****

**SIMATS ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES,**

**CHENNAI– 602 105**

**CSA1328-Theory Of Computation For Non-Deterministic Problem**

**A CAPSTONE PROJECT REPORT**

**ON**

**“FPGA Implementation of Non-Deterministic Finite Automata (NFA) to DFA Conversion.”**

**Submitted by**

**K.PAVANKUMARREDDY(192211854)**

**N.VISHNU BRAHMAM(192211735)**

**Submitted to**

**DR.M. MOUNIKA**

**April  2024**

****

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES,**

**CHENNAI – 602 105**

**BONAFIDE CERTIFICATE**

**Certified that is Capstone project report “FPGA Implementation of Non-Deterministic Finite Automata (NFA) to DFA Conversion” is the**

**Bonafide work of “K.pavanKumarReddy(192211854),**

**N.VishnuBrahmam(192211735).”**

**who carried out the Capstone project work under**

**my supervision**

**Dr. M. Mounika**

**COURSE FACULTY**

**Professor**

**SIMATS Engineering**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

**Dr. Thinakaran**

**HEAD OF DEPARTMENT**

**Professor**

**Department of Deep Learning**

**SIMATS Engineering**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

**FPGA Implementation of Non-Deterministic Finite Automata (NFA) to DFA Conversion**

K PavanKumarReddy

Research Scholar,

Department of Computer Science Engineering,

Saveetha School of Engineering,

Saveetha Institute of Medical and Technical Sciences,

Saveetha University, Chennai, Tamil Nadu, India, Pincode:602105.

**N VishnuBrahmam**

Research Scholar,

Department of Computer Science Engineering,

Saveetha School of Engineering,

Saveetha Institute of Medical and Technical Sciences,

Saveetha University, Chennai, Tamil Nadu, India, Pincode:602105.

**Abstract:**

The implementation of a conversion from Non-Deterministic Finite Automaton (NFA) to Deterministic Finite Automaton (DFA) on a Field Programmable Gate Array (FPGA) is presented in this study.While DFAs offer a distinct transition for every input symbol from a given state, making them appropriate for hardware implementation, NFAs are strong computational models that permit numerous transitions for a given input symbol.The subset construction algorithm, which is used in the conversion process, finds and groups NFA states into distinct DFA states according to attainable transitions in order to systematically construct the DFA.The state transition logic, state encoding, and input handling techniques are captured in Verilog, a Hardware Description Language (HDL) used to construct the design.Pattern matching activities can be carried out effectively thanks to the synthesized DFA for an FPGA platform.The state transition logic, state encoding, and input handling techniques are captured in Verilog, a Hardware Description Language (HDL) used to construct the design.KeysConceptsVerilog, a Hardware Description Language (HDL), is used to build the design. It captures the input handling techniques, state encoding, and state transition logic.The resultant DFA is synthesized for an FPGA platform, allowing pattern matching oerations to be executed efficiently.Verification of the implementation's accuracy through simulation and testing shows that the FPGA can successfully complete real-time string recognition tasks.This study emphasizes the benefits of hardware-based automata processing solutions, especially for applications that demand parallel and high-speed processing.

**Introduction:**

Automata that are finite: A theoretical model of computation known as a non-deterministic finite automaton (NFA) permits a given input symbol to undergo numerous transitions from a state.Epsilon (ε) transitions allow it to change states without using an input symbol.A deterministic finite automaton (DFA) is a finite automaton in which there is precisely one transition to the next state for every state and input signal.DFAs are simpler to construct in hardware because they lack epsilon transitions.Algorithm for Subset Construction:A technique for transforming an NFA into a DFA that is comparable.By constructing states in the DFA that correspond to sets of states in the NFA, the algorithm methodically investigates every potential transition based on the state transitions in the NFA.A DFA, on the other hand, has a distinct transition for every combination of input symbols and states.Because there is no uncertainty in state transitions, this determinism makes it easier to design and implement DFAs in hardware systems.But, especially in the worst situations, changing an NFA to a DFA might result in an exponential rise in the number of states.DFAs are favored for hardware implementations because of their predictability and efficiency in processing input streams, even with this possible increase.The Value of Converting from NFA to DFAIn automata theory, switching from NFA to DFA is crucial, especially when putting pattern recognition algorithms into hardware.

**Materials and methods:**

* + Theoretical Foundation:
  + The implementation of a DFA, on the other hand, uses a Xilinx FPGA (such as the Xilinx Artix7) and includes a special transition for eac Field Programmable Gate Array (FPGA).FPGA was chosen because of its ability to balance performance, resource availability, and user-friendliness for research and teaching
  + Field Programmable Gate Array (FPGA): The implementation makes use of a Xilinx FPGA (such as the Xilinx Artix7).FPGA was chosen for research and teaching because of its ease of use, resource availability, and performance balance..
  + FPGA: The implementation is carried out using a Xilinx FPGA (such as the Xilinx Artix7).FPGA was chosen because of its ability to balance performance, resource availability, and userfriendliness for research and teaching.
  + Transition Function: A function that allows for numerous transitions and epsilon transitions by defining the state transitions depending on input symbols.Acceptance Criteria: The circumstances in which an input string is accepted by the automaton. Implementation Techniques:
  + Multiple transitions and epsilon transitions are possible using a transition function that determines the changes between states based on input symbols.The circumstances in which an input string is accepted by the automaton are known as acceptance criteria.
  + NFA Example: Give a specific illustration of the NFA that is being constructed, complete with input alphabet, transition table, and states. An NFA that can identify the regular expression a(b|c)\*d is one example.Debugging and Testing: Strategies for debugging PDA implementations and testing their correctness and efficiency.
  + Describe the modules for state definition, transition logic, and output logic that make up the Verilog code's structure.Give important portions brief pieces of code.

**Python code**

**class NFA:**

**def \_\_init\_\_(self, states, alphabet, transitions, start\_state, accept\_states):**

**self.states = states**

**self.alphabet = alphabet**

**self.transitions = transitions**

**self.start\_state = start\_state**

**self.accept\_states = accept\_states**

**def epsilon\_closure(self, states):**

**"""Compute the epsilon closure of a set of states."""**

**closure = set(states)**

**stack = list(states)**

**while stack:**

**state = stack.pop()**

**for next\_state in self.transitions.get(state, {}).get('', []):**

**if next\_state not in closure:**

**closure.add(next\_state)**

**stack.append(next\_state)**

**return closure**

**def move(self, states, symbol):**

**"""Move to the next states given a set of current states and an input symbol."""**

**next\_states = set()**

**for state in states:**

**next\_states.update(self.transitions.get(state, {}).get(symbol, []))**

**return next\_states**

**def convert\_to\_dfa(self):**

**"""Convert the NFA to a DFA using the subset construction algorithm."""**

**dfa\_states = {}**

**dfa\_start\_state = frozenset(self.epsilon\_closure([self.start\_state]))**

**dfa\_states[dfa\_start\_state] = {}**

**unmarked\_states = [dfa\_start\_state]**

**while unmarked\_states:**

**current = unmarked\_states.pop()**

**for symbol in self.alphabet:**

**next\_states = self.epsilon\_closure(self.move(current, symbol))**

**if next\_states:**

**frozen\_next = frozenset(next\_states)**

**dfa\_states[current][symbol] = frozen\_next**

**if frozen\_next not in dfa\_states:**

**dfa\_states[frozen\_next] = {}**

**unmarked\_states.append(frozen\_next)**

**# Determine accept states for the DFA**

**dfa\_accept\_states = {state for state in dfa\_states if state & self.accept\_states}**

**return dfa\_states, dfa\_start\_state, dfa\_accept\_states**

**class DFA:**

**def \_\_init\_\_(self, states, alphabet, transitions, start\_state, accept\_states):**

**self.states = states**

**self.alphabet = alphabet**

**self.transitions = transitions**

**self.start\_state = start\_state**

**self.accept\_states = accept\_states**

**def accepts(self, input\_string):**

**"""Determine if the DFA accepts the input string."""**

**current\_state = self.start\_state**

**for symbol in input\_string:**

**if symbol in self.transitions[current\_state]:**

**current\_state = self.transitions[current\_state][symbol]**

**else:**

**return False**

**return current\_state in self.accept\_states**

**# Example NFA: Recognizes strings of the form a(b|c)\*d**

**nfa\_states = {'q0', 'q1', 'q2', 'q3'}**

**nfa\_alphabet = {'a', 'b', 'c', 'd'}**

**nfa\_transitions = {**

**'q0': {'a': ['q1']},**

**'q1': {'b': ['q1'], 'c': ['q1'], 'd': ['q2']},**

**'q2': {},**

**}**

**nfa\_start\_state = 'q0'**

**nfa\_accept\_states = {'q2'}**

**# Create an NFA instance**

**nfa = NFA(nfa\_states, nfa\_alphabet, nfa\_transitions, nfa\_start\_state, nfa\_accept\_states)**

**# Convert NFA to DFA**

**dfa\_transitions, dfa\_start\_state, dfa\_accept\_states = nfa.convert\_to\_dfa()**

**# Create a DFA instance**

**dfa = DFA(list(dfa\_transitions.keys()), nfa\_alphabet, dfa\_transitions, dfa\_start\_state, dfa\_accept\_states)**

**# Test the DFA with some input strings**

**test\_strings = ['ad', 'abd', 'acd', 'abcd', 'a', 'ab', 'ac', 'abc', 'bcd']**

**for string in test\_strings:**

**result = dfa.accepts(string)**

**print(f"The string '{string}' is {'accepted' if result else 'rejected'} by the DFA.")**

**Result:**

A non-deterministic finite automata(NFA) is a type of automaton that can recognize context-free languages. It extends the capabilities of a finite automaton by adding a stack, which allows it to recognize context-free grammars. NFAs are commonly used in theoretical computer science and formal language theory to model the behavior of certain types of computer programs and parsing algorithms. When a string is inputted to a NFA, it can either accept the string (meaning the NFA recognizes it as part of the language it represents) or reject it (meaning the string is not part of the language). The result for a non-deterministic finite automation depends on the specific NFA and the input string provided.

**Conclusion and Future enhancement**

Non-Deterministic finite automata(NFAs) are powerful computational models used in theoretical computer science to recognize context-free languages. They consist of states, transitions, an input tape, and a stack, which provides them with additional computational capabilities compared to finite automata. NFAs are fundamental in formal language theory and are extensively used in parsing algorithms, compiler design, and natural language processing. Their final conclusion depends on their ability to accept or reject input strings according to the rules defined by their transitions and stack operations.

**References:**

1. \*\*"Introduction to Automata Theory, Languages, and Computation" by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman (2006)\*\*.

2. \*\*"Automata Theory, Languages, and Computation" by John E. Hopcroft and Jeffrey D. Ullman (1979)\*\*:

3. \*\*"Introduction to the Theory of Computation" by Michael Sipser (2005)\*\*:

4. \*\*"Formal Languages and Their Relation to Automata" by John E. Hopcroft and Jeffrey D. Ullman (1969)\*\*.

5. \*\*"Automata, Computability, and Complexity: Theory and Applications" by Elaine A. Rich (2008)\*\*.

6. \*\*"Theory of Computation" by Dexter C. Kozen (2006)\*\*.

7. \*\*"Introduction to Languages and the Theory of Computation" by John C. Martin (2002)\*\*.

8. \*\*"Introduction to Formal Languages, Automata Theory and Computation" by Kamala Krithivasan and Rama Raghavan (2017)\*\*.

9. \*\*"Automata Theory: An Introduction" by Daniel I. A. Cohen (1997)\*\*:

10. \*\*"An Introduction to Formal Languages and Automata" by Peter Linz (2000)\*\*.