 0,
10     S1 = 1,
11     S2 = 2,
12     S3 = 3,
13     S4 = 4,
14     S5 = 5,
15     S6 = 6,
16     S7 = 7,
17     S8 = 8,
18     S9 = 9;
19
20  reg [31:0] i;
21  reg [31:0] k;
22  initial begin
23      state <= 0 ;
24      i <= 2;
25      k <= 0;
26  end
27
28  always @(posedge clk)
29  case(state)
30
31      S0:
32      begin
33          k<= 0 ; state <= S1;
34      end
35      S1:
36      begin
37          if(i<=number)
38          begin
39              state <= S2;
40          end
41          else
42          begin
43              state <= S6 ;
44          end
45      end
46      S2:
47      begin
48          if(number%i==0)
49          begin
50              state <= S3;
51          end
52          else
53          begin
54              state <= S4;
55          end
56          i <= i + 1;
57      end
58  end
59  S3:
60  begin
61      k<=k+1;
62      state <= S5;
63  end
64  S4:
65  begin
66      k<=k+0;
67      state <= S5;
68  end
69  S5:
70  begin
71      state <= S1;
72  end
73  S6:
74  begin
75      if(k>=2)
76      begin
77          state <= S7;
78      end
79      else
80      begin
81          state <= S8;
82      end
83  end
84  S7:
85  begin
86      result<=0;
87      state <= S9;
88  end
89  S8:
90  begin
91      result<=1;
92      state <= S9;
93  end
94  S9:
95  begin
96      begin
97      end
98  end
99  endcase
100 endmodule
101
```

Figure 4.20: Verilog code for isprime created by the tool

4.0.5.4. Running the verilog code generated by the tool

The running of verilog code in Quartus 2 64 bit is shown. Prime control test for 12 Prime control test for 7

```

# k = 0      # k = 1      # k = 3      # k = 4      # k = 4      # k = 4
# ,i= 2      # ,i= 4      # ,i= 6      # ,i= 8      # ,i=10      # ,i=12
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 0  # ,state= 3  # ,state= 4  # ,state= 4  # ,state= 4  # ,state= 1
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 0      # k = 2      # k = 3      # k = 4      # k = 4      # k = 4
# ,i= 2      # ,i= 4      # ,i= 6      # ,i= 8      # ,i=10      # ,i=12
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 1  # ,state= 5  # ,state= 5  # ,state= 5  # ,state= 5  # ,state= 2
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 0      # k = 2      # k = 3      # k = 4      # k = 4      # k = 4
# ,i= 2      # ,i= 4      # ,i= 6      # ,i= 8      # ,i=10      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 2  # ,state= 1  # ,state= 1  # ,state= 1  # ,state= 1  # ,state= 3
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 0      # k = 2      # k = 3      # k = 4      # k = 4      # k = 5
# ,i= 3      # ,i= 4      # ,i= 6      # ,i= 8      # ,i=10      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 3  # ,state= 2  # ,state= 2  # ,state= 2  # ,state= 2  # ,state= 5
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 1      # k = 2      # k = 3      # k = 4      # k = 4      # k = 5
# ,i= 3      # ,i= 5      # ,i= 7      # ,i= 9      # ,i=11      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 5  # ,state= 3  # ,state= 3  # ,state= 4  # ,state= 4  # ,state= 1
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 1      # k = 3      # k = 4      # k = 4      # k = 4      # k = 5
# ,i= 3      # ,i= 5      # ,i= 7      # ,i= 9      # ,i=11      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 1  # ,state= 5  # ,state= 5  # ,state= 5  # ,state= 5  # ,state= 6
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 1      # k = 3      # k = 4      # k = 4      # k = 4      # k = 5
# ,i= 3      # ,i= 5      # ,i= 7      # ,i= 9      # ,i=11      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 2  # ,state= 1  # ,state= 1  # ,state= 1  # ,state= 1  # ,state= 7
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= x
#           #           #           #           #           #
# k = 1      # k = 3      # k = 4      # k = 4      # k = 4      # k = 5
# ,i= 4      # ,i= 5      # ,i= 7      # ,i= 9      # ,i=11      # ,i=13
# ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12 # ,number=12
# ,state= 3  # ,state= 2  # ,state= 2  # ,state= 2  # ,state= 2  # ,state= 9
# ,result= x # ,result= x # ,result= x # ,result= x # ,result= x # ,result= 0
#           #           #           #           #           #

```

Figure 4.21: If number is not prime verilog code simulation

# k = 0 # ,i= 2 # ,number= 7 # ,state= 0 # ,result= x # # k = 0 # ,i= 2 # ,number= 7 # ,state= 1 # ,result= x # # k = 0 # ,i= 2 # ,number= 7 # ,state= 2 # ,result= x # # k = 0 # ,i= 3 # ,number= 7 # ,state= 4 # ,result= x # # k = 0 # ,i= 3 # ,number= 7 # ,state= 5 # ,result= x # # k = 0 # ,i= 3 # ,number= 7 # ,state= 1 # ,result= x #	# k = 0 # ,i= 4 # ,number= 7 # ,state= 4 # ,result= x # # k = 0 # ,i= 4 # ,number= 7 # ,state= 5 # ,result= x # # k = 0 # ,i= 4 # ,number= 7 # ,state= 1 # ,result= x # # k = 0 # ,i= 4 # ,number= 7 # ,state= 2 # ,result= x # # k = 0 # ,i= 5 # ,number= 7 # ,state= 4 # ,result= x # # k = 0 # ,i= 5 # ,number= 7 # ,state= 5 # ,result= x #	# k = 0 # ,i= 5 # ,number= 7 # ,state= 2 # ,result= x # # k = 0 # ,i= 6 # ,number= 7 # ,state= 4 # ,result= x # # k = 0 # ,i= 6 # ,number= 7 # ,state= 5 # ,result= x # # k = 0 # ,i= 6 # ,number= 7 # ,state= 1 # ,result= x # # k = 0 # ,i= 6 # ,number= 7 # ,state= 2 # ,result= x # # k = 0 # ,i= 7 # ,number= 7 # ,state= 4 # ,result= x #	# k = 0 # ,i= 7 # ,number= 7 # ,state= 5 # ,result= x # # k = 0 # ,i= 7 # ,number= 7 # ,state= 1 # ,result= x # # k = 0 # ,i= 7 # ,number= 7 # ,state= 2 # ,result= x # # k = 0 # ,i= 8 # ,number= 7 # ,state= 3 # ,result= x # # k = 1 # ,i= 8 # ,number= 7 # ,state= 5 # ,result= x # # k = 1 # ,i= 8 # ,number= 7 # ,state= 6 # ,result= x #	# k = 1 # ,i= 8 # ,number= 7 # ,state= 8 # ,result= x # # k = 1 # ,i= 8 # ,number= 7 # ,state= 9 # ,result= 1 #
--	--	--	--	--

Figure 4.22: If number is prime verilog code simulation

4.0.6. GUI Test

Creating their own diagrams in the tool user interface.

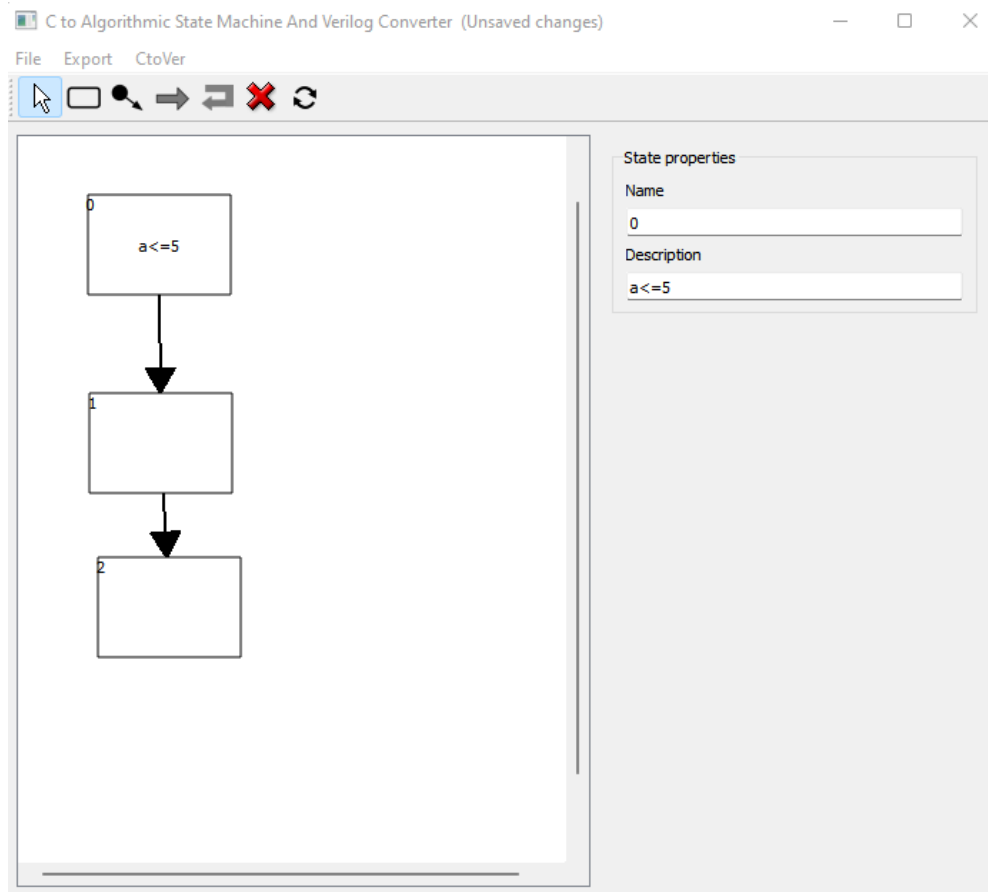


Figure 4.23: GUI displaying

4.0.6.1. Open C Code

The c code is taken from the relevant path.

```
1 void ComputePower ( int base , int *b )
2 {
3     int result ;
4     int exponent ;
5
6     exponent = 5 ;
7
8     result = 1 ;
9
10    while ( exponent != 0 ) {
11        result = result * base ;
12
13        exponent = exponent - 1 ;
14    }
15
16
17    b = result ;
18 }
19
```

Figure 4.24: Open C code file

4.0.6.2. State Diagram in GUI

Parsing the c code and placing it on the diagram.

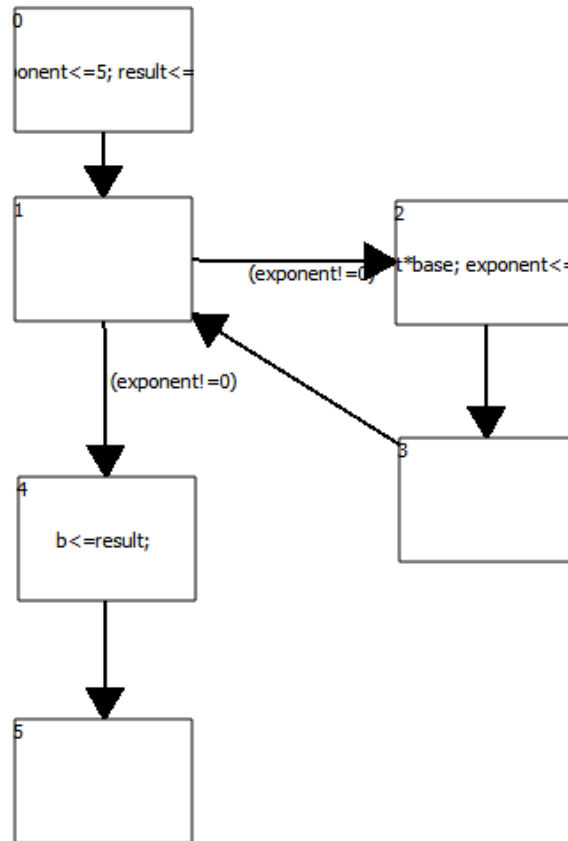


Figure 4.25: Display state diagram

4.0.6.3. Verilog Code in GUI

Conversion of C code to verilog code.

```
module ComputePower(base,b,clk);
input clk;
reg[31:0]state;
input[31:0]base;
output[31:0]b;
reg[31:0]b;

localparam
    S0 = 0,
    S1 = 1,
    S2 = 2,
    S3 = 3,
    S4 = 4,
    S5 = 5;

reg [31:0] result;
reg [31:0] exponent;
initial begin
state <= 0 ;
result <= 0;
exponent <= 0;
end

always @(posedge clk)
case(state)

S0:
begin
exponent<= 5 ;      state <= S1;result<= 1 ;
end
S1:
begin
if(exponent!=0)
begin
state <= S2;
end
else
begin
state <= S4;
end
end
S2:
begin
result<=result*base;
exponent<=exponent-1;

state <= S3;
end
S3:
begin

state <= S1;
end
S4:
begin
b<= result ; state <= S5;
end
S5:
begin

end
endcase
endmodule
```

Figure 4.26: Create Verilog Code

4.0.6.4. Edit State Diagram in GUI

Editing the diagram and converting it back to verilog code

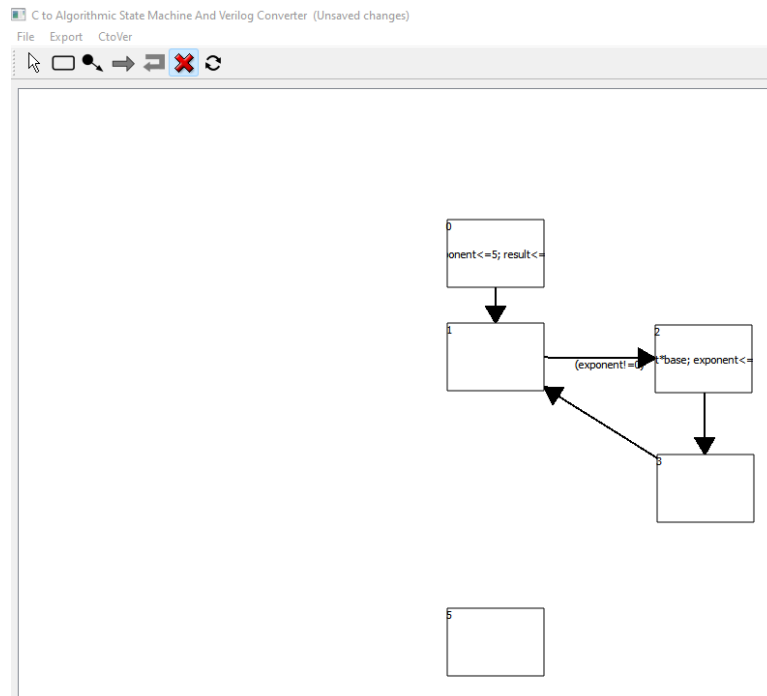


Figure 4.27: Edit State Diagram (Delete Some State)

4. RESTRICTIONS

The c code that the tool will get has some restrictions.

- 1-)The code must be written in the syntax of the C language.
- 2-)The code must be a single function code. It should not contain main.
- 3-)More than one function should not be defined
- 4-)Function should not be nested function and also function should not be recursive
- 5-)The return type of the function must be void
- 6-)Arrays should not be defined in function parameters.
- 7-)The type of variables must be integer only.
- 8-)Different increment variable should be defined for each cycle. (i,k,j...)
- 9-)Assignments should be at the top of the code.
- 10-)There must be a space after each expression.
- 11-) For Outwire, the function must have a result parameter in the form of int *b.

```
1  void diziTopla ( int *b )
2  {
3      int toplam ;
4      int i ;
5      int A[3] ;
6      A[0] = 6 ;
7      A[1] = 18 ;
8      A[2] = 44 ;
9      toplam = 0 ;
10
11     for ( i = 0 ; i < 3 ; i++ )
12     {
13         toplam = toplam + A[i] ;
14     }
15
16     b = toplam ;
17 }
18
```

Figure 4.28: Sample code

4. WORK DONE BY OTHER PEOPLE

An LLVM based mini-C to Verilog High-level Synthesis tool

In this project, they are proposing an LLVM based mini-C to Verilog high-level synthesis (HLS) tool, called c-ll-verilog (C to Verilog). HLS is a trending topic in both the academia and industry which aims to reduce the complexity of hardware design by automatically generating hardware description language (HDL) code from a higher level language such as C. An important difference between programming languages like C and HDLs is that, C and similar languages are meant to be used on a predefined architectures, which usually are conventional von neumann architectures, while HDL itself (in the domain of FPGA programming) is meant to model and design the underlying architecture or hardware. There are different HLS tools which their generated HDL code can be implemented on the latest field programmable gate array (FPGA) or system-on-chips. However, due to the inherent syntax and semantic differences in general purpose high level languages such as C and HDLs and the complexity of this conversion, current HLS tools cannot yet satisfy all of the performance, area, and power requirements of the complex systems. Therefore, in the view of the author these tools are either used in a highly annotated manner (for e.g. using PRAGMA or compiler directives) for a specific FPGA device and application or used for experimental as well as educational purposes. In this project we investigated the HLS process using an LLVM based framework, and by compiling a subset of branchless C statements to dataflow modeling Verilog code. In another project called An LLVM based mini-C to Verilog High-level Synthesis tool, conversion from c code to verilog code was attempted. The difference from our project is that it is independent of the state diagram. [11].

4. SUCCESS CRITERIA

At the beginning of the project, we determined the following success criteria.

The resulting diagrams will be the closest to the manually made Asm diagrams.

A procedure in Verilog corresponds to the same context that a function in C programming does. It is a particular block of statements called procedural statements. If we compare it with the high-level language, it comes out to be that the function arguments and parameters in a language like C is the same as that of the procedural statements. After the conversion, the procedural expressions will also be where they should be in the verilog code.

Translation of Verilog code closest to executable.

By the end of the project, it seems that the specified success criteria have been done.

4. CONCLUSIONS

As a result, as seen in the tests, if the given c code is written according to the rules, there is no problem in the creation of Asm and verilog code. However, in some c codes that may come, there may be situations where the verilog code does not work despite the conversion process. For example, the array that comes as an input parameter is not used as a module parameter in the verilog code. The codes with arrays can only work without errors if they are assigned inside the function. This is due to some fundamental differences between the two languages. But it can still be made workable with some minor changes on the transformed code. These number of changes determine the success criteria of the project. In my opinion, if the rules are followed, the project has become useful in the world of education and work.

BIBLIOGRAPHY

- [1] J. Long and R. Brayton, “A simple c to verilog compilation procedure for hardware/software verification),” *TUGBoat*, vol. c2v, no. 2, pp. 14–26, 2016.
- [2] Altera Corporation. “Implementing state machines verilog hdl.” (2020), [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vlog/vlog_pro_state_machines.htm.
- [3] Educba. “Draw diagram.” (), [Online]. Available: <https://www.educba.com/opencv-rectangle/>.
- [4] Verilogguide. “Implementing state machines verilog hdl.” (), [Online]. Available: <https://verilogguide.readthedocs.io/en/latest/verilog/fsm.html>.
- [5] Digilent. “Implementing state machines verilog hdl.” (), [Online]. Available: <https://digilent.com/blog/how-to-code-a-state-machine-in-verilog/>.
- [6] Allaboutcircuits. “Creating finite state machines in verilog.” (), [Online]. Available: <https://www.allaboutcircuits.com/technical-articles/creating-finite-state-machines-in-verilog/>.
- [7] IBM. “C/c++ parser.” (), [Online]. Available: <https://www.ibm.com/docs/en/developer-for-zos/9.1.1?topic=files-cc-parser>.
- [8] Qt Community. “Qt state machine framework.” (), [Online]. Available: <https://doc.qt.io/qt-5/statemachine-api.html>.
- [9] Zosrothko. “Qfsm is a graphical tool for designing finite state machine.” (), [Online]. Available: <https://github.com/Kampbell/qfsm>.
- [10] Programiz. “Simple c code example.” (), [Online]. Available: <https://www.programiz.com/c-programming/examples/prime-number>.
- [11] Sabbagham. “C-ll-verilog.” (), [Online]. Available: <https://github.com/sabbagham/c-ll-verilog>.