Q-Learning Accelerator Chip Design for Robot Path Planning

by

Harsh Shekhar Advani
202011049

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

May, 2022

Declaration

I hereby declare that

i) The thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) Due acknowledgment has been made in the text to all the reference material used.

HSadvani

Harsh Shekhar Advani

Certificate

This is to certify that the thesis work entitled Q-Learning Accelerator Chip Design for Robot Path Planning has been carried out by Harsh Shekhar Advani for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my supervision.

Tapas Kr. Maiti

Prof. Tapas Kumar Maiti
Thesis Supervisor

Acknowledgments

First and foremost, I want to convey my thanks and appreciation to my supervisor, Prof. Tapas Kumar Maiti, for his essential encouragement, recommendations, and support from the beginning of this project and for giving me with incredible experiences throughout the process. Above all, his invaluable and rigorous monitoring at every stage of the project inspired me in countless ways.

I thank him in particular for his guidance, supervision, and crucial contributions as needed during this project. His interest in creativity has sparked and nurtured my intellectual maturity, which will benefit me for a long time. I am honoured to have had the opportunity to work with such an accomplished Professor.

I appreciate the cooperation of the Dhirubhai Ambani Institute of Information and Communication Technology in Gandhinagar in completing this project. I'll never forget my buddy Jimmy Patel, with whom I went through difficult times, encouraged me on, and celebrated each achievement.

Finally, and most significantly, I'd want to express my gratitude to my family for their unwavering support and encouragement. I am grateful to my parents for their unwavering spiritual support and for affording me the opportunity to express myself and make my own life decisions. I'd want to express my gratitude to my other family members for their unwavering support and constructive criticism.

Contents

Abstract

A variety of frameworks and set of tools used for creating complex and difficult behaviours are provided by reinforcement learning in robotics. Q-learning is one of the most popularly used algorithms in reinforcement learning. The implementation of this algorithm is mostly done in MATLAB and Python using various tools and libraries available there. These types of implementations are software based only, which actually requires more time to provide the output. Another way is to implement the algorithm on hardware i.e., to implement it on Field Programmable Gate Arrays (FPGA) or to design a chip based on the applications. This way of implementation reduces the processing time and provides a faster simulation compared to the one done on software.

This thesis aims to reduce the processing time i.e., latency required for the agent to perform an action in any particular state while performing in an episode. We propose an efficient hardware architecture that incorporates the testing of the algorithm and provides a reduction in processing time. We also provided a chip design based on the proposed hardware architecture. This in turn will increase the performance and accuracy of the application. Thus, we majorly focus on the two parameters i.e., latency and accuracy.

List of Principal Symbols and Acronyms

α Learning Rate

ϵ Exploration factor

γ Discount Factor

π Policy

Q Quality

AI Artificial Intelligence

CAD Computer-Aided Design

CMOS Complementary Metal Oxide Semiconductor

CPU Central Processing Unit

DDPG Deep Deterministic Policy Gradient

DQN Deep Q Neural Network

DRC Design Rule Check

ECAD Electronic Computer-Aided Design

EDA Electronic Design Automation

ERC Electrical Rule Check

FPGA Field Programmable Gate Arrays

GNU GNU's Not UNIX

HDL Hardware Description Language

IC Integrated Circuit

LVS Layout vs Schematic

MCU Microcontroller Unit

MDP Markov Decision Process

ML Machine Learning

MOSIS Metal-Oxide-Semiconductor Implementation System

OS Operating System

RL Reinforcement Learning

RTL Register Transfer Level

SARSA State Action Reward State Action

SPICE Simulation Program with Integrated Circuit Emphasis

SRAM Static Random Access Memory

TD Temporal Difference

TSMC Taiwan Semiconductor Manufacturing Company Limited

VLSI Very Large-Scale Integration

List of Tables

List of Figures

CHAPTER 1
Introduction

In recent times, reinforcement learning (RL) is in much attention, with a large number of successful innovations in various fields. RL a type of machine learning is emerging as a sole computational intelligence through which the system can make decisions in a given environment without any pre-hand information. Artificial intelligence has a power to drive the cutting-edge technology innovations and allows to further extend the use of RL to a tremendous level.

Recent research efforts are done to improve/modify the algorithm so as to work with real-time applications, and also some efforts are made to implement the algorithm on hardware but is limited to the equation of Q-learning. Hence, we decided to implement entire Q-learning testing algorithm for robot control on hardware. This technology can be developed on hardware to reduce system processing time. When compared to software counterparts, it is also possible to achieve high performance not only in terms of processing time but also in terms of power consumption.

1.1 Overview of Artificial Intelligence

Artificial Intelligence is the intelligence shown by the machines, as opposite to the natural intelligence shown by humans and animals. According to John McCarthy, the pioneer of AI, "*The science and engineering of making intelligent machines, especially intelligent computer programs*" [1]. AI is basically used to mimic the cognitive functions that human brains perform such as solving problem, learning, etc [2].

In most cases, AI involves the creation of computer systems capable of doing activities that would typically necessitate human intelligence, i.e., speech recognition, decision making, visual perception and understanding different languages [3]. The above explanation provides general outline of AI, there is no such spe-

cific definition of AI. In general, AI is a field that consist of a broad spectrum of algorithms, technologies and applications.

1.2 About Machine Learning

A type of artificial intelligence called machine learning (ML), provides capability to machines to learn without being explicitly programmed [4], [5]. From the name, one can get an idea that it makes the computer more similar to humans by the ability to learn. Basic ML classification is depicted in Figure 1.1. ML is based on the idea that the machine can learn from data, identify patterns and then make decisions accordingly [6], [7].

1.2.1 Types of Machine Learning

Mainly, three classifications of ML can be done, based on the training of model i.e., Supervised, Unsupervised and Reinforcement Learning. An overview of these is explained below:

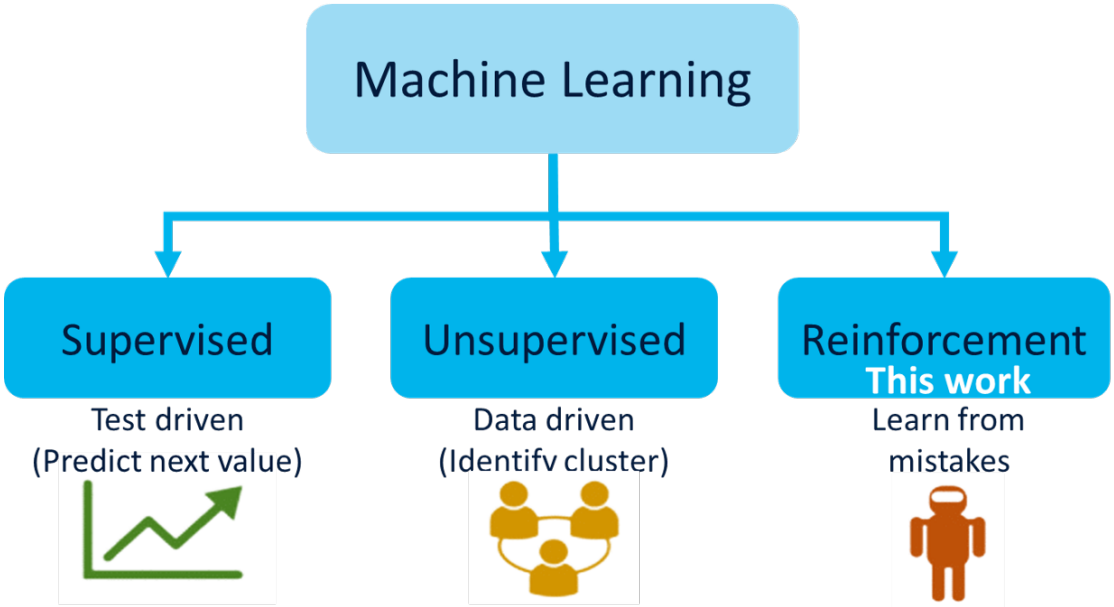

Figure 1.1: Classification of machine Learning (ML) based on training approaches.

1. Supervised Learning: Supervised ML is the most common method in AI. It is the learning in which the model is trained with labeled set of input data and its correlated output [7], [8], [9], [10]. For example, a dataset of vehicles as input data can be labeled as "Trucks" or "Others" as result. The model needs

to be continuously trained to provide accurate results. Once the training of model is done with input data set, it can be feed with additional data to obtain desired output.

2. Unsupervised Learning: Unsupervised ML is the learning in which the input data set is not labeled and output is also not specified [7], [11], [12]. A large data set is feed and the algorithm is expected to identify any hidden meaningful pattern. Results obtained are then analyzed by humans whether they are relevant or not.

3. Reinforcement Learning (RL): It is a type of ML that makes a series of decisions in a given situation [13], [14]. It is an algorithm which learns based on the observations of environment [15], [16], [17]. The advantage of using RL is that, it provides rewards that lead to success even when the environment is too large and complex. Our thesis work focuses on this.

1.3 Details of Reinforcement Learning

An agent can learn through reinforcement learning (RL), by exploring the environment and observing the results and rewards. The goal is to find the suitable action that would result in maximum reward [15], [16], [17]. In this technique there is no requirement to provide training examples, which is actually a large dataset and this is the main advantage of this technique. Real-time learning is possible in RL which implies that it can provide outcomes while improving at the same time. Figure 1.2 illustrates the action-reward feedback loop of a general RL model.

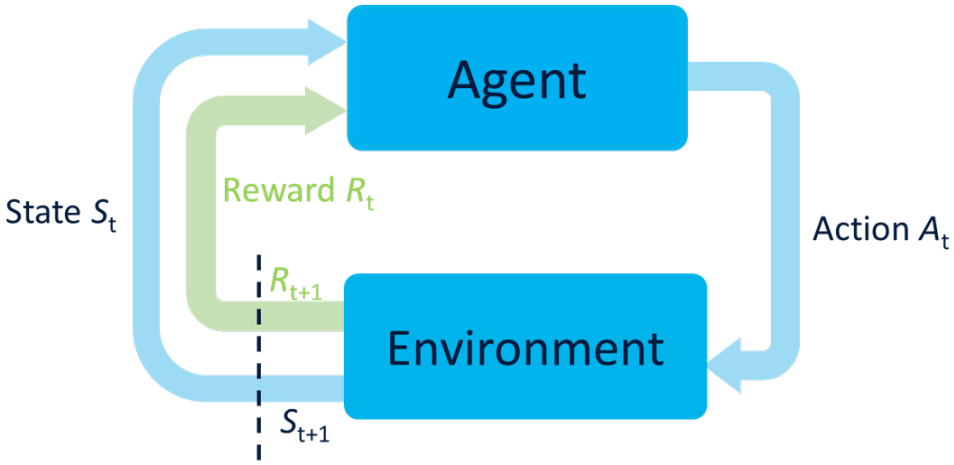

Figure 1.2: Shows the schematic diagram of action-reward feedback loop [14].

1.3.1 Flow of Reinforcement Learning

RL is the method of learning in which decisions are made using experiences. The process of RL can be stated as below:

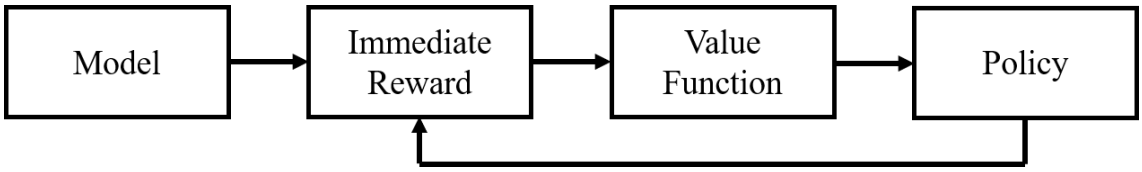

Figure 1.3: Block diagram of reinforcement learning system.

- Observing environment
- Decision making as per strategy
- Action
- Obtaining reward/penalty
- Using previous experience to improve the strategy
- Continue until the best strategy is obtained

1.3.2 Reinforcement Learning Approaches

There are three approaches with which RL can be implemented [15], [16], [17]:

1. Value Based: The value-based approach seeks to maximize value in a state that is governed by policy π. As a result, the agent anticipates long-term gains.
2. Policy-Based: In this approach an optimal policy is developed which can maximize the future rewards, unlike the value-based which employs the value function. There are two types of policy:
   (a) Deterministic: Probability of 1 is given to one action and remaining actions have 0 probability. Hence, the same action is repeated at any state.
   (b) Stochastic: All the actions will have different probabilities.
3. Model-Based: In a model-based strategy, there is no algorithm as the model representation for each environment is different. The model is created virtually for the environment.

1.3.3 Fundamental Elements of Reinforcement Learning

RL consists of four main elements, mentioned below:

1. Policy: A policy describes how the agent should act at all times. The agent's behavior can be defined by the policy [18]. Sometimes, it can be a function or a table, whereas in some cases it can have a computation used for searching.

2. Reward: For each state the agent receives some reward based on its action [19]. It can be positive or negative based on the action taken. The policy can be changed based on the reward value i.e., if due to an action low rewards are received, then there may be a change in policy so that more rewarding action is taken in future.

3. Value Function: The reward and value function has a difference that is, the reward indicates the value for the action taken by the agent, whereas the value function provides information about the state if it is good or bad and action related to it for the future.

4. Model: This basically helps to know about how the environment will behave for certain action [4]. RL algorithm consists of two main types of model approaches:

   (a) Model-Free: A model-free algorithm is an algorithm which does not use transition and reward functions related to Markov decision process (MDP). The transition and reward function are known as model of the environment and thus the name "model-free" [20]. This is related to a trial-and-error algorithm.

   (b) Model-Based: Transition and reward functions are used by model-based algorithm to obtain the best policy [21].

1.3.4 Types of Reinforcement Learning

Till now, we have touched in brief about the flow of RL algorithm, different implementation approaches and major elements that RL consists. In the next session, we will discuss about the main types of RL algorithms available that are used in majority of the applications related to AI. Following are the algorithms:

1. Q-Learning: Q-learning, an RL algorithm, is model-free, which iteratively improves the learning agent's actions using Q-values, also known as action

values. Q in Q-learning stands for quality [18]. The quality represents usefulness of given action for future benefits. Q-learning is an off-policy RL algorithm for determining the best course of action in a given situation. Here, "off-policy" mentions the fact that the Q-learning function learns from random actions and hence no policy is necessary. In our work, we are using Q-learning algorithm.

2. SARSA: It stands for "State", "Action", "Reward", "State", "Action". As opposite to Q-learning, this is an on-policy method [22], [23], [24]. In on-policy method the agent selects the action while learning the policy in each state. The key difference between Q-learning and SARSA is that for the next state to be updated in the Q-table, SARSA does not require the maximum reward. The reward and next action in SARSA are determined by the same policy that determined the first action.

3. Deep Q Neural Network (DQN): Here the Q-learning algorithm works using the Neural Network [4], [14], [25]. For very complex tasks it is hard to update the Q-table. To solve this, we use DQN. Instead of updating the Q-table every time, they are approximated by neural network for each action and state.

1.4 Related Works

ML, AI, and signal processing are widely used in many recent applications. The following are the main parameters that are taken into account in these applications: The amount of processing data is growing and need of robotic equipment is also increasing.

In [26], a brief presentation of hardware implementation of artificial neural network is done. The author also discusses the limitations, benefits, and drawbacks of various strategies. In [27], a detailed analysis of performance, occupancy rate, processing speed, and hardware consumption is undertaken using an FPGA hardware architecture for neural networks on image identification. However, only a few details about the hardware implementation of programmable architecture for the Q-learning RL approach can be uncovered.

A pipeline architecture for the selecting mechanism to select the most optimal action was shown in [28]. The algorithm delay grows as the number of states increases, causing the system to bottleneck. This delay can be decreased by adopting pipeline design to select the best feasible action in each state. However, no hardware implementation was displayed for other mechanisms in Q-learning RL algorithm, nor the details of tools used for hardware implementation of the choice mechanism were presented.

To use the Q-learning algorithm for real-time applications, extensive researches have been performed on improving it, but with certain limitations. These modifications/improvements were based on software [4]. Q-learning algorithm was implemented with certain modifications, which resulted in faster processing as compared to conventional algorithm [29], [30]. Hardware implementation for Q-learning was proposed. The work focused on implementing the Q-learning equation [19]. Controlling a biped robot walking in real time using Q-learning was proposed. The architectural implementation was software based [30].

1.5 Objective

Currently, research efforts have been focusing on improving the RL algorithms. This in turn will provide a smooth operation of RL applications such as controlling robotics arm, autonomous vehicle and humanoid robots, etc. We have a different perspective in terms of research.

Currently, Python/MATLAB are used to implement RL algorithms and perform their training and testing. Here, testing refers to the real-time use of the algorithm incorporated in applications such as humanoid robot, autonomous vehicle, etc. [31]. This has a processing time in milliseconds. Hence, we aim to reduce the processing time i.e., latency of the robot by implementing the Q-learning testing algorithm in Verilog HDL, and thereby providing an efficient hardware architecture that will control the robot. This will reduce the processing time from milliseconds to microseconds.

1.6 Organization of Thesis

The following is the structure of this thesis. The theoretical foundation for Q-learning is presented in Chapter 2. More specific topics like the Markov Decision

Process, the Bellman Equation, and the Exploration vs. Exploitation trade-off are also explored. Before moving on to the proposed idea, the aim of this chapter is to simplify all the above mentioned concepts and set-up all of the knowledge. In Chapter 3, firstly, an example based on Deep Deterministic Policy Gradient (DDPG) algorithm is showed that is done using MATLAB. Then, a description of the designed environment, reward and simulation mechanisms are presented. The training of the model in Python and testing in Verilog HDL along with the proposed architecture is also described in detail. Lastly, the setup parameters, data flow analysis and speed-power analysis are discussed. In Chapter 4, chip design of the proposed architecture is displayed. Also, overview of the EDA tool used for chip design is given. The design related information is also described in brief. Lastly, in Chapter 5, the work is summarized, and potential improvements are suggested and discussed.

CHAPTER 2
Background

In the previous chapter, we discussed the overview of RL algorithm, its types, and also the related works that have touched upon hardware implementations related to Q-learning. Also, we discussed upon our objective, why we are trying to implement Q-learning algorithm on hardware. The Markov Decision Process (MDP) [32] defines the sequential behavior decision problem, which is the foundation of RL. It teaches the concept of value function by describing an agent. The Bellman equation is linked to the value function. To generate the Bellman equation, RL first employs MDP and a value function, and then applies Q-learning to solve the problem. Here, in this chapter we will explain Markov Decision Process, Q-learning algorithm and, its mathematical analysis.

2.1 Markov Decision Process

MDPs are used to describe sequential decision-making processes in which actions are done while keeping future states and rewards in mind, as well as gaining immediate rewards. In other words, if the state's knowledge is adequate to predict future environmental states in response to any activity, the Markov property is said to exist in this condition (or in some literature, the state is deemed as Markovian) [33], [34].

Natural environments can also be stochastic i.e., even if the same action is performed on the same state, you may nevertheless end up in different states. When we look at the mathematical description of Markov Decision Processes, this will become evident.

The policies are termed as the rules for choosing the action that is to be done in a specific state [35]. Required terminologies are mentioned below:

1. STATE
   The state is defined as a set "S" of observable states for agents. After every action, the agent obtains a state from the environment.

2. ACTION
   A set A of feasible actions in state "S" is called an action. Actions are the moves taken by the agent in the environment.

3. AGENT
   An entity capable of exploring and action on its surroundings.

4. ENVIRONMENT
   A location where the agent can be found or is surrounding. The environment is assumed to be unpredictable i.e., it is random in nature.

5. REWARD
   After performing an action the agent will receive feedback from the environment, this feedback is termed as Reward.

6. PROBABILITY MATRIX
   It is a numerical depiction of an agent's migration from one state "S" to another "S'" provided action "A".

7. DISCOUNT FACTOR
   The concept of discount factor refers to how the value of a reward decreases over time. Discount factor diminishes the amount of rewards received by the agent over time and has a value between 0 and 1 [37].

8. POLICY
   When an agent reaches to a specific condition, it uses the policy to determine what action to take. Finally, by learning better policies than the present one, RL algorithm obtains an optimal policy [36].

2.2 Value Iteration Algorithm

The Value Iteration algorithm is based on the Bellman Equation, according to which the maximum reward obtained for an action is the ideal value for the state [34]. The Q-value can be termed as the sum of the expected value of the immediate reward over all potential state transitions plus the penalties of the resulting state. The equation for the Value iteration algorithm is shown below for reference:

$$V(s) = \max_{\alpha} Q(s,a) \quad \ldots \quad (2.1)$$

$$Q(s,a) = \sum_{s'} T(s'|s,a)[R(s,a,s') + \gamma V(s')] \quad \ldots \quad (2.2)$$

Bellman updates are executed in full sweeps of the state space in Value Iteration. That is, the value of all states is set to a random value at the start. The Bellman Equation then sweeps over the full state space to update the value function estimate. These stages are performed until the maximum change in the value function is small or for a set number of iterations. The pseudocode of Algorithm 1 is displayed below.

Algorithm 1 Value Iteration
Initialize V(s) i.e, value function randomly for all states.
Continue until the point of convergence is reached
for each state s do
$V(s) = \max_{\alpha} \sum_{s'} T(s'\|s,a)[R(s,a,s') + \gamma V(s')]$
end for

Table 2.1: Value Iteration Algorithm [34]

The Value Iteration is a planning approach that estimates the Value function using the Bellman Equation [34]. When the probability or reward function are unknown, which is common in actual systems, value iteration techniques cannot be determined. The problem stems from the fact that the planning method necessitates the use of a world model or, at the absolute least, a simulator. Another disadvantage of Algorithm 1 is that it may not perform well when state space is huge or infinite.

2.3 Value Function

The value function is the criterion that is used to determine the best policy. It is the sum of the benefits that should be expected if the policy is followed from the current state [36]. The policy is as follows:

$$v_{\pi}(s) = E_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1})|S_t = s] \quad \ldots \quad (2.3)$$

where V_π is the expected value, E_π and R_{t+1} is the next rewarded value and γ is the discount factor.

It contains the state value function, which sums the rewards that will be received when the state is granted. It permits the agent to select a more advantageous condition.

On the other hand, the action value function considers both the state and the action. Based on the Q-function the agent picks an action. The Q-function is presented as follows:

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_{t+1} = a] \quad \ldots \quad (2.4)$$

The following equation depicts the relationship between the Q-function and the value-function:

$$v_\pi(s) = \sum_{a \epsilon A} \pi(a|s) q_\pi(s,a) \quad \ldots \quad (2.5)$$

The policy and the Q-function value are summed together for all activities. Bellman equations are used to express the Q-function and value function. The Bellman equation depicts the connection between the current state's value function and the next state's value function.

2.4 Bellman Equation

The Bellman equation was created in 1953 by mathematician Richard Ernest Bellman, and is thus known as a Bellman equation. It is linked to dynamic programming and is used to calculate the values of a decision problem at a given moment by factoring in previous states' values.

The Bellman equation is a crucial component of many RL algorithms and may be found throughout the Reinforcement Learning literature. Referring to the Bellman equation, the value function can be broken down into two parts i.e., the current reward and discounted future values, according to the Bellman equation [33].

Also, the computation of the value function gets simplified by cutting it down into small parts and finding answer for each of them, instead of summing it all

together.

Modern reinforcement learning is a method of determining value functions in dynamic programming or in an environment.

2.4.1 Bellman Expectation Equation

The Bellman equation expresses the connection between the present state's value function and the value function of the future state [36], [38].

$$v_{\pi'}(s) = \sum_{a \epsilon A} \pi(a|s)(R_{t+1} + \gamma \sum_{s' \epsilon S} P^{a}_{ss'} v_{\pi}(s')) \quad \ldots \quad (2.6)$$

where $\sum_{a \epsilon A} \pi(a|s)$ is the probability policy for doing action, $\sum_{s' \epsilon S} P^{a}_{ss'}$ is the state transition probability matrix. R_{t+1} and γ are the reward and discount factor respectively.

2.4.2 Bellman Optimality Equation

The Bellman optimal equation uses the value function to obtain the optimal value. The Bellman optimal equation is expressed below:

$$v_*(s) = \max_{\alpha} E_{\pi}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s] \quad \ldots \quad (2.7)$$

where $\max_{\alpha} E_{\pi}$ is the maximum expected reward value under given policy.

Reinforcement learning uses the Bellman expectation and Bellman optimum equations to solve the MDP problem.

2.5 Q-Learning

Unlike prior algorithms that did not distinguish between behavior and learning, Q-learning an off-policy algorithm, separates the training and testing policies. As a result, even if the action chosen in the following state was mediocre, the information was not incorporated in the updating of the current state's Q-function, resulting in the dilemma that it was a bad choice [36]. Q-learning, on the other hand, overcomes the problem because it uses off-policy. The equation for the Q-value is as follows:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_{\alpha} Q(s_{t+1}, a)) \quad \ldots \quad (2.8)$$

where, α is learning rate at which the model will be trained and has a value between 0 and 1, R_t is the reward for each action taken, γ is the discount factor.

The difference between the value iteration and the Q-learning algorithm is that the Q-value in value iteration is the total of rewards and discounted value of the next state, whereas the Q-value in Q-learning is the sum of rewards and discounted max Q-value of the next state. If we continue to perform random actions on the same state, all the possible states can be reached. This will allow us to reach close to ideal Q-value. Some parameter setup ideas could be very useful in practice for ensuring guaranteed convergence. First, over a set of training steps, the greedy strategy should reduce the value linearly from 1.0 to a tiny fractional value, say 0.1, and then be fixed at that small fraction. Thus, the agent can explore more for a number of combinations of state-action, at the beginning of training and gradually by reducing the randomness as and when the agent develops an experience. Another technique is to gradually reduce the learning rate over time.

In the initial state, if an agent chooses an action based on a policy, it jumps to the next state. Until the total Q-value reaches to a specific value that can be used to perform a task using the Q-table, the process continues for multiple times [38]. Q-learning is a simpler method in comparison to other methods, that demonstrates good learning ability in single-agent situations. It has become the foundation of numerous RL algorithms [45].

2.5.1 Q-Learning Algorithm Flow Chart

- Q*(s, a) is the expected reward obtained on doing an action "a" in state "s".
- Q-Learning uses temporal difference (TD) to obtain the Q*(s, a) value. Using TD means that the agent learns in the environment without having any prior knowledge of it.
- The agent needs to maintain a Q-table i.e., Q [S, A], where S is the available states and A is the possible actions that agent can perform.

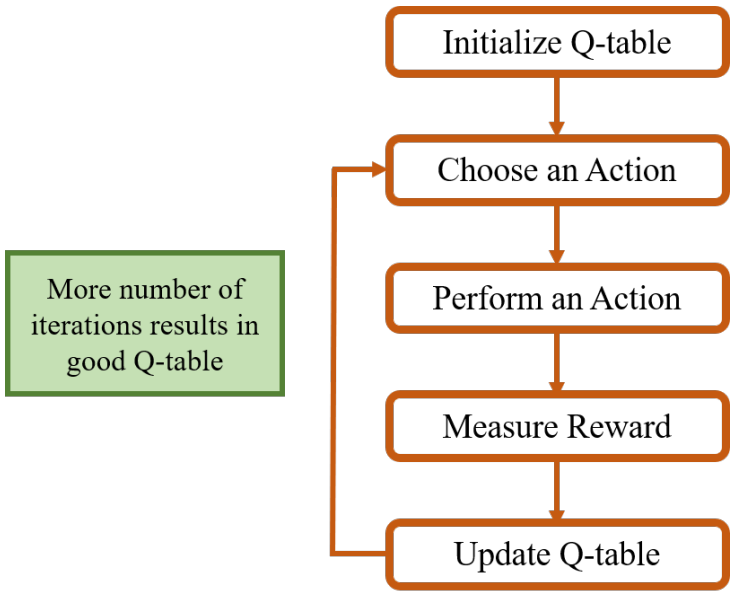

Figure 2.1: Basic Q-learning Flow

Step 1: To initialize the Q-table

The Q-table is a matrix kind of structure, which is used to store the future rewards obtained for each state. This will end up with providing an optimal action in any state. There are four possible number of actions at each state of environment. When the agent is in a state it can either move forward, backward, left or right. The Columns here represents action, and the rows represent the states. Each Q-table value represents the robot's maximum projected future reward if it does that action in that condition. We must enhance the Q-table at each iteration, so this is an iterative process.

Step 2 and 3: Choose and perform an action

This sequence of steps is repeated for an unspecified amount of time. This signifies that this phase will continue until the training is completed. Based on the Q-table, the agent will choose an action in a particular state. However, as mentioned earlier, when the episode begins for the first times, every Q-value is 0. Here the concept of exploration and exploitation comes into picture. For this trade-off we use epsilon greedy strategy. The epsilon greedy strategy will be explained in section 2.5.2.

Step 4 and 5: Measure Reward Update Q-table

After performing action, the agent receives reward and hence it should be updated in the Q-table. This is reiterative process; it will repeat until the learning is stopped. The Q-table is updated in this manner, and the value function Q is maximized. The expected future rewards of the action taken in that state is returned by Q (state, action).

2.5.2 Exploration Vs. Exploitation

All Reinforcement Learning agents gain knowledge from their previous experiences. As a result, an agent's decisions about which action to take have an impact on the various kinds of experience it has and the lessons it learns. The agent then claims that in order to reach new states with the highest future reward, it is sometimes necessary to avoid the optimal acts. If an agent always takes the action with the highest Q-value, it will almost surely find a sub-optimal or no solution. Exploit refers to adopting a policy that is exclusively based on the actions that have highest Q-value, whereas explore refers to ignoring those best actions in order to find new states and rewards. This is why it's critical to strike a balance between exploring and exploiting. This demand for stability is termed as the Exploration and Exploitation problem or Exploration and Exploitation trade-off [39], [40].

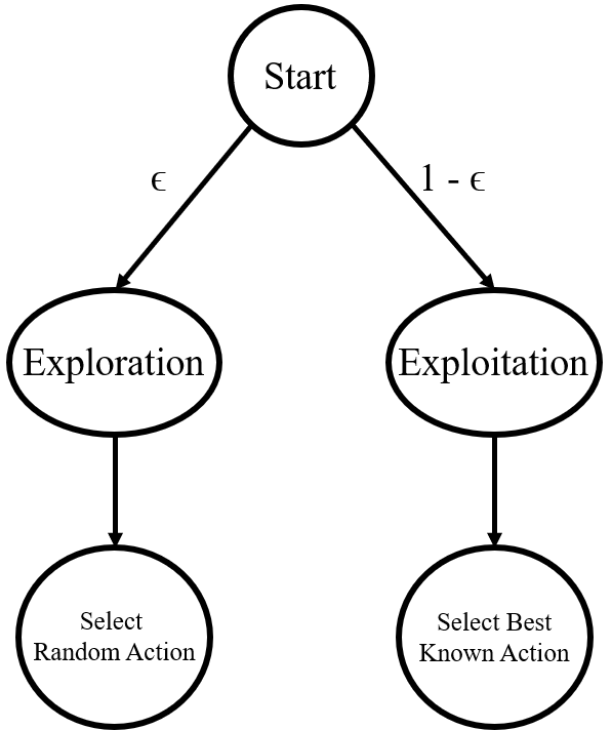

Figure 2.2: Epsilon-Greedy action selection.

We utilize an epsilon greedy technique to achieve this balance between exploitation and exploration. We initially set the exploration rate to be 1. This represents the likelihood that the agent will explore rather than exploit. It is 100% certain that the agent will begin by exploring the environment if $\epsilon = 1$.

In the beginning of an episode, ϵ will decay by a rate that we set, once the agent has great learning about the environment, through which the exploration reduces. The agent is termed as "greedy" in terms of exploitation, when it gets a chance to explore the environment [39], [40].

2.5.3 Q-Learning Algorithm Pseudo-code

Algorithm 2 Q-Learning Algorithm
Initialize Q-table randomly for each state-action pair
for each episode do
get initial state s
repeat
select a using policy derived from Q (e.g., ϵ-greedy)
take action a, observe next state s' and obtain reward r
update Q (s,a)
until s is terminal state
end for

Table 2.2: Q-Learning Algorithm

Algorithm 1 provides an explicit technique for implementing the Q-learning algorithm. Inputs consists of the discount rate, reward function and learning coefficient. The first stages are to set up the initial state s0 and the Q-table. The algorithm selects an action for the current state from a list of options and monitors the next state and reward. The Q-value then gets updated. This continues until the final version of Q-table is not obtained.

CHAPTER 3
Related Work and Proposed Method

In previous chapter, RL and Q-learning algorithm were explained in more detail. In this chapter, we describe the implementation of Deep RL algorithm for Walking Robot and Q-learning algorithm Hardware and setup parameters, implementation results and analysis based on it. First of all, Deep RL algorithm for Walking Robot implementation in MATLAB is presented. This implementation was done to explore the algorithm and working of it. Subsequently, we have introduced the implementation of Q-learning algorithm by differentiating it in two parts i.e., training and testing. Also, we have presented the task that agent needs to perform and the environment used for the same. The parameter used in implementation are also discussed. Finally, we have shown the simulation results and their analysis.

3.1 Deep Reinforcement Learning for Walking Robot

RL a type of machine learning is all about how the virtual agent thinks of taking an action to maximize the total reward in an environment. Figure 3.1 shows the RL model used in this example. The Environment is the world where the actions need to be taken in order to achieve the target. Here the environment is the lower body of the robot i.e., two limbs. Here the actor-critic agent model is chosen, DDPG algorithm. There are two neural networks in this example one is actor and the second is critic. The actor has state observations and actions to be performed as input and output respectively. The critic calculates the value of the state of the environment based on the actions performed by the actor. This helps to update the weight values of both the neural networks. This example aims to provide the ability to balance and walk to the lower body of robot. Simscape Multibody was used to model the robot in this example.

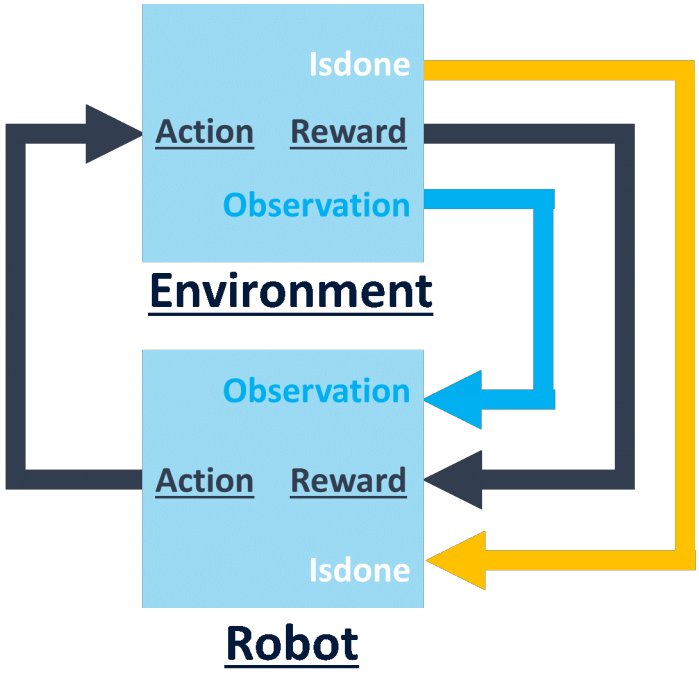

Figure 3.1: RL model implemented for biped robot simulation.

This example trains the agent in the robot environment using the model parameters. The following settings are shared by all of the agents in the example [41].

- The robot's initial condition strategy
- Actor and critic network structure
- Possibilities for actor and critic representation
- Actor and critic training options

3.1.1 Deep Deterministic Policy Gradient (DDPG) Agents

The Deep Deterministic Policy Gradient (DDPG) algorithm is an off-policy reinforcement learning method that is model-free and online. A DDPG agent is an actor-critic reinforcement learning agent that seeks out the best policy for maximizing the predicted long-term reward [41]. DDPG agents can be trained in environment with observation space to be continuous or discrete and action space to be continuous. During training, a DDPG agent does the following set of things:

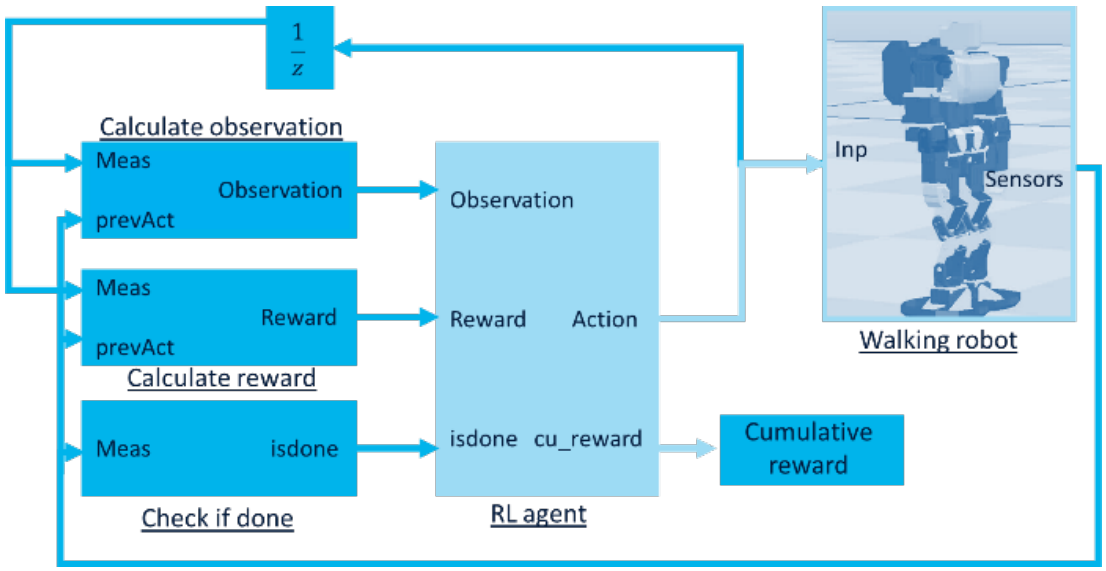

Figure 3.2: Block diagram of reinforcement learning model for the control of humanoid robot waking.

- At each learning phase, the actor and critic properties are updated.
- A circular experience buffer is used to store previous experiences. Using a mini-batch of experiences randomly taken from the buffer, the agent updates the actor and critic
- At each training step, a stochastic noise model is used to change the policy's action

3.1.2 Training Result

The DDPG agent seems to learn more quickly (around episode number 1700 on average), shown in Figure 3.3. In this example, the training was done for around 2200 episodes. It took around 24 hours to train the model for these number of episodes. Also, episode reward started to increase after 1500 episodes.

3.1.3 Simulation Results

Figure 3.4, shows the simulation of the project i.e., robot walking along the path towards the final destination. In this work, the most important thing is the interaction of the robot with the floor. The main aim of this example was to achieve a movement relatable to human walking, which here is forward motion along with balancing and improving the movement.

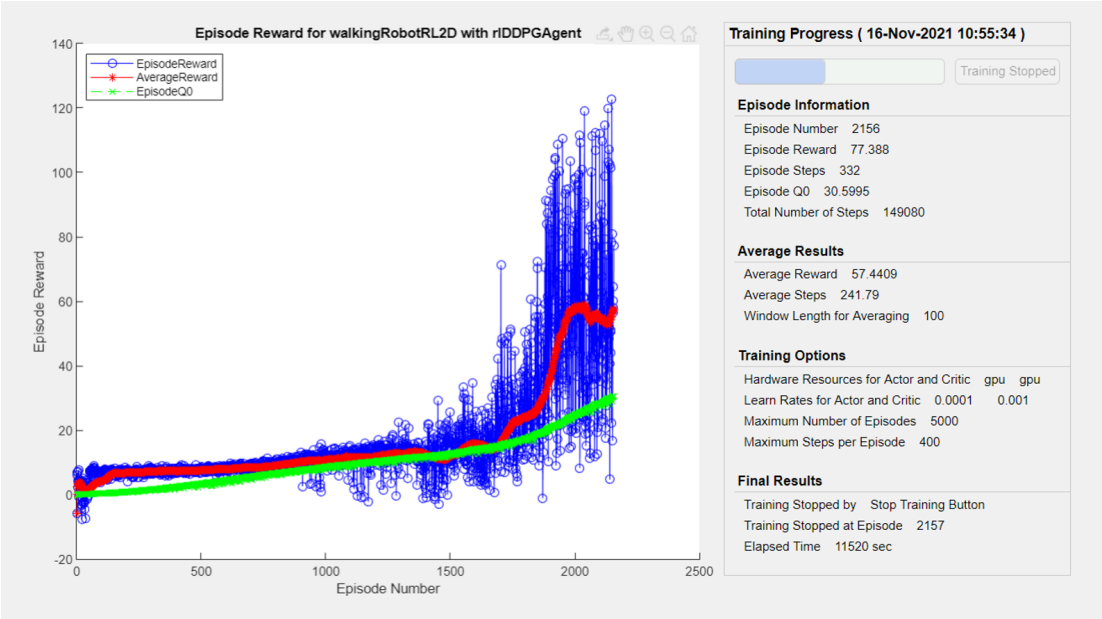

Figure 3.3: Training results obtained from MATLAB simulation.

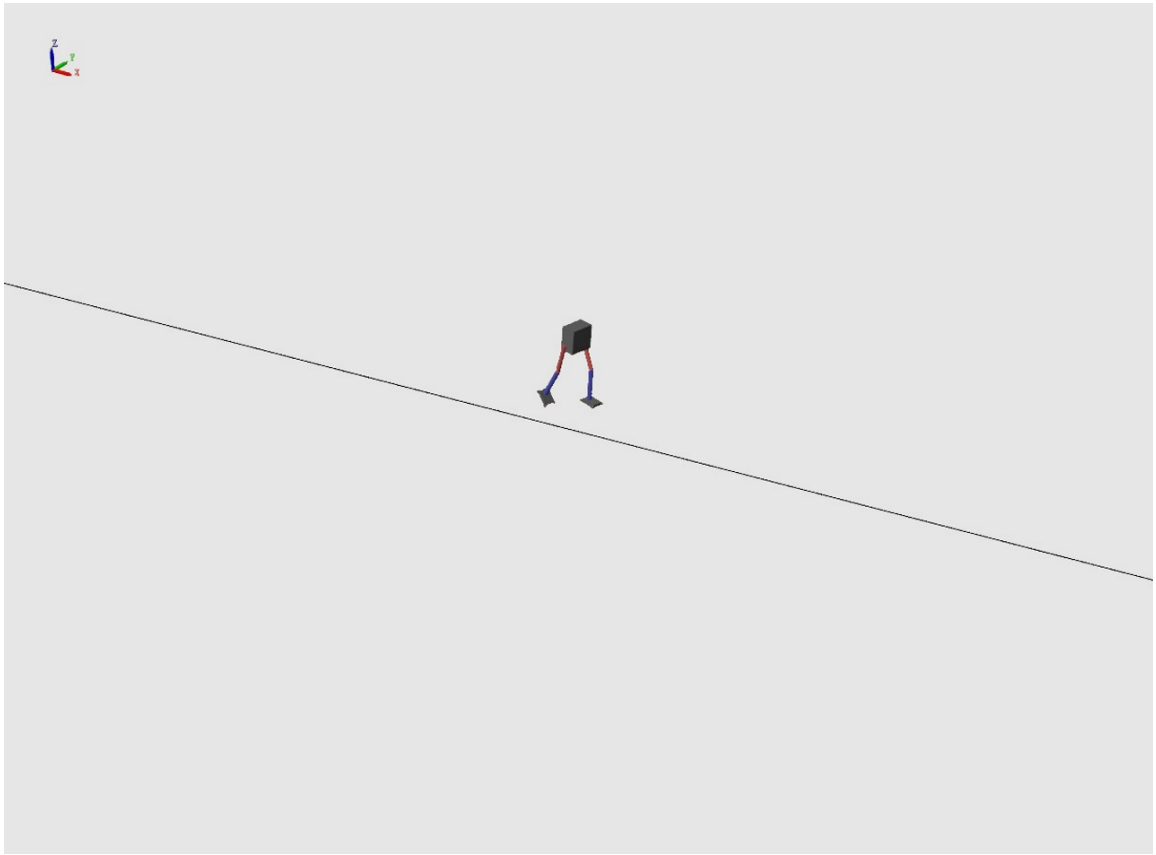

Figure 3.4: Lower body simulation of humanoid robot in the environment after training the model.

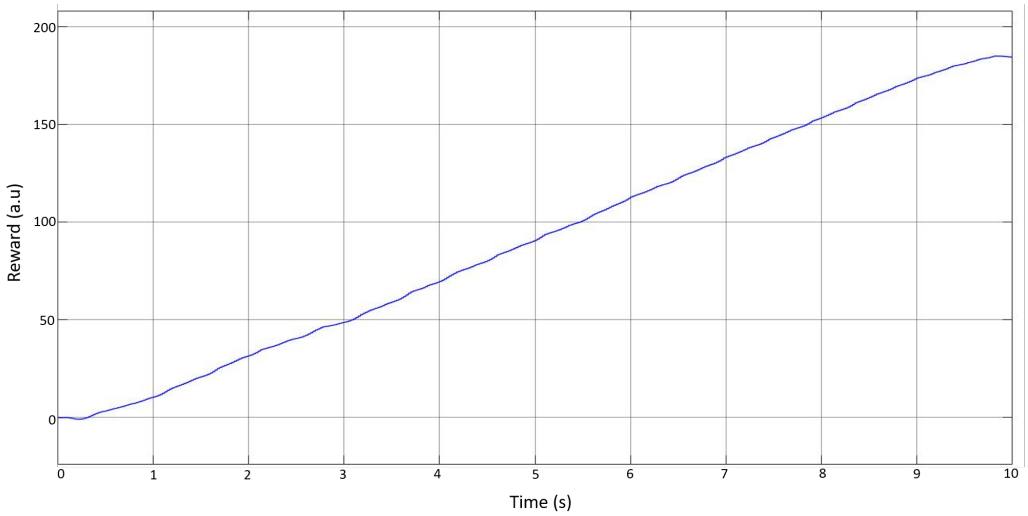

Figure 3.5: Illustrates the cumulative reward with training time.

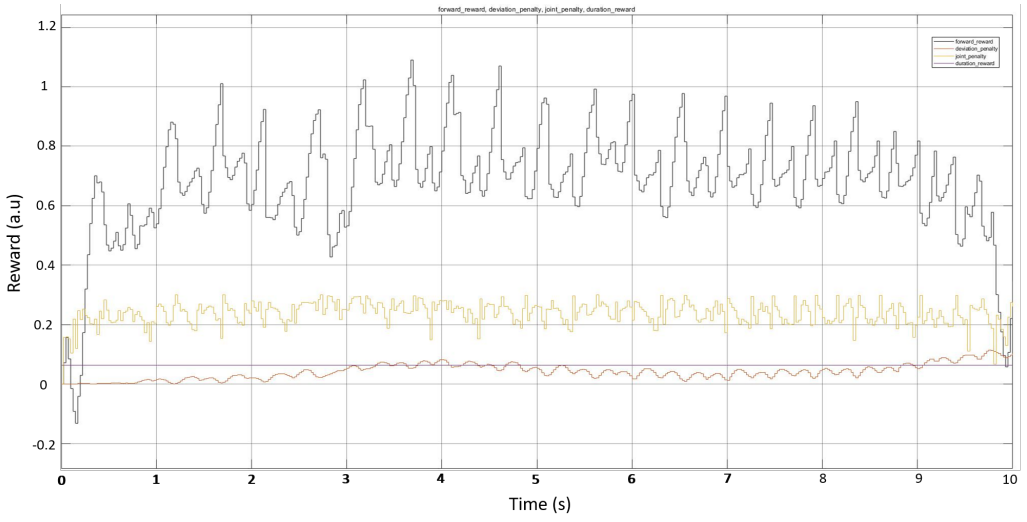

Figure 3.6: Illustrates individual reward with training time.

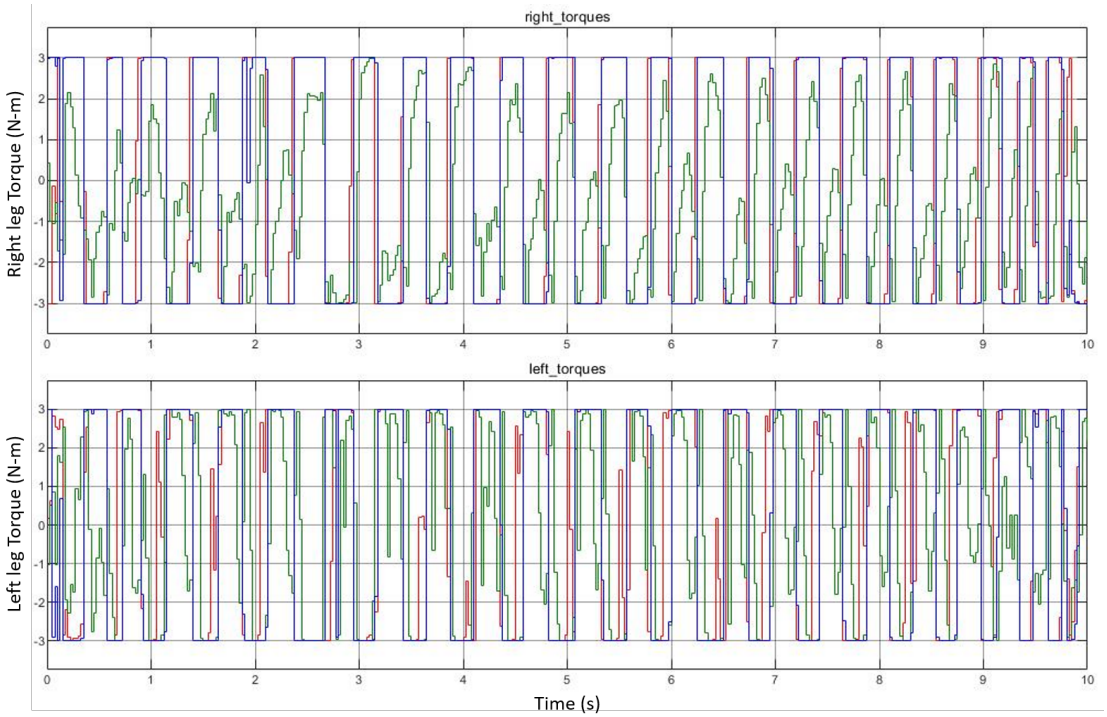

Figure 3.7: Lower body actions (torque in N-m) for left and right legs movements of humanoid robot.

The simulation was performed for 10 seconds. The agent needs to reach the end point, without deviating from the straight-line path. The results presented above consists of cumulative reward, which is the average reward received till that time. It increases gradually as the agent follows the path. Also, there will be different rewards/penalties for different robot movements, which will result in increment/decrement of the total reward obtained respectively. As this example consists of the biped robot walking, each limb will have some torque value for every step, which will make the robot walk which is shown in Figure 3.7.

The major aspect to discuss here is the very high computational cost is required for the training of the agent using deep RL, which takes few days to complete. MATLAB is majorly used for Deep RL applications, where more computational power is needed.

This example is considered to analyze the working flow and complexity of RL and to have an exposure of MATLAB as a tool used for RL.

3.2 Proposed Method

As the robot movement is a very complicated task and requires a lot of resources, we divided in two parts, one of which is training and second is testing. We are implementing an efficient hardware for the Q-learning algorithm, which in turn will provide a faster execution compared to its current implementation. The area in which we are focusing has a very negligible or have no research papers that aim to implement a hardware for this purpose. They are focusing on improving the RL algorithms that would make smooth simulations of the application. Thus, we started to explore from an initial level. Hence, to reduce the complexity and to provide better results, we are developing Q-learning instead of Deep Q-learning, which is the base of RL for hardware implementation. The training of Q-learning model is done in Python, whereas the testing is done in Verilog HDL. The Q-table is obtained by training the model in Python, the same will be used in Verilog HDL for testing it.

3.2.1 Task Description

This project's major goal is to create and develop an agent which can act upon certain task. The three sub-tasks or sub-objectives that make up this task are as follows:

- Respect the environment's boundaries.
- Avoid the obstacles.
- Reaching the goal location.

As a result, in order to assume that the task is completed, the agent must complete the three sub-objectives listed above. Furthermore, completing the task outlined is not the primary purpose of this thesis because it can be done in a variety of ways, some of which are more efficient or complete than others. The optimality and completion of the solution found by the agent are determined by the following criteria:

- Number of times the agent hits the obstacle in total number of episodes completed.
- Number of steps needed to complete each episode.

3.2.2 Environment

When experimenting with RL, one of the most important aspects to consider is the environment design. The environment's primary responsibility is to imitate the inputs that the agent would get in a real or non-simulated setting and to adapt itself in response to the agent's outputs. We have considered a 2D environment where we taught the agent to move from one tile to another and also it will optimize by learning from the mistakes. The 2D environment is a 4×4 grid which contains four possible areas – Start (S), Tile (T), Obstacle (O) and End (E) as illustrated in Figure 3.8. In Verilog HDL, as the hardware doesn't understand the characters, the binary representation of possible areas is done i.e., Start (00), Tile (01), Obstacle (10) and End (11). Here is visual look of 2D environment:

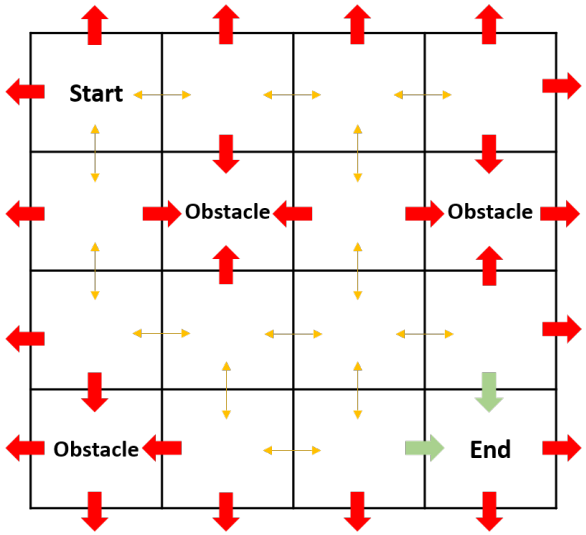

Figure 3.8: Visual representation of 2D environment used for this work.

Case	1	2	3	4
|S|	16	64	256	1024
|A|	4,8	4,8	4,8	4,8

Table 3.1: Possible Test Cases based on Environment Size and Actions

The maximum total number of states is represented by |S|, and the maximum total number of actions is represented by |A|. (a, b) coordinates are used to address the states. When there are 64 total number of states, the a and b coordinates are represented by 3-bits. Actions are represented by a series of integers. Each action is represented by a 2-bit binary value in the instance of four actions, where 00 refers to left, 01 refers to right, 10 refers to forward, and 11 refers to backward. While 000 refers to left, 001 refers to top-left, 010 refers to up, 011 refers to top-

right, and so on in a clockwise direction. Total 8 actions will be there, which will be represented by 3-bit binary number. In this thesis, we are assuming 16 states, each with 4 possible actions.

3.2.3 Simulation Mechanism

Aside from the environment's structure and features, there are various rules that controls the simulation and determine the dynamics of the interaction between the agent and an environment.

Firstly, the agent will have four possible number of actions that it can perform i.e., Left (0), Right (1), Forward (2) and Backward (3). The agent will then travel to the adjacent cell in that direction by doing any of the four potential actions, which are all moves. A step in the simulation is defined as the selection and execution of any activity. However, the two possible instances where the agent does not carry out the movement resulting from a chosen action are the agent's choice of action that leads to an obstacle or the environment's restrictions (grid).

In terms of the resulting interaction of the agent with the objects scattered throughout the environment, an episode is termed as finished if the agent reaches to the end or if it hits to an obstacle. Once the agent performs the action, the position of it in the environment is updated.

Finally, an episode completes if the set number of steps have been performed or it has reached the goal/obstacle location

3.2.4 Reward Mechanism

The reward principle is what makes Reinforcement Learning so diverse and powerful. Giving good rewards to the agent when it performs as expected and punishing it when it won't. As a result, the reward system design becomes extremely relevant and important in any RL design or implementation.

In our case, if the agent meets with an obstacle, then it will be rewarded with 0 and the agent then starts with a new episode, whereas if it reaches to the finishing point, then it will be rewarded with 1, and the agent will start with the new episode again. Reward for each step is evaluated using the Q-learning equation mentioned in chapter-2 (section 2.5).

3.3 Training the Agent

The agent is the only component that has the ability to adapt and learn through simulations, given that the environment is a static component with already set dynamics. The Q-learning algorithm guides the agent's dynamics as well as its learning process, as detailed in Chapter-2. The world refers to the two-dimensional grid, its element configuration, and the agent.

To know and grasp their situation in relation to the environment at all times, all RL agents require their own representation of the world. The agent's condition refers to the agent's own representation. The state of base agent, created initially, was directly specified by its grid co-ordinate positions. This was the only information except reward, which the agent received from the environment. This basic agent's learning process was identical to the one described in Algorithm 2 (refer chapter-2, section 2.5). As a result, while taking each step, the agent was learning from the information it encountered. Also, initially when the agent starts to learn we need to allow random movements and then gradually its probability should be reduced. Thus we can minimize the error by minimizing the loss. Finally, a table was created that contained all of the Q-values which needs to be updated at each step.

3.3.1 Q-table Rewards obtained from Training

There are 16 possible locations where the agent can be at any given instant of time. In the present state the agent will have four possible actions that can be performed for the next state. Hence, 16 possible locations, each having 4 possible actions which forms a Q-table of size 16×4, where states are the rows and actions are columns.

We will have an updated Q-table once the training is concluded, which is shown in fig. 3.9. Also, for analysis purpose we can have average rewards per thousand episodes, shown in Figure 3.10. Both the figures are obtained while training the agent. It can be concluded that the average reward value increases along with the number of episodes.

```
[[0.54280691 0.49507268 0.49046761 0.4879708 ]
 [0.34076411 0.38259701 0.34038782 0.50364463]
 [0.41623005 0.4275343  0.39877404 0.47476726]
 [0.33092483 0.27176289 0.30464656 0.44931632]
 [0.55549163 0.19025993 0.23049143 0.41713372]
 [0.         0.         0.         0.        ]
 [0.25712276 0.12218955 0.15739433 0.10350397]
 [0.         0.         0.         0.        ]
 [0.36276422 0.36806197 0.46927636 0.59200631]
 [0.41750877 0.61508794 0.46192973 0.33323423]
 [0.54586577 0.36901888 0.33970338 0.23047793]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.54680973 0.56743614 0.74351255 0.55522483]
 [0.70158606 0.86730397 0.73039726 0.72871119]
 [0.         0.         0.         0.        ]]
```

Figure 3.9: Q-table obtained while training the agent.

```
*****Average rewards per thousand episodes*****

1000 : 0.0540000000000004

2000 : 0.19800000000000015

3000 : 0.4090000000000003

4000 : 0.5530000000000004

5000 : 0.6200000000000004

6000 : 0.6600000000000005

7000 : 0.6720000000000005

8000 : 0.6760000000000005

9000 : 0.6910000000000005

10000 : 0.6640000000000005


Score over time: 0.5197
```

Figure 3.10: Average rewards per thousand episodes.

3.4 Testing in Python

Using the above obtained Q-table values, the agent was tested in the 2D environment for five episodes in Python, out of which two times the agent hits with an obstacle and three times it reaches to the goal. The results are shown in Table 3.2, along with the number of steps the agent took for each episodes. The visual transformation in the environment is also presented in Figure 3.11, 3.12.

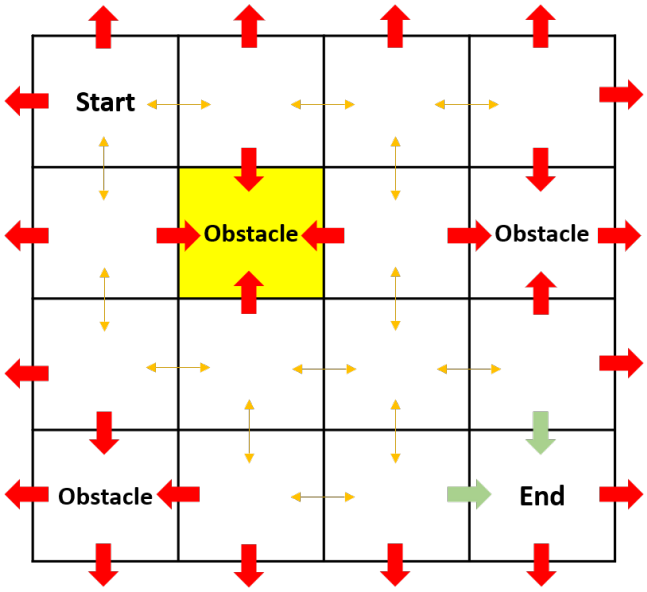

Figure 3.11: Environment visualisation when the agent hits an obstacle.

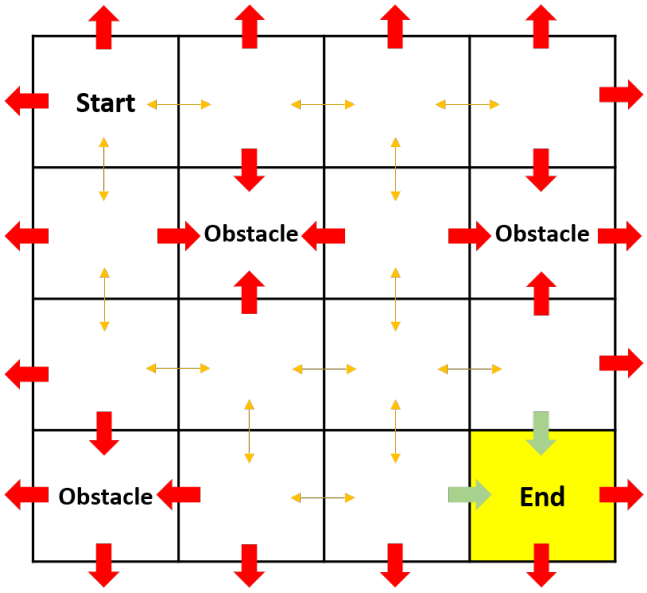

Figure 3.12: Environment visualisation when the agent reaches the goal.

Episode No.	Last tile	No. of steps	Reward
0	Obstacle (O)	19	0
1	Obstacle (O)	28	0
2	End (E)	16	1
3	End (E)	25	1
4	End (E)	32	1

Table 3.2: Simulation results

Also, we observed that the run time of the simulation was 6.08 milliseconds in Python

3.5 Testing in Verilog HDL

As mentioned in Chapter-1, we focused on the reduction of processing time, required by the agent to take the decision during the testing part. Hence, we proposed an architecture for Q-Learning algorithm for testing purpose, that will reduce the processing time.

3.5.1 Proposed Architecture

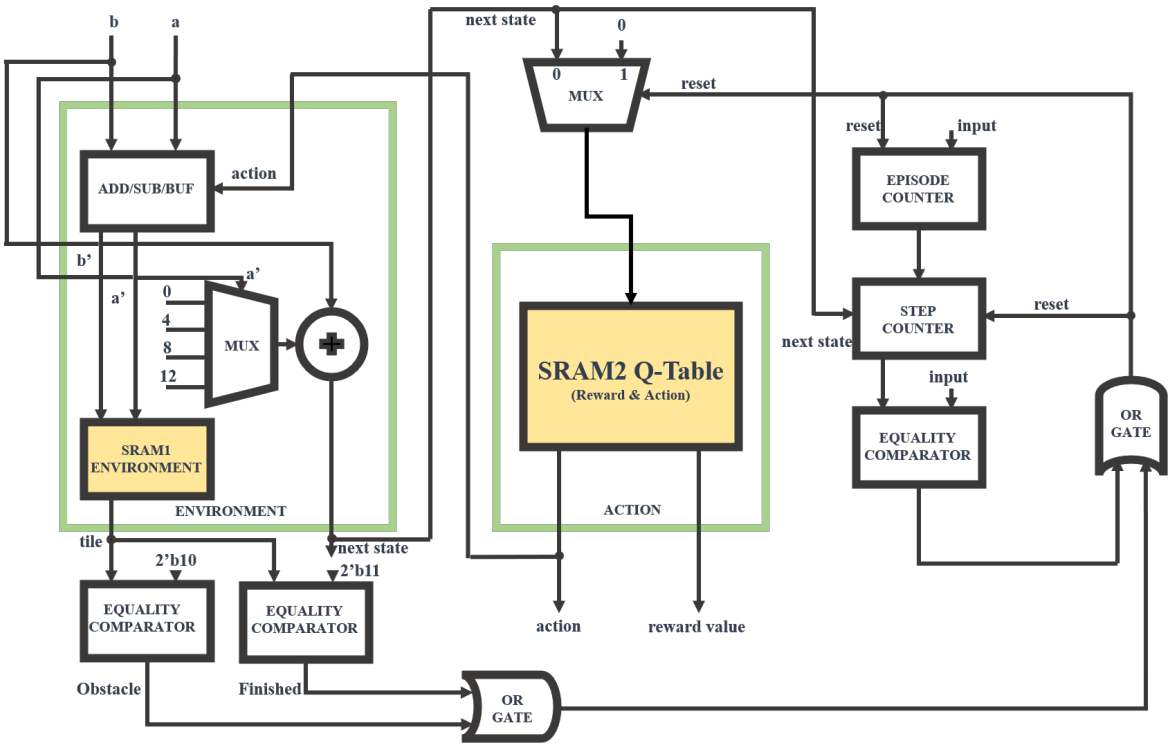

Figure 3.13: Proposed VLSI architecture.

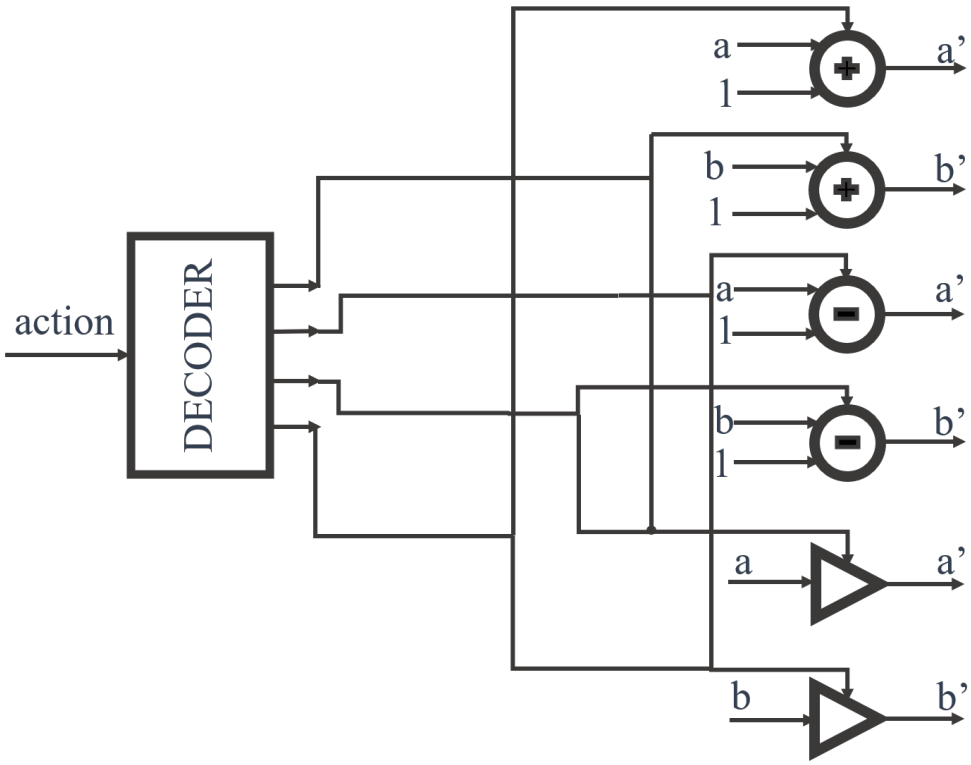

Figure 3.14: ADD/SUB/BUF blocks which are considered in the proposed architecture.

To accelerate this algorithm, we used the following resources: (i) SRAM1 for storing the 2D environment i.e., 4x4 grid. (ii) SRAM2 for storing Q-table. Here we have stored the Q-table as a labelled data set, i.e., action:reward, the first two bits from MSB are the action bits, while the rest of bits depict the reward. So, when the read operation from SRAM2 happens, action bits will be given further so that the agent's position can be updated in the environment. Thus, completion of a single step i.e., to choose an action based on the Q-table stored in SRAM2, the process for the next step will also start from SRAM2 only, thus making it the heart of architecture. (iii) Episode Counter for counting number of episodes. (iv) Step Counter for counting number of steps per episode. (v) Comparators are used to compare the results of each episode i.e., if the agent hits an obstacle or it reaches to the goal. Also, one comparator is used to compare the number of steps, if it reached the set limit. (vi) Multiplexer in the environment block is used to update the state of the agent according to the action performed and co-ordinates it reached in the environment. (vii) Multiplexer at the top is used to reset the next state if the reset signal is triggered. (viii) Adder in environment block is used to add the offset obtained from multiplexer with the previous co-ordinate value b. (ix) OR gates are used for generating the reset signal based on the episode results or if the

step count limit is reached. (x) ADD/SUB/BUF module shown in Figure 3.14, is basically a combination of adder, subtractor and buffer (refer Chapter-4, section 4.4.3 for the role of buffer). The module is used to calculate the new co-ordinates changed according to the action taken by the agent. The proposed architecture is displayed in Figure 3.13.

3.5.2 Simulation Results

The simulations are performed using the proposed architecture and are based on the Q-table obtained from training the model in Python. The number of episodes for which we have done simulation is five. The results are presented in Table 3.3. The agents hits the obstacle in all the episodes (for visual representation of the environment refer Figure 3.11). For this we need to implement dynamic policy block, through which the agent will be able to perform different actions based on Q-table values in different episodes, that will end up with taking a different paths in each episodes. This will be included in the future works.

Episode No.	Last tile	No. of steps	Reward
0	Obstacle (O)	5	0
1	Obstacle (O)	5	0
2	Obstacle (O))	5	0
3	Obstacle (O)	5	0
4	Obstacle (O)	5	0

Table 3.3: Simulation Results

The run time of the simulation in Verilog HDL is 30 nanoseconds, which is a huge reduction in comparison to that of Python which was 6.08 milliseconds. Hence, we succeeded to achieve our aim to reduce the processing time by implementing Q-learning algorithm on Hardware. The next section will discuss about how the data flow occurs in training as well as in testing process.

3.6 Data Flow Analysis

As we have mentioned above that the overall Q-learning algorithm is broken down into two parts i.e., training and testing. So, it is important to analyze each of the following part's data flow to know how it works. Training being the first part of any ML algorithm, we will discuss it first.

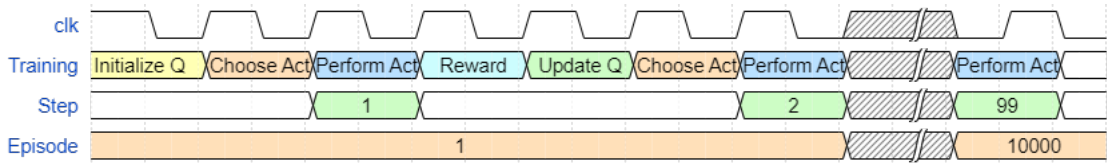

Figure 3.15: Simulated training data flow diagram.

As shown in Figure 3.15, the first step in training is to initialize Q-table. Here as the grid is of size 4×4 and total number of actions possible are 4, the size of Q-table becomes 16×4. Now the agent starts in a state, according to which it chooses the action. It performs the action and based on it, receives the reward. For the next action, the agent will have two choices i.e., referring the Q-table and to select the action with highest value or taking a random action. After performing the above steps, the agent updates the Q-table. This repeats until the number of episodes of training is reached to an end.

After training, the Q-table is ready to use. Testing is done based on the values stored in Q-table. We analysed the data flow of testing. There are two cases (a) agent hits an obstacle (b) agent reaches to the end. First, we will discuss case(a).

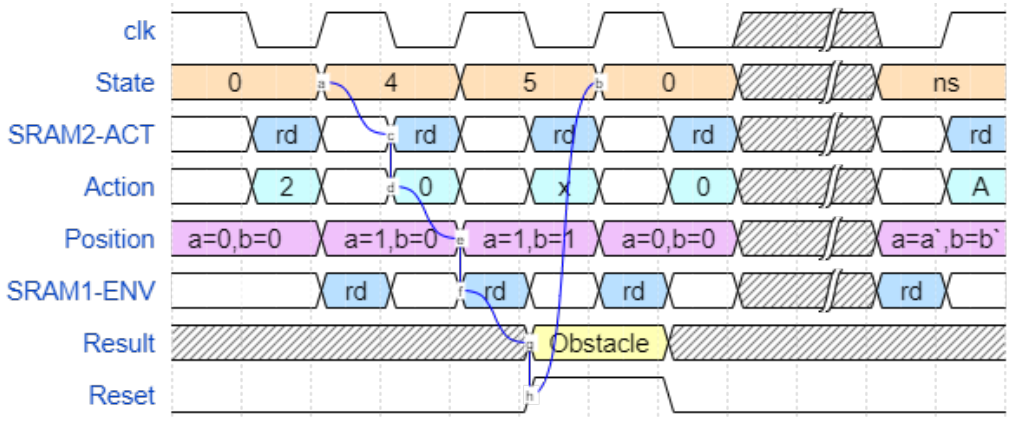

Figure 3.16: Data flow diagram when the agent hits an obstacle (case(a)).

Here in Figure 3.16 the data flow diagram for case(a) is shown, SRAM1-ENV is the SRAM that is used to store the environment values and SRAM2-ACT is the SRAM used to store Q-table and Reward values that helps us to choose the action. a=1, b=1 is the position at which obstacle is there (refer Figure 3.8). It reads the value from SRAM1-ENV for the value of a, b and detects it as an obstacle. The reset signal becomes High and it resets the state. Next episode begins and this loop continues till the episode count is reached.

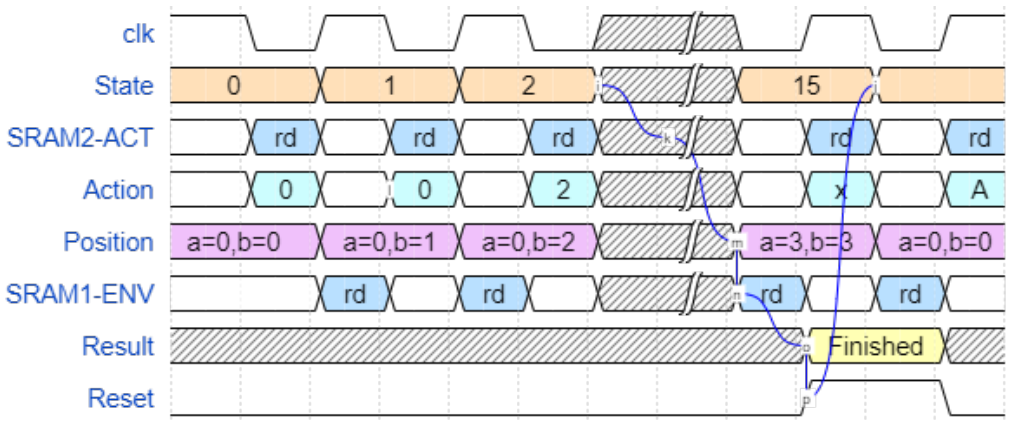

Figure 3.17: Data flow diagram when the agent reaches to the end (case(b)).

The data-flow diagram for case(b) is shown in Figure 3.17. The agent starts from a=0, b=0 and continues the movement in the environment. At certain point of time, it reaches a=3, b=3 that is the end point i.e., finish. Thus, the reset signal becomes High and resets all the counters i.e., step and episode. Also, it resets the state to 0 and hence starts a fresh episode. Here, the agent will receive reward as 1, as it completes the task. In earlier case i.e., case(a) the agent will receive reward as 0 as it hits an obstacle.

3.7 Setup Parameters

Here, we detail the design and setup of the tests that were conducted in order to assess the proposed strategy in the previous sections. To begin, the hyper-parameters used in the experiment are determined, as well as their values. Then, to compare and analyze the technique's performance, we show several technique and environment parameters, such as the utilization of the agent's vision and the size of the environment grid, and obstacles in the path.

In all RL problems, parameter tuning is a necessary step. The parameters described here, optimize the systems using a combination of theoretical knowledge and trial-and-error. To do so, a 2D, 4 × 4 grid with the agent's vision and barriers in between is used to evaluate the numerous permutations of these parameters in the same environment setup and complexity.

The values chosen for each parameter are listed below, along with their justifications [33]:

- **Learning Rate (α) (0.01, 0.05, 0.001, 0.005):** The learning process will be too sluggish if the rate is set too low; but, if the rate is set too high, the learning process will be divergent, leading to no solution. The values tried are relatively modest because we favour finding a solution to speeding up the learning process.
- **Discount Factor (γ) (0.99, 0.975, 0.95, 0.9):** Our purpose is to influence the agent to act in such a way that it emphasizes long-term rewards over short-term rewards. Then, to see if they work, significant discount values are used.
- **Initial epsilon (ϵ) (1, 0.9, 0.8):** The epsilon value indicates how unpredictable the agent's actions are. We then test large values for this parameter since we want the agent to explore the environment rather than following learned rules.
- **Epsilon Decay Rate (0.999, 0.995, 0.99, 0.975, 0.95):** This option, like the epsilon value, determines how much the epsilon decreases after each episode. To give the agent sufficient exploration and episodes at learn, the parameter is set to a consistent reduction.
- **Epsilon Minimum Value (0.3, 0.2, 0.1, 0):** This parameter is mostly concerned with the experimentation's testing phase. As a result, low values are essential because we don't want the agent to keep exploring but rather stick to the taught policy.

Performance metrics such as precision, convergence time, and overall reward are used to calculate the final parameter values. Table 4.1 shows the values that perform the best.

Learning Rate	Discount Factor	Epsilon Decay	Epsilon Rate	Epsilon Minimum Value
0.1	0.99	1	0.995	0.01

Table 3.4: Setup Parameter

3.8 Speed Analysis

The comparison between the processing time required for simulation in Python and Verilog HDL is done. The reduction in processing time can be clearly seen from the Table 3.5. These results are obtained based on frequency of 62.5 MHz.

	Processing Time
Python	6.08ms
Verilog HDL	30ns

Table 3.5: Processing Time

3.9 Power Dissipation

Different types of power dissipations resulted in the hardware implementation are presented in Table 3.6.

Dissipation	Power (mW)
Core Dynamic Power	0.73
Core Static Power	39.76
I/O Power	7.88
Total Power	48.37

Table 3.6: Power Dissipation

CHAPTER 4
Chip Design

4.1 Introduction

Electronic design automation (EDA), often known as electronic computer-aided design (ECAD), is a software program that allows you to create electronic circuits. A design flow is usually included in the software application to design and analyze the electronic circuit. There are many commercial EDA tools, which follow industry standards and are very costly in terms of license such as Cadence, Mentor Graphics, Synopsys, etc. But using Open-Source or free software EDA tool is the one and only effective way for research students and teachers to implement their ideology and also to learn. In the EDA community, Open-Source EDA has become a prominent project since it promises a number of benefits, particularly in terms of shared infrastructures like internal representations and databases, as well as tool compatibility.

Steven M. Rubin created the Electric VLSI Design System in the early 1980s as an EDA tool [42]. Electricity is utilized to create logic wire schematics and do integrate circuit architecture analysis. It also supports VHDL and Verilog as well as other hardware description languages. Design rule checking, Logical Effort, Simulation, Layout vs. Schematic, Routing, etc tools are included in the system. Electric is a Java application that was released in 1998 as part of the GNU project under the GNU General Public License.

Electric VLSI provides various types of design related environments. They exist as a technology in Electric VLSI. Each of these environments consists of a set of primitive nodes, arcs and some information about design rules. The environments present are more focused on layout of ICs and also for non-ICs.

4.2 Existing Open-Source EDA Tools

A decent EDA tool should offer capabilities like logical design, circuit schematics, layout creation, and design rule check based on the VLSI design flow. Most VLSI CAD systems that support all of these elements are not open-source software in today's market, although there are some open-source solutions that offer parts of these features individually. Table 4.1 has a list of these [43]. Other open-source EDA tools have the basic disadvantage of not being as complete as Electric. Even though certain tools provide certain aspects, the entire feature set is not integrated into a single Open-Source CAD system. Only the Electric CAD system is comprehensive and covers all needed features [42]. Due to these reasons, we used Electric VLSI for chip design purpose.

Tools	Compatible OS	Functions
Electric (9.07)	Windows, Linux, Mac OS	HDL to Layout
gEDA (3.0)	Mac OS, Linux	PCB Designing, Schematic
Magic (8.3)	Linux	Circuit Layout
eSim (2.2)	Linux, Windows	PCB Designing, Schematic
Xcircuit (3.10)	Unix, Windows	Schematic Capture
Qucs (0.0.19)	Windows, Linux, Mac OS	PCB Designing, Schematic

Table 4.1: Currently Available Open-Source EDA Tools

4.3 Electric VLSI Design Flow

The design flow followed by both Digital and Analog designs in Electric VLSI is shown below.

- **Step 1: Specifications for Digital/Analog Circuits**
- **Step 2: Schematic Design**
  - Worked on in the Schematic Design Environment.
  - Naming convention "$Cell\_name\{Sch\}$"
- **Step 3: Schematic Verification**
  - Check for schematic errors with DRC.
- **Step 4: Schematic Simulation**

  - Using the Simulation Tool, create a SPICE deck.

- **Step 5: Layout Design**
  - Standard CMOS processes are usually done in a MOCMOS design environment.
  - Naming convention "$Cell\_name\{lay\}$"

- **Step 6: Physical Verification**
  - Run DRC to ensure that the layout design follows the rules.
  - Check Layout versus Schematic using LVS.
  - Check wells and antenna rules with ERC.

- **Step 7: Layout Simulation**
  - Using the Simulation Tool, create a SPICE deck.

- **Step 8: Final Layout**

4.4 Digital Design using Electric VLSI

Electric VLSI is used to examine digital circuits in this section. Multiplexer, Adder, Comparator, Counter, Decoder, and other combinational circuits, as well as memory design, such as SRAM, are discussed. MOSIS design standards for TSMC 180nm technology were used to generate all of the physical level designs. The MOSIS website was used to obtain the transistor model files.

4.4.1 Adder 2-bit

In VLSI systems, adders are extremely important. Adders are used in microprocessors, digital signal processing architectures, parity checks, and other applications. Here, two full adders are used to construct 2-bit adder. The adder has three inputs A, B and C_{in}, where A and B are the input values which needs to be added and C_{in} is the carry obtained from the previous operation. Besides these inputs it has two outputs C_{out} and sum. Each of the inputs and outputs, except C_{in} and C_{out} are of 2-bits. When A = 01, B = 10 and C_{in} = 0, then the sum = 11 and C_{out} = 0. Similarly, when A = 01, B = 10 and C_{in} = 1, then the sum comes out to be 00 and C_{out} = 1. The 2-bit adder was used in ADD/SUB/BUF block (refer chapter-3, section 3.5), to add the coordinate values. The layout area of the 2-bit adder is 137μm x 236.5μm.

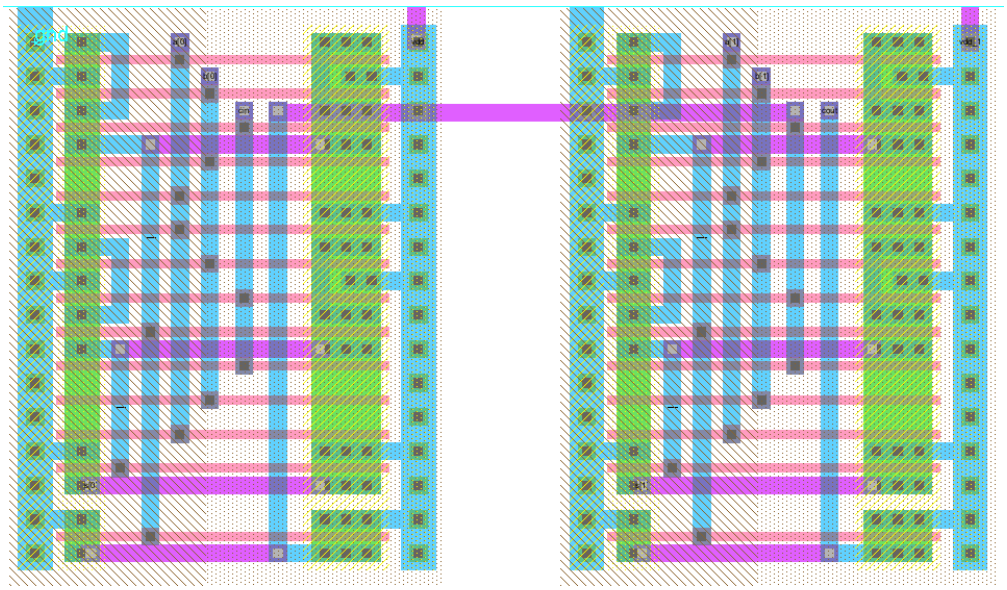

Figure 4.1: Adder 2-bit layout.

4.4.2 Adder 4-bit

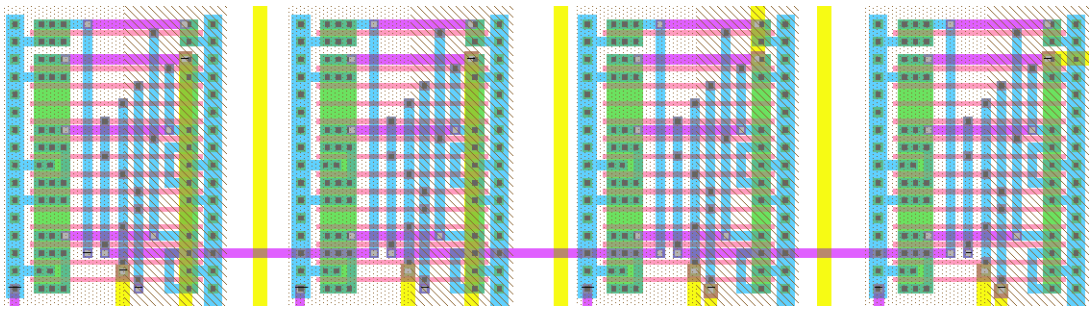

Figure 4.2: Adder 4-bit layout.

Here, four full adders are used to construct 4-bit adder. The operation here, remains the same as it was in 2-bit adder. A, B and sum are of 4 bits, while C_{in} and C_{out} are of 3 bits. The layout area of the 4-bit adder is 137μm x 499μm.

4.4.3 Buffer

A buffer has only one input and one output and behaves in the opposite way as a NOT gate. It merely sends its input to its output unmodified. Four back-to-back inverters are used to construct buffer. A buffer is mostly utilized in a Boolean logic simulator to increase propagation latency. A buffer can be used to amplify a signal in a real-world circuit if its current is too weak. We have used buffer to provide a delay in the ADD/SUB/BUF block (refer chapter-3, section 3.5). There are two

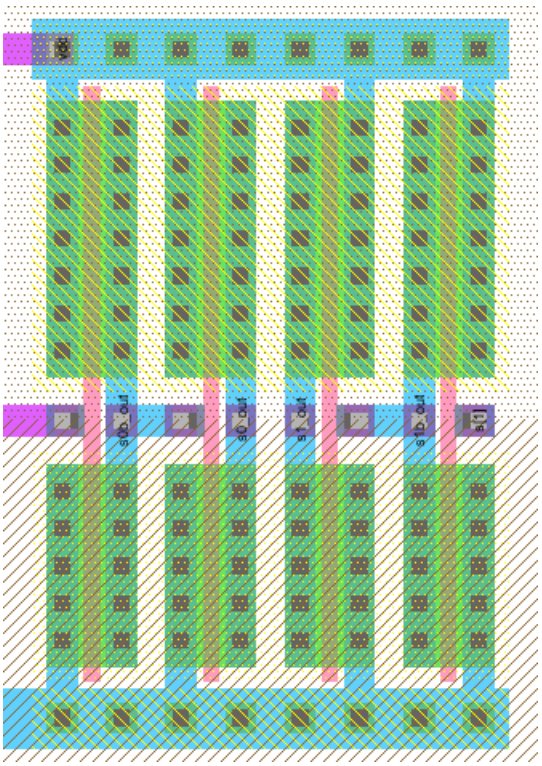

Figure 4.3: Buffer layout.

coordinates in the environment (discussed in chapter-3, section 3.2), which gets updated when an action is taken by the agent. Based on action, at most one of the coordinates will get updated and the other will remain unchanged. Thus, the unchanged coordinate will reach the output prior to the changed coordinate. As the changed coordinate will go through adder/subtractor, it will reach after some delay. Thus, to make both the coordinates reach to the output at the same time we have used buffer, which will provide same amount of delay as the add/subtractor will provide. The layout area of the buffer is 73μm x 107μm.

4.4.4 Decoder 2-to-4

A combinational circuit that decrypts the inputs, is known as Decoder. Thus, the input has lesser bits as compared to the output. We have used a 2-to-4 decoder. Three inputs are processed into eight outputs in a 2-to-4 decoder. It has two inputs and four outputs. Only one of the four outputs is chosen based on the combinations of the two inputs. When both the inputs are 0, then the output is 0001. When both the inputs are 1, then the output is 1000. The layout area of 2-to-4 decoder is 169μm x 113μm.

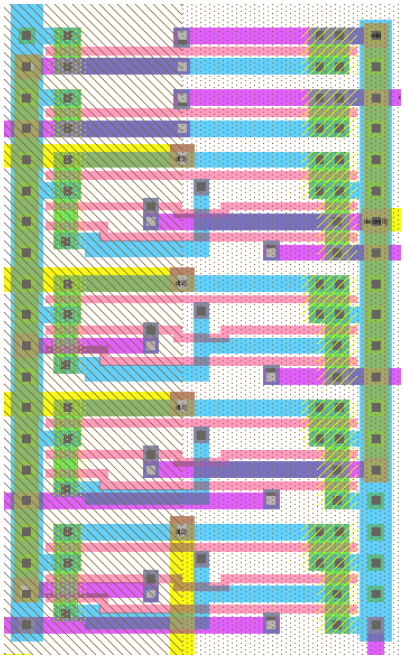

Figure 4.4: Decoder 2-to-4 layout.

4.4.5 Equality Comparator

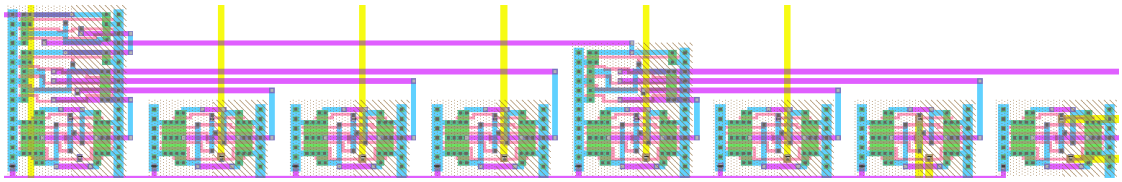

Figure 4.5: Equality comparator layout.

A digital comparator, also known as a magnitude comparator, is a digital circuit that compares two binary integers to see if one is greater than, less than, or equal to the other. Central processing units (CPUs) and microcontrollers both use comparators (MCUs). Here, we have used an 8-bit equality comparator, which will only check the equality between the two binary numbers. The layout area of 8-bit equality comparator is 153μm x 2391μm.

4.4.6 Counter

Counter is a sequential circuit. It is used to count the pulses. Flip-flops are the basic elements that are used to construct the counters. We have used two counters one for episode counting and the other for step counting. Both the counters are of 8-bits. The layout area of the 8-bit counter is 137μm x 1027μm.

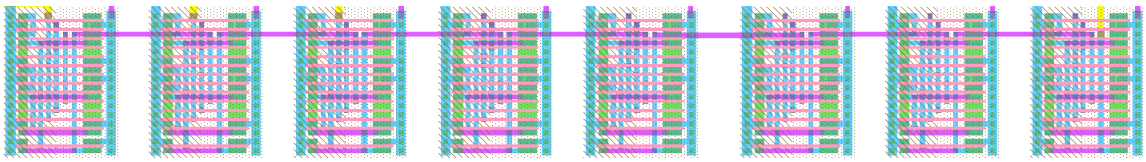

Figure 4.6: Counter layout.

4.4.7 4-to-1 Multiplexer 4-bit

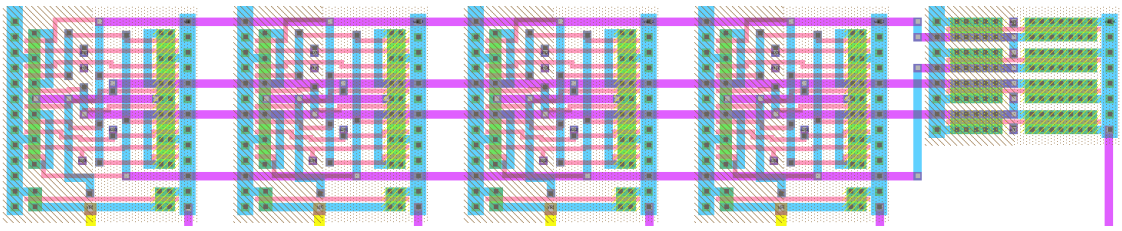

Figure 4.7: 4-to-1 Multiplexer 4-bit layout.

A multiplexer is a digital switch with numerous inputs but only one output. With the use of control inputs, it selects one of multiple inputs and transmits a single output through it. Here, we have used 4-bit 4-to-1 mux, which has 4 inputs each of 4 bits. The layout area of 4-bit 4-to-1 mux is 113μm x 587μm.

4.4.8 SRAM

Any computing platform would be incomplete without memory. During the execution of any application or program, it is responsible for storing temporary and permanent data and instructions. SRAM (static random-access memory) serves as a processor cache and is volatile memory. Data that is stored electronically and does not need to be refreshed on a regular basis is referred to as "static." It is a volatile memory, meaning it does not store data permanently, but it does provide high-speed buffering for processors. The phrase random access was invented because any item of data can be seen at any time. A 1-bit SRAM circuit is explained and developed at the schematic and physical levels in this section.

We have used SRAM as we need a small amount of memory. Conventional SRAMs have less reliability while holding the data in presence of noise, since 12T SRAM provides this as an advantage over the conventional SRAMs. We thought of using 12T SRAM, as our application is used for prediction and needs data that is error free i.e., less affected by noise, knowing that this would increase area requirement and a bit of power consumption [30], [44], [46].

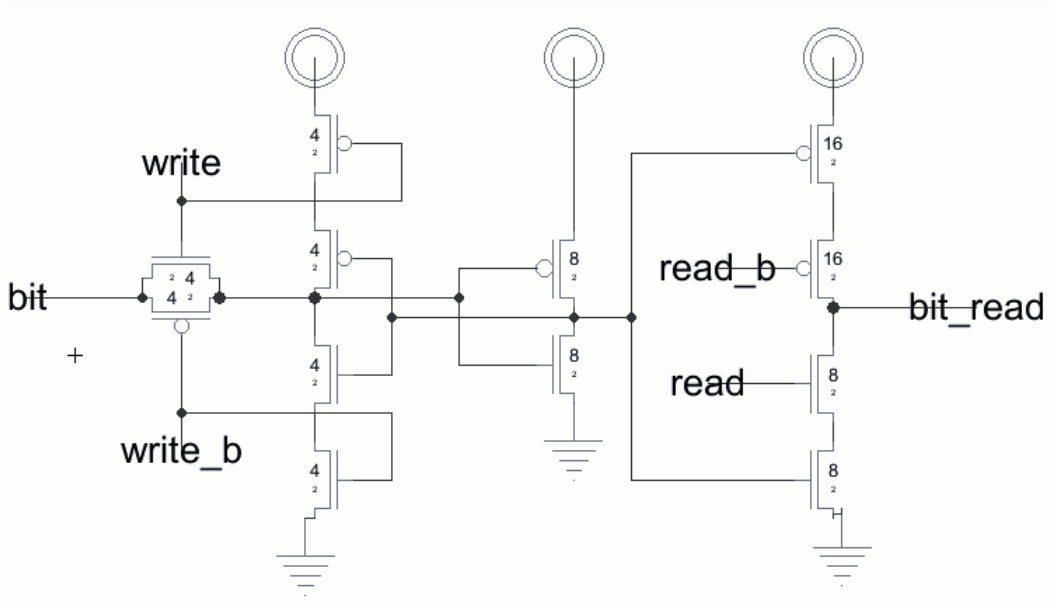

Figure 4.8: Schematic diagram of 1-bit SRAM cell.

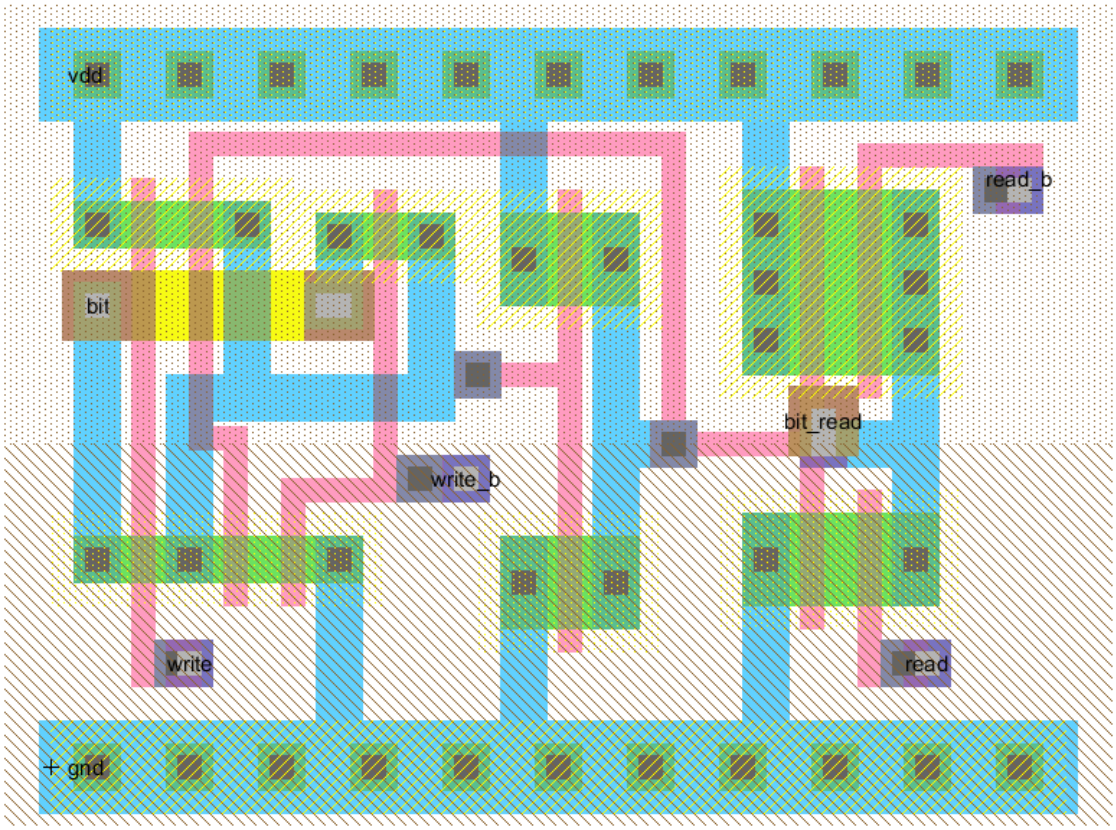

Figure 4.9: 1-bit SRAM cell layout.

The schematic diagram of 12T SRAM cell is shown in Figure 4.8. The layout of 12T SRAM cell is shown in Figure 4.9. We have used 2 SRAMs of size 64×4 bits each designed using 12T SRAM cell. SRAM1 is used for storing the environment variables and SRAM2 is used to store Q-table and reward values that will help to choose an action. The layout of SRAM is shown in Figure 4.10. The area of SRAM is 3934μm x 826.μm.

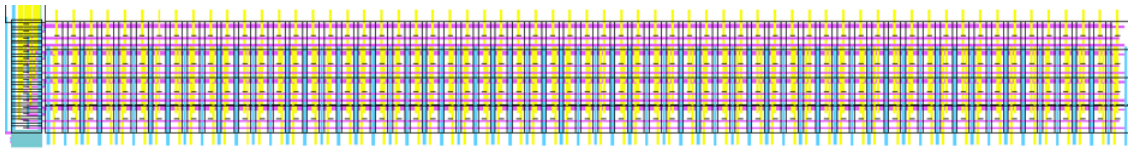

Figure 4.10: SRAM layout.

4.5 Final Chip Design

Final design of chip for the proposed algorithm is presented in Figure 4.11. The chip design is done with 180nm technology file and is a 40-pin chip. The pin description of the chip design is given in the following table.

PIN NUMBER	DESCRIPTION
1	Power Supply
2-3	SRAM_ENV inputs
4-10	Comparator2 inputs
11-12	Comparator1 inputs
13-14	Comparator0 inputs
15-16	A input
17-18	B input
19	Output
20-23	Next_State
24-28	SRAM_ACT
29	Reset
30-32	Episode_counter inputs
33-36	Step_counter inputs
37	Read_write SRAM2
38	Read_write SRAM1
39	Clock
40	Ground

Table 4.2: Pin Description

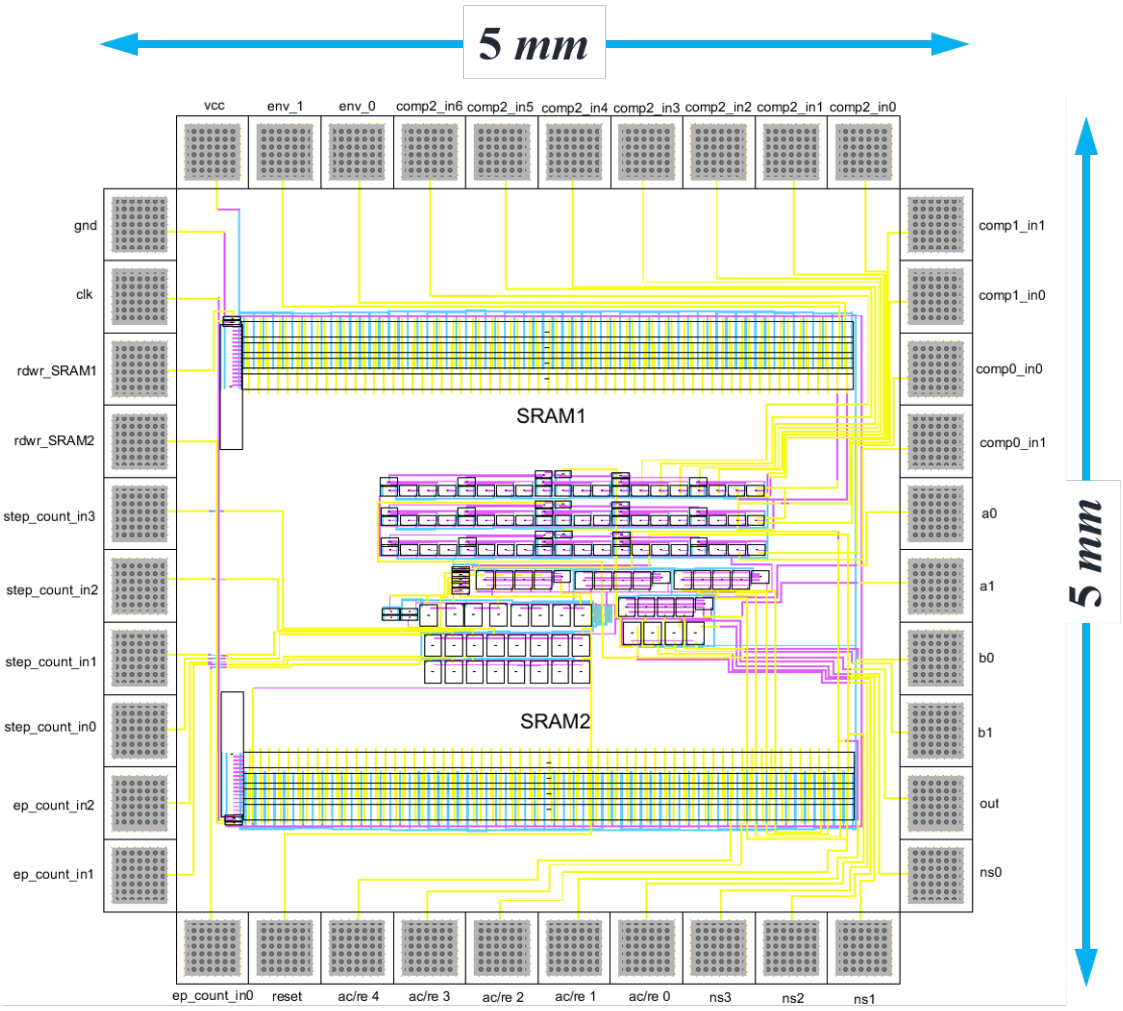

Figure 4.11: Chip layout for the proposed algorithm.

4.5.1 Area

The area of the chip design for proposed architecture is 25 mm^2.

Sr. No.	Layout	Area	Unit
1	Inverter	2675	μm^2
2	Buffer	7811	μm^2
3	2 Input OR Gate	4387	μm^2
4	2 Input AND Gate	4280	μm^2
5	3 Input AND Gate	5243	μm^2
6	2 Input NAND Gate	3531	μm^2
7	3 Input NAND Gate	3424	μm^2
8	2 Input NOR Gate	6955	μm^2
9	Half Adder	10767	μm^2
10	Full Adder	14659	μm^2
11	Full Subtractor	14873	μm^2
12	2 to 4 Decoder	19097	μm^2
13	4bit 4x1 Mux	66331	μm^2
14	2bit Full Subtractor	36643	μm^2
15	2bit Adder	32400	μm^2
16	4bit Adder	68363	μm^2
17	8bit Counter	0.1407	mm^2
18	8bit Comparator	0.3659	mm^2
19	1bit SRAM cell	7469	μm^2
20	256bit SRAM	3.248	mm^2

Table 4.3: Area Chart

4.6 Summary

As our work is started from scratch and as our initial aim is to get the desired outputs, we have used 180nm technology. We started with building the standard cells such as different logical gates, multiplexer, half adder, full adder, etc. After building and simulating the standard cells we started designing other required modules i.e., different bit (8, 16, 32 etc. as per requirement) adders, comparators, counters, decoder, multiplexers, SRAMs, etc.

CHAPTER 5
Conclusion

This paper demonstrated a chip-based hardware architecture for the Q-learning technique. The focus was given on implementation of reinforcement learning techniques on hardware at the cutting edge. The key objectives for the development of this work were illustrated by a list of applications that use Q-learning as a technique.

The design and implementation of an agent having the ability of avoiding the obstacles and reaching the target in an environment with minimum number of steps were given in this thesis. We also spoke about how the agent needs vision and how an experience replay memory might force it to learn from past random experiences.

To assess the planned system's performance, we provided many scenarios and arrangements that allowed us to examine the impact of various aspects and features, such as the agent's eyesight and the difficulty of the environment. The findings mentioned in the previous chapters demonstrated the significance of these characteristics in terms of the agent's ability to comprehend its surroundings and behave appropriately. Even though the trials just scratched the surface of our system's capabilities, they demonstrated the vast potential of RL approaches and its derivative algorithms. We will also discuss the future work and design problems faced in our research work, in this chapter.

Q-learning, an RL technique, finds a best policy for interacting with the environment without having any prior knowledge of the system model [47]. By implementing this algorithm on hardware, the processing time of the system can be reduced. The Hardware architecture's implementation has been described in detail. Details of various system modules were also explained, as well as the hardware techniques utilized to implement the Q-learning algorithm.

Finally, one of the thesis's long-term goals is to create a system that can do real-world activities. In that respect, the system demonstrated that it is conceivable if more effort is put into this direction. This is why we finish this thesis by suggesting future improvements to our model that could help us get closer to the stated aim. Also, we were able to provide chip design based on the proposed architecture.

5.1 Summary

To summarize the whole work, we first started with the research and reviewed many papers and articles about the researches going on in the field of Reinforcement Learning, and found that there is negligible research done on reducing the processing time. This gave an ideology to us, that lead to use of hardware instead of software for Q-learning Algorithm which is the base of Reinforcement Learning. First, we tried to analyze the existing projects done on Q-learning, Deep Reinforcement Learning for Walking Robot (section II) describes the same. As MATLAB uses a lot more resources and as our aim was to implement hardware, we just needed training data, which can also be obtained from Python. So, we moved to Python and implemented to training part and succeeded to obtain the desired results. Now for testing part we started implementing the Hardware level coding on Xilinx ISE/Quartus Prime. Once the Hardware coding was done, we started working on layout in open-source tool Electric VLSI. Here we first started with standard cell layouts and their simulations. Then, we implemented large modules using those standard cells, and finally we designed layout for the whole algorithm. At last, we ended up with chip design, major part of our thesis. In this work, we have designed the layout using 180nm technology, as our priority for now is to implement the Q-learning algorithm on hardware and its chip level implementation. Once we start getting the desired outputs in 180nm technology, we will scale it down. As mentioned above we need to implement dynamic coding and also some addition of functionalities, seeing this there would be some changes in the hardware also which would lead to layout design changes. So, it would be more practical to stick to 180nm until we have incorporated all the changes and reached to desired outputs.

5.2 Future Work

5.2.1 Deep Reinforcement Learning

Various RL approaches and algorithms were evaluated during the research phase of this thesis in order to improve the system's performance. Some of them were adopted, while rest were not due to a lack of time or inability to produce better results.

In Q-learning, however, the value is changed only once per action. As a result, it's hard to handle complex problems efficiently in a broad state-action context because these many number of states and actions may be unfamiliar. In addition, the Q-table for rewards is pre-programmed and hence a significant quantity of storage memory is required. A substantial state-action memory is necessary in a multi-agent system with two or more agents, which causes issues. As a result, basic Q-learning algorithms are limited in their ability to achieve effective learning in a multi-agent context.

Furthermore, Deep Reinforcement Learning is a notion that is now being researched and deployed extensively. We didn't used it in our system because the problem's complexity did not need it. However, if we want to push the system closer to solving real-world problems, we must collaborate with it. Deep Q-Networks, whose superior performance has been demonstrated in problems such as vintage Atari games, could be one option. This would undoubtedly aid in the handling of more complicated settings and allow us to create more adaptable agents.

5.2.2 Different Environment with Dynamic Configurations

Addition of, policy control block to the proposed architecture which will govern the action mechanism can be added to eradicate the static decision problem. We propose a complex environment with larger dimensions, different position of obstacles and positioning of walls, as in a real situation. So as to make the agent stretch, using the available system and sufficient processing power.

Also, it would be fascinating to see how the agent reacts in a dynamic environment where the walls and obstacles are ordered differently each time it begins the task. This last option would necessitate the usage of Deep RL, as previously

noted, to aid the agent's learning process, but it would be a huge step forward in terms of real-world applications.

5.2.3 Real-Time Implementation of the Algorithm in a Robot

Finally, we also propose that the system be implemented in a real robot with the enhancements and characteristics previously discussed. To do so, the robot must meet many criteria. It needs sensors to detect whether it is hitting a wall or an object. We propose two options to accomplish this. The first is to programmatically avoid barriers whenever they are encountered. The second option is to include an avoidance action in addition to the basic actions available with the agent and push the agent to learn it. This action may consist of specific predetermined moves. Last but not least, the system must have enough computational capability to run and process.

These characteristics enable them to be used in more complex practical situations and with a wide range of real-time applications.

Publications from this Thesis

1. **Harsh Advani**, Jimmy Patel, Tapas Kumar Maiti, "Hardware-Efficient *Q*-Learning Accelerator for Robot Path Planning", **Published** in Springer, 5th International Symposium on Devices, Circuits and Systems, March, 2022.

2. Jimmy Patel, **Harsh Advani**, Tapas Kumar Maiti, "VLSI Implementation of Neural Network Based Emergent Behavior Model for Robot Control" **Submitted** in International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics, May, 2022.

3. **Harsh Advani**, Jimmy Patel, Tapas Kumar Maiti, "*Q*-Learning Accelerator Chip Design for Robot Path Planning", **in progress**.

4. Jimmy Patel, **Harsh Advani**, Tapas Kumar Maiti, "VLSI Implementation of Neural Network Driven Augmented FSM", **in progress**.

References

[1] Rajaraman, V. JohnMcCarthy — Father of artificial intelligence. Resonance, 19(3), 198–207, March 2014. doi:10.1007/s12045-014-0027-9

[2] J. Liu et al., "Artificial Intelligence in the 21st Century", IEEE Access, vol. 6, pp. 34403-34421, March 2018.

[3] K. Rusia, S. Rai, A. Rai and S. V. Kumar Karatangi, "Artificial Intelligence and Robotics: Impact Open issues of automation in Workplace", International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE), pp. 54-59, April 2021.

[4] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie and X. Zhou, "DLAU: A Scalable Deep Learning Accelerator Unit on FPGA", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 3, pp. 513-517, March 2017.

[5] Y. Kumar, K. Kaur and G. Singh, "Machine Learning Aspects and its Applications Towards Different Research Areas" IEEE International Conference on Computation, Automation and Knowledge Management (ICCAKM), pp. 150-156, April 2020.

[6] S. Ray, "A Quick Review of Machine Learning Algorithms", International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), pp. 35-39, October 2019.

[7] Y. Meng, S. Kuppannagari, R. Rajat, A. Srivastava, R. Kannan and V. Prasanna, "QTAccel: A Generic FPGA based Design for Q-Table based Reinforcement Learning Accelerators", IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 107-114, July 2020.

[8] A. Singh, N. Thakur and A. Sharma, "A review of supervised machine learning algorithms", 3rd International Conference on Computing for Sustainable Global Development (INDIACom), October 2016.

[9] M. A. El Mrabet, K. El Makkaoui and A. Faize, "Supervised Machine Learning: A Survey", 4th International Conference on Advanced Communication Technologies and Networking (CommNet), pp. 1-10, December 2021.

[10] R. Choudhary and H. K. Gianey, "Comprehensive Review On Supervised Machine Learning Algorithms", International Conference on Machine Learning and Data Science (MLDS), pp. 37-43, March 2018.

[11] N. Amruthnath and T. Gupta, "A research study on unsupervised machine learning algorithms for early fault detection in predictive maintenance", 5th International Conference on Industrial Engineering and Applications (ICIEA), pp. 355-361, June 2018.

[12] K. K. Htike and O. O. Khalifa, "Comparison of supervised and unsupervised learning classifiers for human posture recognition", International Conference on Computer and Communication Engineering (ICCCE'10), pp. 1-6, August 2010.

[13] H. Watanabe, M. Tsukada and H. Matsutani, "An FPGA-Based On-Device Reinforcement Learning Approach using Online Sequential Learning", IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 96-103, June 2021.

[14] S. Gu, E. Holly, T. Lillicrap and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates", IEEE International Conference on Robotics and Automation (ICRA), pp. 3389-3396, July 2017.

[15] W. Qiang and Z. Zhongli, "Reinforcement learning model, algorithms and its application", International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), pp. 1143-1146, September 2011.

[16] M. Naeem, S. T. H. Rizvi and A. Coronato, "A Gentle Introduction to Reinforcement Learning and its Application in Different Fields", IEEE Access, vol. 8, pp. 209320-209344, November 2020.

[17] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction", IEEE Transactions on Neural Networks, vol. 9, no. 5, pp. 1054-1054, September 1998.

[18] B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications", IEEE Access, vol. 7, pp. 133653-133667, September 2019.

[19] S. Spanò et al., "An Efficient Hardware Implementation of Reinforcement Learning: The Q-Learning Algorithm", IEEE Access, vol. 7, pp. 186340-186351, December 2019.

[20] Q. Huang, "Model-Based or Model-Free, a Review of Approaches in Reinforcement Learning," 2020 International Conference on Computing and Data Science (CDS), 2020, pp. 219-221, doi: 10.1109/CDS49703.2020.00051.

[21] C. G. Atkeson and J. C. Santamaria, "A comparison of direct and model-based reinforcement learning," Proceedings of International Conference on Robotics and Automation, 1997, pp. 3557-3564 vol.4, doi: 10.1109/ROBOT.1997.606886.

[22] I. Aleo, P. Arena and L. Patané, "SARSA-based reinforcement learning for motion planning in serial manipulators", International Joint Conference on Neural Networks (IJCNN), pp. 1-6, October 2010.

[23] M. A. Samsuden, N. M. Diah and N. A. Rahman, "A Review Paper on Implementing Reinforcement Learning Technique in Optimising Games Performance", IEEE 9th International Conference on System Engineering and Technology (ICSET), pp. 258-263, November 2019.

[24] A. Kekuda, R. Anirudh and M. Krishnan, "Reinforcement Learning based Intelligent Traffic Signal Control using n-step SARSA", International Conference on Artificial Intelligence and Smart Systems (ICAIS), pp. 379-384, April 2021.

[25] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey", IEEE Signal Processing Magazine, vol. 34, no. 6, pp. 26-38, November 2017.

[26] L. M. Reyneri, "Implementation issues of neuro-fuzzy hardware: going toward HW/SW codesign", IEEE Transactions on Neural Networks, vol. 14, no. 1, pp. 176-194, Jan. 2003.

[27] B. J. Leiner, V. Q. Lorena, T. M. Cesar and M. V. Lorenzo, "Hardware Architecture for FPGA Implementation of a Neural Network and Its Application in

Images Processing", IEEE Electronics, Robotics and Automotive Mechanics Conference (CERMA '08), pp. 405-410, October 2008.

[28] Zhenzehn Liu and I. Elhanany, "Large-scale tabular-form hardware architecture for Q-Learning with delays," IEEE 50th Midwest Symposium on Circuits and Systems, pp. 827-830, April 2007.

[29] Y. Meng, S. Kuppannagari, R. Rajat, A. Srivastava, R. Kannan and V. Prasanna, "QTAccel: A Generic FPGA based Design for Q-Table based Reinforcement Learning Accelerators", IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 107-114, July 2020.

[30] J. L. Lin, K. S. Hwang, W. C. Jiang and Y. J. Chen, "Gait Balance and Acceleration of a Biped Robot Based on Q-Learning", IEEE Access, vol. 4, pp. 2439-2449, May 2016.

[31] A. Konar, I. Goswami Chakraborty, S. J. Singh, L. C. Jain and A. K. Nagar, "A Deterministic Improved Q-Learning for Path Planning of a Mobile Robot", IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 43, no. 5, pp. 1141-1153, Sept. 2013.

[32] C. C. White and J. D. White, "Markov decision processes," Eur. J. Oper. Res., vol. 39, no. 1, pp. 1–16, Mar. 1989.

[33] G. (Guillem) Casado Rubert, "Performing a piece collecting task with a Q-Learning agent," Master's Thesis, University of Groningen, June 2019.

[34] Xu Wang, "Deep Reinforcement Learning - Case Study with Standard RL Testing Domains," Philips Research, Eindhoven University of Technology, March 2016.

[35] Even-Dar and Y. Mansour, "Learning rates for Q-learning," in Computational Learning Theory, D. Helmbold and B. Williamson, Eds. Berlin, Germany: Springer, 2001, pp. 589–604.

[36] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," J. Artif. Intell. Res., vol. 13, pp. 227–303, Nov. 2000.

[37] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in Proc. Adv. Neural Inf. Process. Syst., 2000, pp. 1057–1063.

[38] M. Irodova and R. H. Sloan, “Reinforcement learning and function approximation,” in Proc. FLAIRS Conf., 2005, pp. 455–460

[39] I. J. Sledge and J. C. Príncipe, "Balancing exploration and exploitation in reinforcement learning using a value of information criterion," 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2017, pp. 2816-2820, doi: 10.1109/ICASSP.2017.7952670.

[40] R. Patrascu and D. Stacey, "Adaptive exploration in reinforcement learning," IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339), 1999, pp. 2276-2281 vol.4, doi: 10.1109/IJCNN.1999.833417.

[41] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous Control with Deep Reinforcement Learning.", July 5, 2019. https://arxiv.org/abs/1509.02971.

[42] Website of electric vlsi design system, http://www.staticfreesoft.com/.

[43] Nanoelectronic Mixed-Signal System Design, no. 9780071825719 and 0071825711, McGraw-Hill Education, 2015.

[44] J. Sun and H. Jiao, "A 12T Low-Power Standard-Cell Based SRAM Circuit for Ultra-Low-Voltage Operations," IEEE International Conference on IC Design and Technology (ICICDT), pp. 1-4, August 2019.

[45] L. M. D. Da Silva, M. F. Torquato and M. A. C. Fernandes, "Parallel Implementation of Reinforcement Learning Q-Learning Technique for FPGA", IEEE Access, vol. 7, pp. 2782-2798, December 2018.

[46] R. Navajothi and A. K. Rahuman, "Implementation of high performance 12T SRAM cell," IEEE International Conference on Electrical, Instrumentation and Communication Engineering (ICEICE), pp. 1-6, December 2017.

[47] W. Qiang and Z. Zhongli, "Reinforcement learning model, algorithms and its application," IEEE International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), pp. 1143-1146, September 2011.

CHAPTER A
Standard Cell Layouts & Simulations

A.1 Inverter

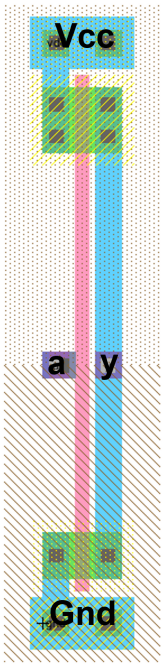

Figure A.1: Inverter layout.

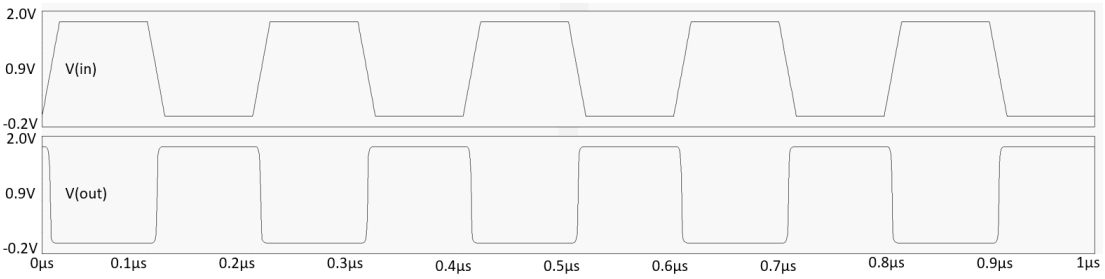

Figure A.2: Inverter waveform.

A.2 AND

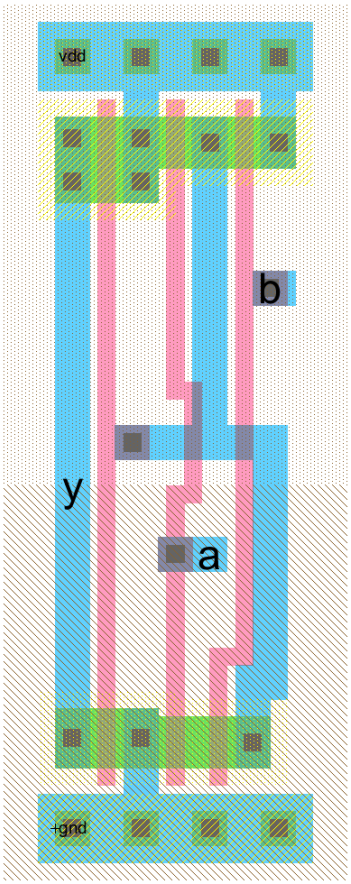

Figure A.3: AND Gate layout.

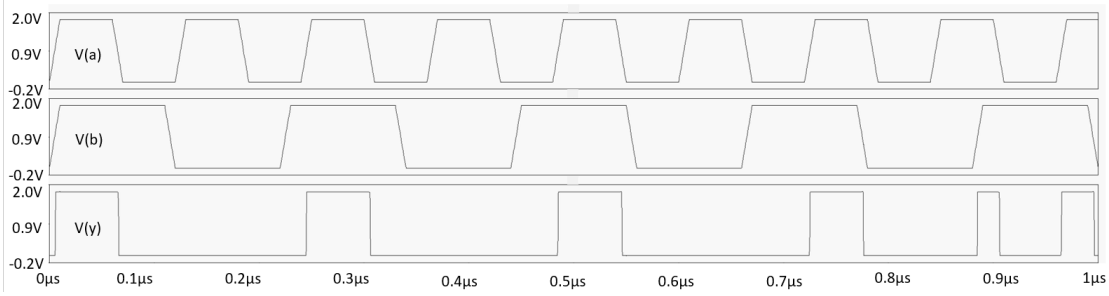

Figure A.4: AND Gate waveform.

A.3 OR

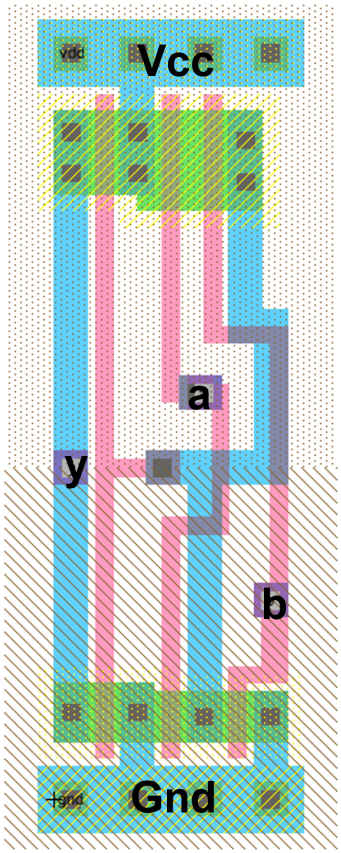

Figure A.5: OR Gate layout.

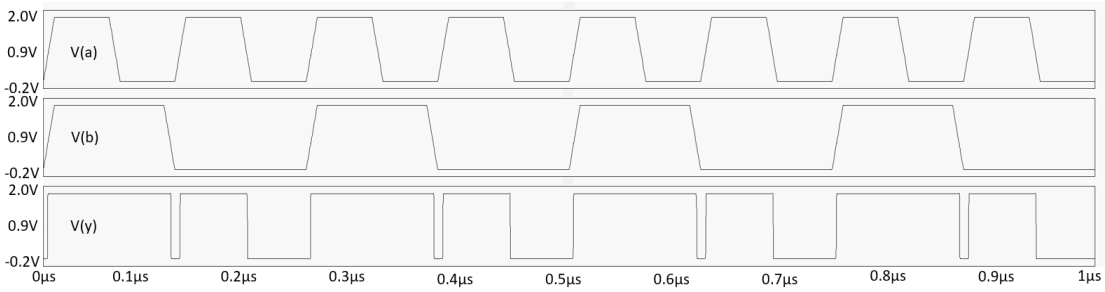

Figure A.6: OR Gate waveform.

A.4 NAND

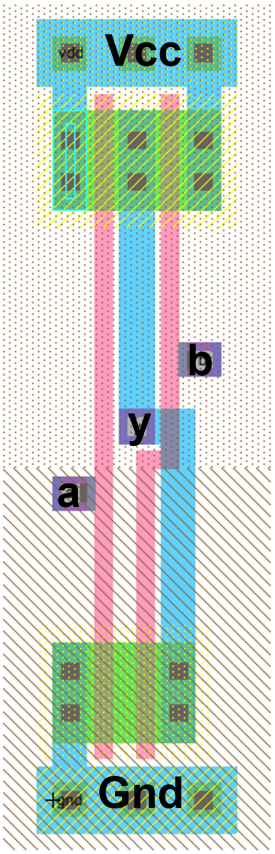

Figure A.7: NAND Gate layout.

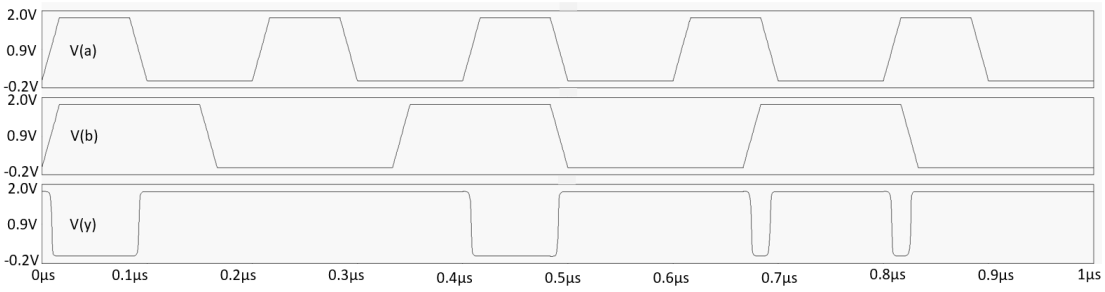

Figure A.8: NAND Gate waveform.

A.5 NOR

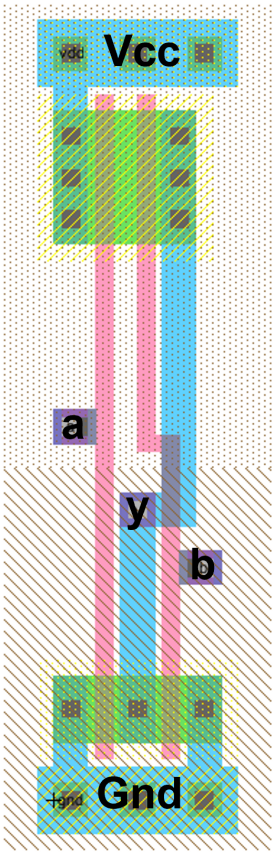

Figure A.9: NOR Gate layout.

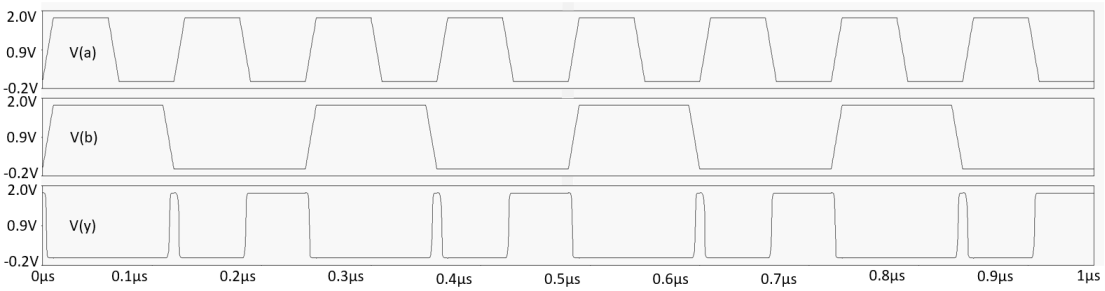

Figure A.10: NOR Gate waveform.

A.6 XOR

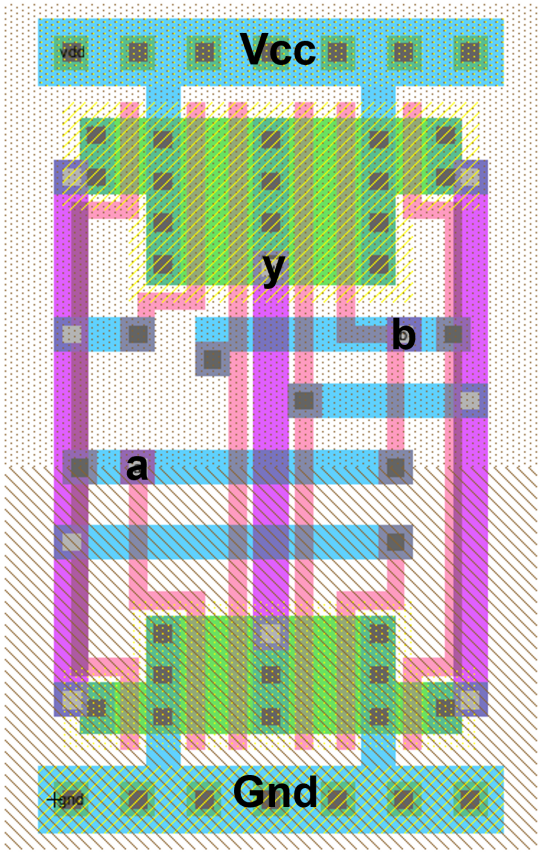

Figure A.11: XOR Gate layout.

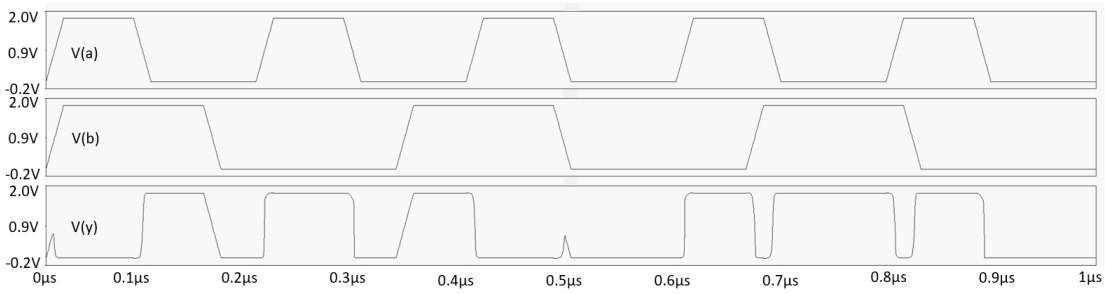

Figure A.12: XOR Gate waveform.

A.7 Half Adder

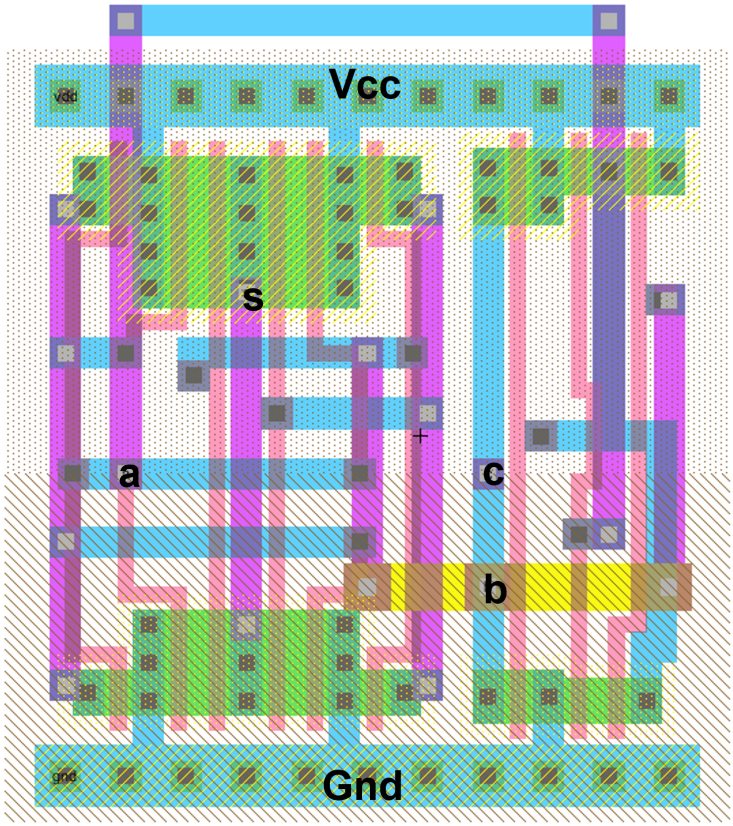

Figure A.13: Half Adder layout.

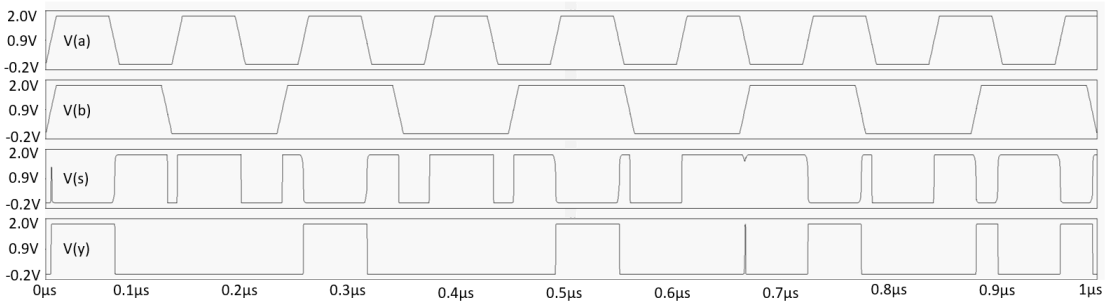

Figure A.14: Half Adder waveform.

A.8 Full Adder

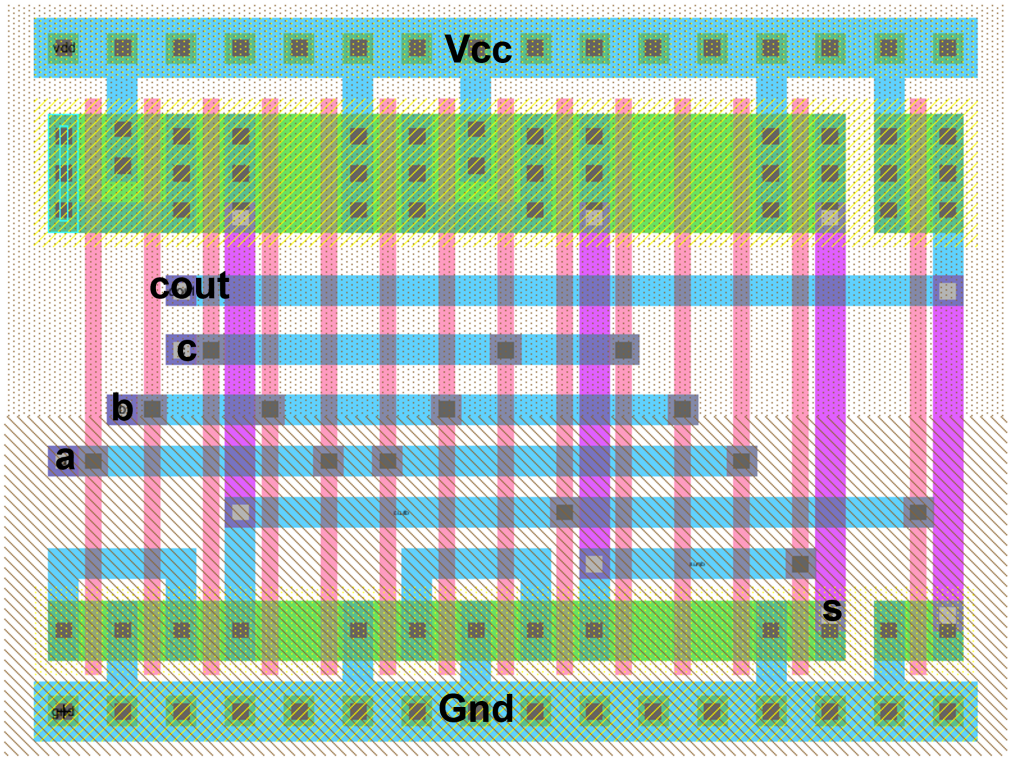

Figure A.15: Full Adder layout.

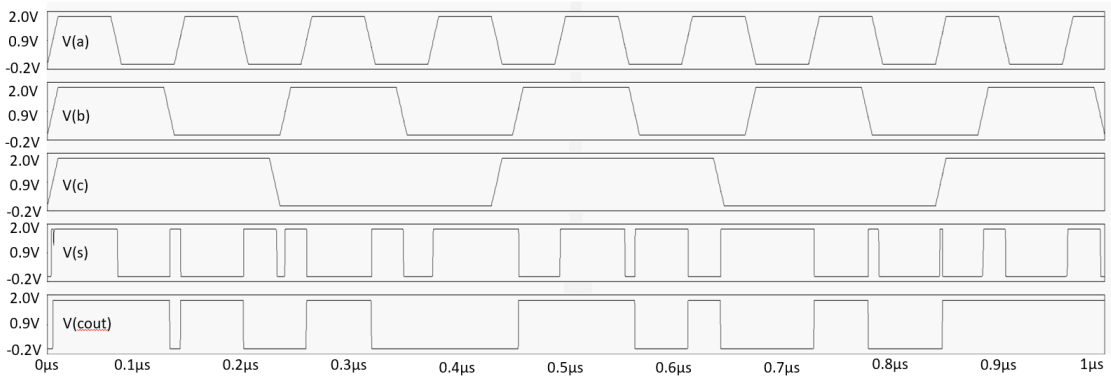

Figure A.16: Full Adder waveform.