Chapter 8
Simulators for Quantum Circuits

The goal of developing quantum circuit simulators in our research work is to verify the operational logic of a given quantum circuit and synthesize the verified circuit in terms of nearest neighbor configuration. In general a quantum circuit simulator helps in the development of quantum computing activities in the following ways:

- It provides an environment to describe a given quantum circuit in terms of the basic quantum gates.
- It provides means to perform a set of quantum operations defined by the given circuit on a quantum register as well as obtain the final state of the register after successive evolution, in a virtual environment.
- Logic verification of the quantum circuit can be done efficiently.
- The state of the system can be observed at the intermediate steps of a computation, which cannot be done in a real quantum mechanical system.
- The library of predefined circuits can be utilized to build and test larger circuits.

In this chapter, Section 8.1 gives a brief review of the existing Quantum Circuit Simulators. The basic features and the design methodology of the HDL based quantum circuit simulator are presented in Section 8.2. Section 8.3 discusses the features and design methodology of the GUI based quantum circuit simulator and concluding remarks in Section 8.4.

8.1 Existing Simulators

A number of quantum simulators exist that vary in complexity, purpose, state representation and implementation. The study of quantum simulation began when Deutsch (1985) [2] introduced the notion of a Quantum Turing Machine (QTM). Many QTM simulators have been implemented including the Quantum Turing Machine Simulator (QTS) developed by Hertel (1999) [139] in a Mathematica environment.

Most quantum simulators use complex numbers to represent quantum states. Other approaches have used Quantum Decision Diagram (QDD). A C++ library developed by
Greve (1999) [140] uses binary states which are represented by Binary Decision Diagrams (BDD). This allows QDD to model relatively large quantum states although this feature limits QDD to representing a digital quantum computing model only as opposed to an analog model.

Quantum Bayesian Nets are another common representation of quantum states and are used by Quantum Fog [141] and Qubiter (Tucci 1998) [142]. Using Bayesian Nets, quantum system can be represented graphically. Quantum computing language developed by Omer (1998) [143] was the first architecture-independent programming language for quantum computers. It is a quantum computer simulation language designed to work with any qubit-based quantum architecture. It is useful in studying quantum-computing theory but cannot capture hardware-specific phenomena.

The Parallel Quantum Simulator, developed by Obenland and Despain (1997) [144] was specially designed to examine the effect of errors during quantum computation. Most quantum computing simulators are designed to simulate a single algorithm on a single type of hardware, most commonly Shor’s quantum factoring algorithm and the algorithms needed to implement it.

QuaSi is a general purpose quantum circuit simulator written at the University of Karlsruhe [145]. The user is able to build and simulate quantum circuits in a graphical user interface.

8.2 QHDL based Simulator

In digital hardware design, a hardware description language or HDL [146] is any language from a class of computer languages for formal description of hardware circuits. For a given circuit, its design can be described by the language, and its operation can be verified by means of simulation. An HDL simulation program provides the hardware designer with the ability to model a piece of hardware before it is created physically.

In this research work the design and development of a Quantum Hardware Description Language (QHDL) based tool for simulation of quantum computing circuits have been taken up. A QHDL parser has been designed which can take the input in a hardware description language format and can evaluate the operations accordingly. The HDL is capable of performing single gate level operations for desired inputs by
generating the transfer matrix for the corresponding gate operations. A user can make a
design entry of a given quantum circuit by means of the QHDL editor, and can provide
the desired input combination as well. The output is the equivalent transfer matrix
corresponding to a composition of all the gate operations of the desired quantum circuit,
and the input state matrix. The key features of this simulator can be listed as:

- Gate level simulation
- A circuit level simulation
- Generation of transfer matrix
- Generation of output matrix
- Simulation of error performance

The specialty of this simulator compared to the existing ones is that it can simulate the
error performance of a given quantum circuit according to the quantum circuit error
model as described in Chapter 6.

8.2.1 Design of the QHDL Simulator

Figure 8.2.1 shows the basic operations performed by the simulator. The simulation
involves (i) parsing of the HDL text description input to retrieve the gates from the
syntax and also the input, (ii) formation of the input matrix from the input string, (iii)
calculation of the tensor product to produce an equivalent unitary matrix for each level,
and then the vector product of all these matrices to generate the circuit matrix, and finally
(iv) multiplication of the circuit matrix and the input matrix to report the output matrix.
8.2.2 Syntax Rules for Circuit Simulation:

As a quantum circuit has a number of qubit lines and circuit levels, the level at which the specific gate is placed and also the qubit lines on which the gate works need to be specified in the QHDL entry.
The syntax rules for the simulator are enumerated below.

1. The syntax of the different gates in the library are:
   - NOT gate: not(level number, qubit line number)
   - HADAMARD gate: qhadamard(level number, qubit line number)
   - CNOT gate: cot(level number, control qubit line number, target qubit line number)
   - SWAP gate: qswap(level number, qubit line number1, qubit line number2)
   - TOFFOLI gate: qtoffoli(level number, control1 qubit line number, control2 qubit line number, target qubit line number)
   - Controlled-SWAP gate (Fredkin): qcoswap(level number, qubit line number1, qubit line number2, qubit line number3)

2. Different gates at any particular level are delimited by “;”

3. Different levels are delimited by “:”

4. All the gates in same level are to be given consecutively before the delimiter sign for that level.

5. An empty gate at any qubit line is represented by ( ).
An example of a quantum circuit is shown below in the Figure 8.2.2. It has 4 qubit lines (q_1 to q_4) and 3 levels (L_1 to L_3), and is formed of various gates occurring at different levels. For our simulator, this circuit is represented with the following syntax:

\[ \text{not}(1,1); (); (); \text{cnot}(2,2,3); (); \text{qhadamard}(2,2); \text{qswap}(3,3,4); \]

The matrix for a level is formed by putting identity matrices at all the empty positions as shown in the Figure 8.2.3 and then computing the tensor product of the all the gates at a particular level. Having determined these matrices for all the levels, multiplying all of them in reverse order results in the matrix for the entire circuit. A single qubit input (either 0 or 1) is to be entered for each level and the input matrix is formed by putting a 1 at the position where combination of the inputs occur in a \(2^n\) by 1 matrix, \(n\) being the number of levels, and rest of the entries are 0. The circuit matrix and input matrix are multiplied in order to get the output matrix.

Figure 8.2.2: An example quantum circuit
8.2.3 Implementation

The salient features of this simulator are user-friendly interface and definition of a hardware description language for quantum computations, built on to the math. For the HDL portion JAVA was a feasible choice. The basic quantum gates in the library are directly stored as class files, so that whenever computation is required involving that particular gate, the corresponding class file needs to be called.

Advantages of using JAVA:

- Any Operating System can have JAVA, as it is a platform independent and object oriented programming language
- JAVA can create good user interface through applet programming, and this technique is used to create the editor for QHDL which collects the input from the user and prints the desired output.
- JAVA functions can calculate the matrix programs efficiently.

8.2.4 Basic Features of the QHDL Simulator

Figure 8.2.4 shows the basic steps of the simulator. The software modules for these steps are the following:

INPUT: User specifies the input in QHDL syntax
SYNTAX CHECKING:

grammar checking:

- At the end of every level, there should be a ":", otherwise an exception is thrown.
- If a gate is missing at a level, an exception is thrown.
- If user specifies an illegal gate name, an exception is thrown.

Input Label Checking:

The input in QHDL has to be given in order of the qubit lines of a gate, or according to the levels of the circuit.

PARSER:

- Gates are recognized from the given QHDL input by delimiting every gate by "".
- For each gate name, the pattern is matched with those in Circuit.java to obtain the predefined matrix.

CIRCUIT PROCESSOR:

- After getting sequence of the matrices for the levels, calculation is done by multiplying from right to left to produce the transfer matrix for the circuit.
- The user specified input stream is converted to an input matrix by using tensor product.

OUTPUT:

After getting the transfer matrix and the input matrix it is been multiplied to generate the final output matrix.
Figure 8.2.4 Basic execution steps of the simulator
8.2.5 Simulation Results

Hadamard Gate:

Here \texttt{qhadamard(1,1)}: is the syntax to represent the HADAMARD gate. The first \texttt{l} represents the gate level and the second \texttt{l} the qubit line on which the gate acts. Here the given input is 1. The desired output value is shown in Figure 8.2.5. .

![Quantum Editor](image)

Figure 8.2.5: Simulation Results for Hadamard gate
Controlled NOT Gate:

Here $\text{qcnot}(1, 1, 2) : () :$ is the syntax to represent the $CNOT$ gate. The first $1$ represents the gate level, the second and third values $1, 2$ specify that this $CNOT$ gate is between the first and second qubit lines. Here the given input is 01. The desired output value is shown in Figure 8.2.6.

![Quantum Editor](image-url)

Figure 8.2.6: Simulation Results for CNOT gate
Toffoli Gate ($C^2$NOT):

Here $\text{qtoffoli}(1,1,2,3):():():$ is the syntax to represent $C^2$NOT gate. The first $/$ represents the gate level, the second to fourth values $1,2,3$ specify that this 3-input Toffoli gate is on the first, second and third qubit lines. For the given input is 001, the desired output value is shown in Figure 8.2.7.

Figure 8.2.7: Simulation Results for Toffoli gate
An example circuit as shown in Figure 8.2.8 is simulated. The given input is 01. The desired output value as shown in Figure 8.2.9.

![Figure 8.2.8: An example circuit for simulation](image)

![Figure 8.2.9: Simulation results for the circuit shown in Figure 8.2.8](image)
8.2.6 Error Simulation

The error modeling involves both the bit flip and the phase flip error, which are the two most common errors in the context to quantum computing circuits. The Pauli matrices X and Z are used for this purpose. $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ for bit flip, and $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ for phase flip, the definitions of these appear in Chapter 7. An error $e_i$ on a qubit line $q_i$ is denoted by a 4-tuple $e_i: \langle \text{bitflip}, l_1, l_2, q_i \rangle$ consisting of its type, the indexes of the two levels previous and post between which $q_i$ lies. The circuit synthesis for the error is done by placing the X or the Z gates in the particular qubit wires of the circuit as specified in the error description. The output state is then computed as outlined above, in Section 8.2.1., by re-evaluating the circuit matrix incorporating the newly inserted X or Z gates.

We induce the bit flip and the phase flip errors for an example circuit shown in the Figure 8.2.10, through the QHDL for error specification.

![Figure 8.2.10: The circuit with X and Z errors and the QHDL for error specification](image)

8.3 Quantum Circuit Simulator with GUI

In this section a quantum circuit simulator with GUI, named as Qubit Quantum Circuit Simulator (QQCSim) is introduced. It performs gate level and circuit level operations of quantum circuits. It is implemented with MS Visual Basic and MS-Access. A circuit can be constructed with any valid combination of the quantum logic gates, and it can be simulated with the inputs supplied by the user to compute the corresponding output matrix. There are several options to simplify the user operations. A circuit can be saved and retrieved for the latter use. The main feature of this simulator is that it also
provides the option for nearest neighbor synthesis using \textit{CNOT} and \textit{C}^2\textit{NOT} gates as per the methods reported in Chapter 6.

The key features of the \textit{QQCSim} are:

- Gate level simulation
- Circuit level simulation up to 20 qubit lines
- Intermediate gate level corrections can be made
- Nearest neighbor synthesis option for CNOT & \textit{C}^2\textit{NOT} gates
- Generation of output state matrix
- Generation of the circuit transform matrix
- A circuit can be stored for re-use

8.3.1 Advantages of using Visual Basic as the Application Platform:

- Any PC with MS Windows and a VB500.DLL can run VB applications. This is the chosen platform for about 75\% of the application software existing in the world.
- The VB Debugger is a powerful and elegant tool which helps a programmer to detect semantic errors quickly by single-stepping through a program. This leads to better quality applications.
- Visual Basic supports rapid prototyping with realistic interfaces, as well as its ability to provide a powerful GUI development experience to new developers.

8.3.2 Bells and Whistles of \textit{QQCSim}

8.3.2.1 System Requirements

In order to run the \textit{QQCSim}, the Visual Basic runtime libraries provided by Microsoft are required.

8.3.2.2 Circuit Management

The toolbar as shown in Figure 8.3.1 provides basic circuit management functionality. Details of the toolbar buttons appear in Table 8.3.1.
Figure 8.3.1: Toolbar of the Simulator QQCSim

Table 8.3.1: Detailed Functionality of the Toolbar Buttons of QQCSim

<table>
<thead>
<tr>
<th>BUTTONS</th>
<th>FUNCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Bit</td>
<td>Adds a bit to the circuit.</td>
</tr>
<tr>
<td>Add Gate</td>
<td>Adds a level of gate to the circuit.</td>
</tr>
<tr>
<td>Save Transfer Matrix</td>
<td>Saves the transfer matrix of the current circuit to either Matrix A or Matrix B.</td>
</tr>
<tr>
<td>for Comparison</td>
<td></td>
</tr>
<tr>
<td>Show Output Vector</td>
<td>Displays the matrix of the output vector.</td>
</tr>
<tr>
<td>Clear Circuit</td>
<td>Clears the current circuit which has been constructed with the GUI based editor of QQCSim</td>
</tr>
<tr>
<td>Simulate</td>
<td>Evaluates the circuit fed to QQCSim</td>
</tr>
<tr>
<td>Remove Bit</td>
<td>Removes a qubit from the circuit.</td>
</tr>
<tr>
<td>Remove Gate</td>
<td>Removes a level of gates from the circuit.</td>
</tr>
<tr>
<td>Show Transfer Matrix</td>
<td>Displays the transfer matrix of the entire circuit.</td>
</tr>
<tr>
<td>Show Equivalence Program</td>
<td>Displays a Visual basic form which compares two transfer matrices, which have been stored earlier, and determines whether these two circuits are equivalent.</td>
</tr>
<tr>
<td>Save</td>
<td>Stores a created circuit for latter use.</td>
</tr>
<tr>
<td>Open</td>
<td>Reloads a previously stored circuit.</td>
</tr>
<tr>
<td>Add Intermediate Gate Level</td>
<td>Inserts a new intermediate gate level after creation of a circuit</td>
</tr>
<tr>
<td>Remove Intermediate Gate Level</td>
<td>Deletes an existing intermediate gate level after creation of a circuit</td>
</tr>
<tr>
<td>Nearest Neighbour Synthesis</td>
<td>Apply rules of Nearest Neighbour Synthesis to the given circuit</td>
</tr>
<tr>
<td>Help</td>
<td>Opens a help file which assists the user to operate the simulator.</td>
</tr>
</tbody>
</table>
8.3.2.3 Configuring QQCSim

An input qubit line can be assigned either a $|0\rangle$ or a $|1\rangle$. If any superposed input is required, a line is initialized with either $|0\rangle$ or $|1\rangle$ and operated on by a Hadamard gate. The input can be changed by simply clicking on the qubit values. Due to practical limitations, the simulator QQCSim has an upper bound on the number of input qubit lines permitted for a circuit. This limit is such that with base 2 raised to the power of the number of inputs is less than 1040. As $2^{10}$ is the largest integer satisfying the constraint, upto 10 input qubits are allowed for base 2. Figure 8.3.2 shows a view of the simulation window.

![Figure 8.3.2: A view of the simulation window of QQCSim](image)

Figure 8.3.2: A view of the simulation window of QQCSim
8.3.2.4 Transfer Matrix

The transfer matrix of a circuit is a unitary matrix, which represents the equivalent matrix operator for the entire circuit. Once \textit{QQCSim} completes evaluation for a given circuit, one can click on 'Show Transfer Matrix'. A new Visual Basic form appears showing the transfer matrix. Each value is represented by a real part and a complex part. It is not unusual for many of the values to be of zero value. Figure 8.3.3 gives the view of a window showing the transfer matrix output.

![Transfer Matrix Output](image)

Figure 8.3.3: The view of a window showing the transfer matrix output in \textit{QQCSim}
8.3.2.5 Constructing a circuit in QQCSim

The number of inputs to the circuit can be changed by clicking ‘add bit’ or ‘remove bit’ buttons on the options menu or the buttons above the circuit. Similarly, the number of gates can be increased or decreased by clicking ‘add gate’ or ‘remove gate’. There is no limit on the number of gates a circuit can have.

Building the circuit is simply a matter of clicking on one of the gates in the library on the right hand pane and dragging it over to the appropriate gate location on the circuit.

8.3.2.6. Simulation

Once the circuit has been designed and the inputs have been specified, the next task is simulation. This is simply a matter of clicking the simulate button. The output of the circuit is provided by QQCSim at the top of the screen towards the right, as shown in Figure 8.3.4.

![Screenshot of Simulation results produced by QQCSim](image)

Figure 8.3.4: Screenshot of Simulation results produced by QQCSim
8.3.2.7 Circuit Equivalence

Circuit equivalence is essentially comparison of two circuits. The simulator QQCSim has this feature. After simulating a given circuit, the ‘Save Transfer Matrix for Comparing’ option is to be clicked; there is a choice of storing the transfer matrix as either matrix A or matrix B, as shown in Figure 8.3.5. For comparison with another circuit which is to be designed, the same steps as that for the earlier one need to be followed, i.e., construct, simulate and save it as second matrix.

![QQCSim](image)

Figure 8.3.5: Saving of the transfer matrix for equivalence checking in QQCSim

In order to test the equivalence ‘Show Equivalence Program’ tab needs to be invoked and then the ‘Compare Matrices’ option. A sample output for testing the equivalence is as shown in Figure 8.3.6.

![Circuit Equivalence Form](image)

Figure 8.3.6: Results for Circuit Equivalence Checking in QQCSim
8.3.2.8 Save and Open Option

On clicking the SAVE button; the user can save the circuit design. In future, it can be opened and the user can re-design or modify the circuit by clicking the Open submenu in the File menu. The window for opening a pre-designed circuit is shown in Figure 8.3.7.

![Figure 8.3.7: Window for opening a pre-designed circuit in QQCSim](image)

8.3.2.9 Removal of an Intermediate Gate

User can delete an undesired gate level at the circuit design time by clicking the Remove Intermediate Gate button in the main window, the screenshot for the removal operation is shown in Figure 8.3.8 and the circuit after the removal of the intermediate gate is shown in Figure 8.3.9.
Figure 8.3.8: Screenshot for the removal of intermediate gate level in QQCSim

Figure 8.3.9: Circuit after the removal of intermediate gate level in QQCSim
8.3.2.10 Add an Intermediate Gate Level

The user can add an intermediate gate level in an existing circuit of *QQCSim*. Figure 8.3.10 shows the screenshot for the operation of adding an intermediate gate level in the circuit and Figure 8.3.11 the circuit after the addition of the gate level.

![Figure 8.3.10: The screenshot for the operation of adding an intermediate gate level](image1)

![Figure 8.3.11: The circuit after addition of an intermediate gate level](image2)
8.3.3 Nearest Neighbor Synthesis using CNOT Gates

It is a feature of the simulator which makes it a very unique from the other existing quantum circuit simulators. The synthesis technique has been adopted from our work [147], which describes the synthesis of quantum circuits using nearest neighbor templates of CNOT and C^2 NOT gates. The user can apply the nearest neighbor synthesis techniques in the simulator QQCSim. The total number gates required for nearest neighbor synthesis is created dynamically. Figure 8.3.12 shows an initial circuit built in QQCSim and Figure 8.3.13 shows the nearest neighbor synthesis result for the same circuit.

Figure 8.3.12: An example quantum circuit
8.3.4 Nearest Neighbor Synthesis using Toffoli Gates

In the previous sub-section, nearest neighbor synthesis in \textit{QQCSim} for a circuit using a CNOT gate was demonstrated. This section demonstrates the nearest neighbor synthesis of a quantum circuit having Toffoli (C$^2$NOT) gates using our simulator \textit{QQCSim}. Figure 8.3.14 shows the initial circuit and Figure 8.3.15 shows the resulting circuit after nearest neighbor synthesis rules are applied.
Figure 8.3.14: The initial circuit

Figure 8.3.15: The nearest neighbor circuit produced by QQCSim
8.4 Conclusion

In this chapter, the design and development of two distinct software packages for simulating quantum circuits using a well-defined quantum gate library are described. Although the scheme for the formation of the input quantum state, the quantum circuit matrix and the output quantum state evaluation is same for both the simulators, the key difference lies in terms of the design entry; one (QHDL) follows text based design entry and the other is based on a GUI. The specialty of the QHDL based simulator is that apart from evaluating the output quantum state for a given quantum circuit corresponding to a given input state, it can also evaluate the error performance of the given quantum circuit for the different values of error probability, as per the method proposed in Chapter 7. The GUI based simulator \textit{QQCSim} not only simulates a quantum circuit at the gate-level but also can transform its nearest neighbor equivalent based on the template-based rules devised in Chapter 6.