IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 29 November 2022, accepted 20 December 2022, date of publication 28 December 2022,
date of current version 2 January 2023.

Digital Object Identifier 10.1109/ACCESS.2022.3232853

RESEARCH ARTICLE

FARANE-Q: Fast Parallel and Pipeline Q-Learning Accelerator for Configurable Reinforcement Learning SoC

NANA SUTISNA[1,2], (Member, IEEE), ANDI M. RIYADHUS ILMY[1], INFALL SYAFALNI[1,2], (Member, IEEE), RAHMAT MULYAWAN[1,2], (Member, IEEE), AND TRIO ADIONO[1,2], (Senior Member, IEEE)

[1]School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Bandung, West Java 40132, Indonesia
[2]University Center of Excellence on Microelectronics, Institut Teknologi Bandung, Bandung, West Java 40132, Indonesia

Corresponding author: Infall Syafalni (infall@ieee.org)

This work was supported in part by the Bandung Institute of Technology [Institut Teknologi Bandung (ITB)] Research Program 2021.

ABSTRACT This paper proposes a FAst paRAllel and pipeliNE Q-learning accelerator (FARANE-Q) for a configurable Reinforcement Learning (RL) algorithm implemented in a System on Chip (SoC). The proposed work offers flexibility, configurability, and scalability while maintaining computation speed and accuracy to overcome the challenges of a dynamic environment and increasing complexity. The proposed method includes a Hardware/Software (HW/SW) design methodology for the SoC architecture to achieve flexibility. We also propose joint optimizations on the algorithm, architecture, and implementation to obtain optimum (high efficiency) performance, specifically in energy and area efficiency. Furthermore, we implemented the proposed design in a real-time Zynq Ultra96-V2 FPGA platform to evaluate the functionality with an actual use case of smart navigation. Experimental results confirm that the proposed accelerator FARANE-Q outperforms state-of-the-art works by achieving a throughput of up to 148.55 MSps. It corresponds to the energy efficiency of 1747.64 MSps/W per agent for 32-bit and 2424.33 MSps/W per agent for 16-bit FARANE-Q. Moreover, the proposed 16-bit FARANE-Q outperforms other related works by an improvement of at least 1.23× in energy efficiency. The designed system also maintains an error accuracy of less than 0.4% with optimized bit precision for more than eight fraction bits. The proposed FARANE-Q also offers a speed up of processing time up to 1795× compared to embedded SW computation executed on ARM Zynq processor and 280× of computation of full software executed on i7 processor. Hence, the proposed work has the potential to be used for smart navigation, robotic control, and predictive maintenance.

INDEX TERMS Q-learning, reinforcement learning, HW accelerator, FPGA, SoC.

I. INTRODUCTION

Recently, Reinforcement Learning (RL) [1] has gained much attention in the Machine Learning area and it is widely employed in various applications, including robotics [2], [3], [4], [5], communication [6], [7], biomedical or healthcare [8], [9], and finance [10]. Advances in the RL algorithm can generate and evaluate the data through exploration without preassigned labels for the training data [1], [11]. It is a fully autonomous technique that can learn and take action individually according to the condition of the environment.

The RL algorithm has been recognized as a promising AI technique, particularly in a dynamic environment. However, this algorithm has intensive computation which will pose more challenges on time-constrained applications. A reinforcement learning system requires high-performance computation since it needs to perform data calculations and decide on an action in real time.

To fulfill this requirement, a high-performance hardware platform is required. ASIC (Application Specific Integrated Circuit) based implementation is preferable since this

The associate editor coordinating the review of this manuscript and approving it for publication was Marco Cococcioni.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

platform can deliver high-performance computing with lower power consumption. Unfortunately, the flexibility and scalability of this platform are limited. Programmable computing platforms such as Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) could be alternative solutions with a high degree of flexibility.

Several works on the RL system based on FPGAs have been presented. For example, Q-learning based systems are presented in [12], [13], and [14]. In [12], an efficient Q-learning algorithm is implemented using approximate multipliers and offers low complexity and low power consumption. Moreover, a parallel approach is proposed in [13] for increasing the throughput. To increase the accuracy, the RL system combined with deep learning to update the Q-value is presented in [14]. In short, all these works mainly focus on performance improvements, particularly in achieving high calculation throughput, low-complexity implementation, and low-power consumption.

In the previously mentioned works [12], [13], the analysis of bit precision for Q-learning calculation is not yet available. Therefore, the designed systems result in a higher complexity which also corresponds to higher power consumption. This can be a barrier to target implementation on a limited hardware budget (*e.g.,* low logic resource or small chip size). Additionally, from the perspective of system architecture, these implementations were only focused on HW accelerator design, resulting in limited reconfigurability and re-programmability when implemented. However, in [14], [15], and [16], analysis of fractional bits are presented and compared with existing works. The analysis is useful for finding the most suitable and efficient architecture for the corresponding device implementation such as FPGA, GPU, or other boards.

To address the limitations of the previous works, this paper proposes a FAst paRAllel and pipeliNE Q-Learning accelerator (FARANE-Q) for a configurable Reinforcement Learning (RL) system based on System on Chip design methodology. To enable scalability and reconfigurability, we utilize registers and memory blocks, which can be reprogrammed by the host Central Processing Unit (CPU) through the Advanced eXtensible Interface (AXI) bus interface. The register and memory block store parameters or configurations of the RL system. Specifically, the registers store parameters of rewards, learning rates, discount factors, and the number of episodes for the training process. In addition, the memory blocks can be configured with an initial state transition matrix, which represents the environment (*e.g.,* floor plan or layout). Finally, we also implement HW/SW co-design based on the SoC to realize the reconfigurability.

Furthermore, we perform bit optimization on the RL accelerator datapath by performing design exploration to obtain an optimum bit width, while also maintaining the accuracy of the Q-value. Several approaches such as providing flexible datapath design and implementing approximate computing are also proposed in this work. Finally, the main contributions of this work are listed as follows:

- We present FAst, paRAllel, and pipeliNE Q-learning accelerator (FARANE-Q) for a scalable architecture of RL accelerator by employing SoC architecture.
- We implement the proposed architecture in Avnet Ultra96-V2 platform [17] and perform a real-time evaluation in realistic use case applications, especially for smart navigation.
- We also perform joint design optimization by optimizing multiple layers of design, including algorithm, architecture design, as well as implementation aspects. Hence, a high-performance system in terms of high accuracy, low complexity, high throughput, and a power-efficient system can be achieved.
- Finally, the proposed system supports a system model that can be executed in the embedded CPU of the designed SoC. Hence, it can be used for HW/SW co-simulation and reduces the gap between the simulation and the actual deployment.

This paper is organized as follows: Section I introduces the work. Section II shows brief explanations of reinforcement learning and Q-learning algorithms. Section III explains the detailed architecture of the proposed RL accelerator design. Section IV discusses the experimental setup, evaluation, and performance results. Finally, Section V concludes the work.

II. BACKGROUND

A. REINFORCEMENT LEARNING

Reinforcement learning (RL) is a machine learning approach that simulates learning through trials and errors [1]. The learning simulation is carried out by an agent inside a defined environment. This agent can execute a pre-defined set of actions that can affect the state of the environment. Then, based on the resulting environment state, the agent will receive a feedback value called a reward. This learning process, which is illustrated in Fig. 1, continues to iterate so that the agent can determine the best course of action that can result in the maximum cumulative reward. The learning process is focused on goal-directed learning based on the agent's interaction with its environment.

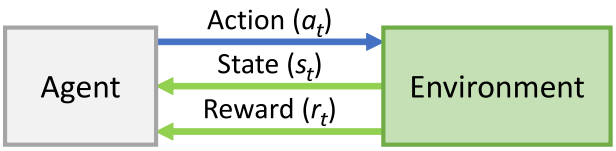


FIGURE 1. Illustration of RL system.

Recently, research on reinforcement learning has been conducted in many fields, such as in IC implementations [18], [19], [20], Deep Neural Network (DNN) applications [21], [22], [23], [24], [25], [26], as well as Network on Chip (NoC) applications [27], [28], [29]. In [18], a deep reinforcement learning processor is implemented in 28nm CMOS technology. Multiple DNNs are used to observe the policy training procedure in the RL system. Moreover, a memristor-based reinforcement learning system is proposed in [19].

In [20], a 55nm time-domain mixed-signal (TD-MS) neuromorphic accelerator is proposed to perform the Q-Learning. The TD-MS improves the efficiency of energy for mobile robots.

Reinforcement learning is also used to optimize DNN applications. In [21], optimization of hardware resources for DNN is proposed by using reinforcement learning. Moreover, reinforcement learning is used to search the optimal quantization-voltage-frequency (QVF) policy before DNNs [22]. In [24], on-chip deep reinforcement learning is implemented by a network-on-chip. Finally, the works in [25] and [26] present DNNs with reinforcement learning to optimize the available resources *e.g.,* GPU and FPGA, respectively.

To some extent, reinforcement learning algorithms are also applied in NoCs. In [27], distributed reinforcement learning is implemented by using in-switch computing. Furthermore, the works in [28] and [29] explore some methods using reinforcement learning for efficient energy and efficient framework in NoCs, respectively.

A value function in RL is represented by a look-up table, where all state-action pair values are stored individually. Thus, memory consumption is linear to the sum of states and actions [30]. This limitation can be addressed by replacing the look-up table with a function approximator, such as DNN [31], [32]. The neural network will then be trained using a reinforcement learning algorithm. This technique is called Deep Reinforcement Learning due to the merging of DNN and RL technologies. However, DRL has some drawbacks. First, it does not guarantee the convergence of the learning. Second, in the case of medium-sized state space, RL outperforms the execution speed of DRL due to the latter having more complex architecture [33]. Thus, for small-sized state space applications, RL is preferred.

B. Q-LEARNING

Q-Learning is an off-policy RL algorithm to learn action-value policy which has been proven to converge to an optimal policy [1]. Q-Learning stores the action values in an $N \times Z$ matrix, namely the Q-Matrix, where N is the number of states and Z is the number of possible actions. The Q-value for every state and action are updated using Eq. (1) as follows:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha\Big((r_t + \gamma \max_a Q(s_{t+1}) - Q(s_t, a_t)\Big) \quad (1)$$

where

- $Q_{new}(s_t, a_t)$ is the new Q-value,
- $Q(s_t, a_t)$ is the current Q-value,
- α is the learning rate,
- r_t is the received reward,
- $\gamma \max_a Q(s_{t+1}) - Q(s_t, a_t)$ is the discounted future reward for s_{t+1}.

Algorithm 1 shows the step-by-step for the Q-Learning process. Q-Learning accepts several input parameters such as the number of episodes, learning rate coefficient, and discount factor. In the beginning, the Q-matrix will be initialized for every state and action. Q-Learning is then run for the number of episodes. In each episode, the state of the agent will be initialized to an initial value. The agent will take action according to the policy, observe the reward and next state information from the environment, and update the Q-value for the action taken from that state using Eq. (1). This process will be repeated until the agent reaches the terminal state. Then, the learning process will be repeated for the number of episodes. After all the number of episodes has been executed, it will give an output of the Q-Matrix.

Algorithm 1 Q-Learning Algorithm

Input: Number of episodes, learning rate (α) and discount factor (γ)
Output: Q-Matrix

```
Initialize the Q-value matrix for every state and action
while count < episode do
    Set state to initial state s ← s_0
    while s ≠ s_terminal do
        Choose action A from state S
        Do action A, observe reward R and next state S'
        Find the new Q-value by Eq. (1)
        Move to next state s ← S'
    end while
    count ← count + 1
end while
```

As one of the most commonly applied RL algorithms, Q-learning algorithm has been proposed to solve different problems in various fields [34]. In the field of industrial process, computer networking, and robotics, it is used to improve the adaptability and performance of systems in the mentioned fields. With its online learning capabilities, many systems can learn to adapt into different real conditions without the need to redesign. It is also used to solve many optimization problems such as dynamic load management of electricity demand, efficient resource distribution of mobile games, Dynamic Job Shop Scheduling (DJSS), and device placement for neural network computation [34]. Other notable applications include smart systems, particularly smart traffic light systems [35], [36], [37].

C. RELATED WORKS

Prior to this work, there have been a few hardware implementations of the Q-learning algorithm on FPGA boards. In this section, we will discuss previous works of the Q-Learning accelerator to provide fundamentals in designing the proposed system.

The first complete implementation of the Q-learning accelerator was proposed by Da Silva et al. [13]. Their work proposed an architecture that focuses on parallelization based on the number of available states. This architecture calculates Q-values for all available state-action pairs simultaneously and then selects the Q-value of the agent's current state

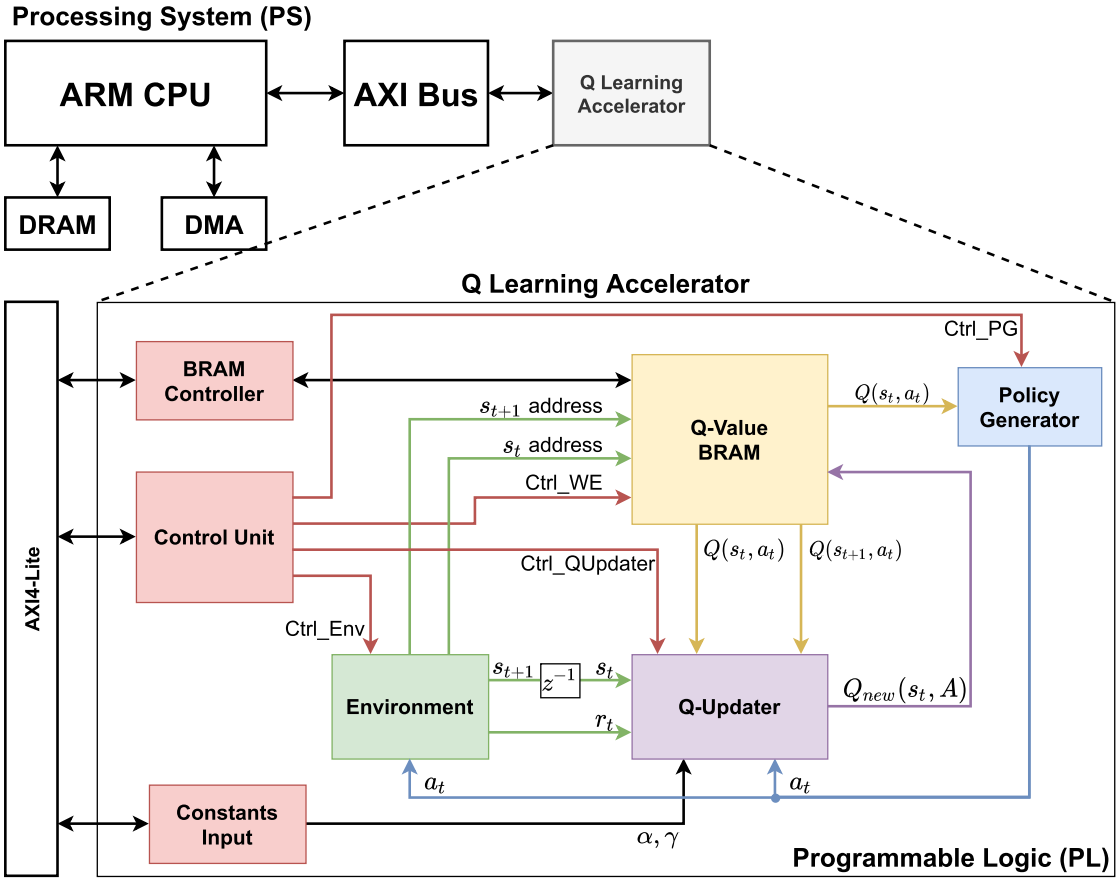


FIGURE 2. Overview Architecture of the proposed Q-Learning Accelerator at the System on Chip Level.

and action. All of the calculated Q-values are stored using registers in fixed-point format. The architecture was implemented in a Xilinx Virtex-6 FPGA board and was able to achieve a throughput of 13.4 MSps for an environment with 132 possible states and 4 actions using 20-bit Q-values.

Another architecture was proposed by Spanò et al. [12]. The proposed architecture separates the state transition function of the environment and instead receives the current and future state as input of the architecture. Also, the architecture does not define specific policy generator modules to choose the agent's action. The architecture chooses the state and action before calculating the Q-values allowing it to reduce the number of computation modules to one. This architecture uses on-chip RAM blocks to store its Q-Matrix. The number of RAM blocks used is equal to the number of the agent's actions and each memory block has a depth equal to the number of available states. With this configuration, the architecture can read Q-values for all available actions in a given state simultaneously. This work also proposed the use of approximate multipliers using barrel shifters when performing multiplication with learning rate and discount factor. The architecture was implemented in a Xilinx ZCU106 FPGA board and was able to achieve a throughput of 112 MSps for an environment with 128 possible states and 4 actions using 32-bit Q-values. This architecture can also be adapted to accelerate the SARSA algorithm.

In [33], Meng et al. proposed a pipelined Q-learning accelerator. The accelerator is divided into four-stage architecture. The current action is selected in the first stage, and the next state is computed. This stage also reads reward and Q-values stored in the RAM blocks. In the second stage, the next action is chosen for the next state based on the given update policy, and the Q-value for the next state-action pair is read from the memory. The main computation is performed in the third stage to generate the updated Q-values. Lastly, the updated Q-value is written back to the Q-Matrix memory in the fourth stage. Also, an update is made to the maximum Q-value table if the updated Q-value is higher than the maximum Q-value of the current state. This use of a maximum Q-value table replaces the max-tree architecture used to compute the maximum Q-values used in [12]. The proposed architecture was implemented in a Xilinx Virtex-7 FPGA board and was able to achieve a throughput of 189 MSps for an environment with 64 possible states and 8 actions using 16-bit Q-values.

Another notable work was proposed by Gankidi et al. in [14]. The work proposed a DRL accelerator for space rovers applications. The proposed architecture was implemented in a Xilinx Virtex-7 FPGA board and was able to

achieve a throughput of 2.34 MSps for an environment with 24 possible states and 9 actions using fixed point Q-values. A detailed comparison between our proposed architectures and the architectures mentioned in this section is discussed in Section IV-G.

III. PROPOSED HARDWARE ARCHITECTURE

A. SoC ARCHITECTURE

The proposed SoC Architecture mainly consists of Processing System (PS) and the hardware accelerator implemented in Programmable Logic (PL), as depicted in Fig. 2.

The PS, which includes CPU, DMA, DRAM, and the AXI4 Bus, acts as the interface between the user and the PL. The CPU executes a program to initialize all necessary system configurations, such as the environment data and learning parameters. This system configuration will then be sent to the accelerator via the AXI4 Bus [38]. The CPU is also responsible for starting the PL learning process by asserting a start signal using the same AXI4 interface. Then, the CPU will wait until it receives an interrupt signal from the PL which indicates the completion of the learning process. Finally, the learning outcomes received from the PL via the AXI4 interface will be displayed by the CPU to the user.

The mentioned system configurations are stored in the Control and Status Registers. The Control and Status Registers consist of 7 registers with a data width of 32 bits. The Control parameters are written by the CPU and read by the PL while the Status parameters are written by the PL and read by the CPU. The bitmap for the Control and Status Registers are shown in Fig. 3. The control parameters are listed as follows:

- **Number of episodes**: Number of episodes to be run.
- **Start**: The start signal of the accelerator with a rising edge.
- **Initial state**: Maze's initial state.
- **Terminal state**: Maze's terminal state. If the agent reaches this state, one episode will be concluded and the state will be reset to the initial state.
- **Learning rate** and **Discount factor**: Learning rate and discount factor for the corresponding Q-value update.

While the status parameters are listed as follows:

- **Total clock cycles elapsed**: To calculate the execution time of the accelerator.
- **Number of Q-value updates**: To calculate the number of Q-value updates in the learning process.

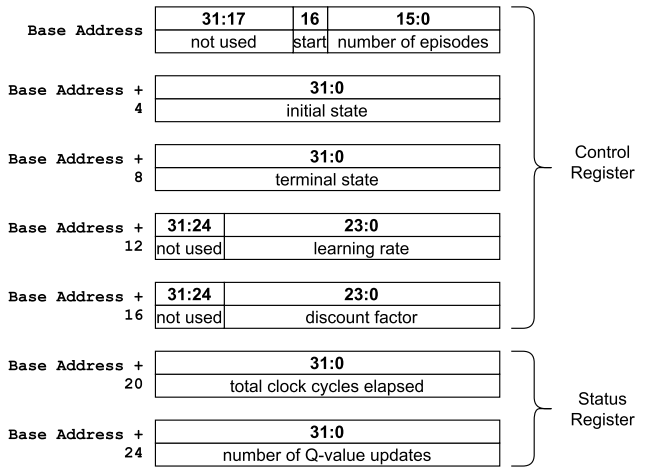


FIGURE 3. Control and status registers bitmap.

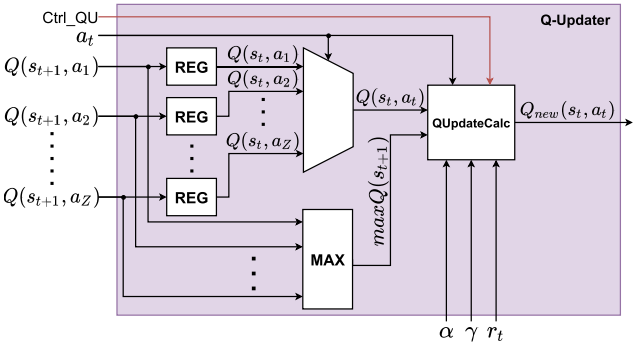


FIGURE 4. Q-Updater module architecture.

B. PROPOSED Q-LEARNING ACCELERATOR

The proposed Q-Learning architecture is inspired by the Q-learning hardware accelerator in [12]. This architecture is constructed by the following modules: 1) Q-Updater 2) Policy Generator, 3) Environment, 4) Q-Value BRAM, and 5) Control Unit.

1) Q-UPDATER

Fig. 4 shows the architecture for the Q-Updater module, based on a proposed architecture in [12]. This module updates the Q-Value of the state-action pair based on Eq. (1) using Fixed Point calculation.

The *MUX* module takes Z inputs at Q-Values for every action and selects Q-Value based on the action taken (a_t). Meanwhile, the *MAX* module generates a maximum value from the Z inputs of $Q(s_{t+1}, a_t)$ values. The MAX module uses a binary tree architecture. The output of each module is then passed to *QUpdateCalc* which implements Eq. (1) to update the Q-Value.

The implemented *QUpdateCalc* module uses barrel shifters as approximate multipliers. This approach is considered since its implementation requires low complexity logic which is only shifting operation, instead of using a direct multiplier and other approximate methods such as an approximate compressor. In addition, since the operation is a multiplication of one data with a constant value, we may exploit the characteristic of barrel shifters which employ several operations of shifting. Hence, this implementation will consume much lower logic resources (or gate counts) compared to other approaches.

Specifically, the values of α and γ are approximated to their nearest sum of powers of two. The number of powers of two is equal to the number of shifters used as approximate multipliers. Since the values of α and γ are ranging

from 0 to 1, only negative powers are used to represent the approximate values. For 3 barrels, the number can be approximated as:

$$B = 2^{-i} + 2^{-j} + 2^{-k} \tag{2}$$

Therefore, if we want to calculate $Y = A \cdot B$, then

$$Y = A \cdot 2^{-i} + A \cdot 2^{-j} + A \cdot 2^{-k} \tag{3}$$

The term 2^{-x} is implemented using the right shifter, where x is the number of shifts. The diagram for the approximate multiplier using 3 barrel shifters is shown in Fig. 5.

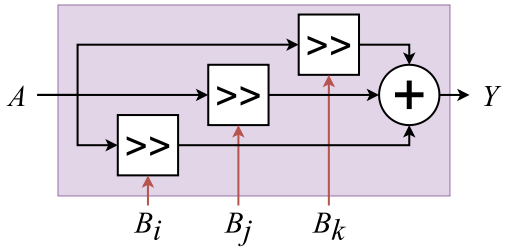


FIGURE 5. Approximate multiplier using 3 barrel shifters.

2) POLICY GENERATOR

Fig. 6 shows the architecture of the Policy Generator (PG) module, which is similar to [39]. The module selects the current action a_t for the agent based on an action-selection policy, which is ϵ-greedy policy. This policy allows the module to generate a random action with the probability of ϵ or a greedy action with the probability of $1 - \epsilon$.

A greedy action is generated by the Greedy Action Selector, which compares Q-Values for all actions of the current state ($Q(s_t, a_1$ through $Q(s_t, a_Z)$). An action with the highest Q-Value will be chosen as the greedy action. The Greedy Action Selector uses a MAX module, similar to the one used in the Q-Updater, to determine the highest value of the Q-Values.

Meanwhile, a random action is generated using a random value from a Pseudo-Random Number Generator (PRNG) module. The Least Significant Bits (LSBs) of the generated random value represent the random action where its width corresponds to Z number of actions.

The generated random value also acts as a comparison value for the action-selection policy. This comparison value is compared with a threshold value (σ). The PG module selects the greedy action if the comparison value is less than σ. On the other hand, if the comparison value is more than or equal to σ, then random action is selected. This comparison happens in a Comparator which generates a selector signal for the multiplexer. The threshold value is determined based on Eq. (4)

$$\sigma = \lfloor \epsilon \cdot (2^L - 1) \rfloor, \tag{4}$$

where ϵ is the epsilon value and L is the bit width of the PRNG.

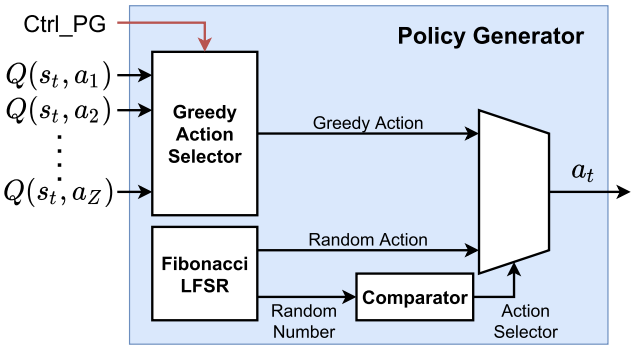


FIGURE 6. Policy Generator module architecture.

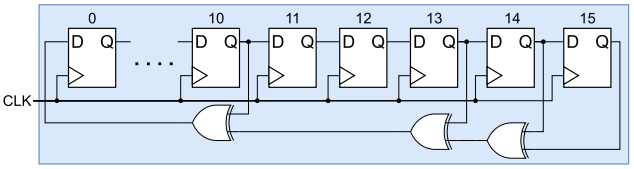


FIGURE 7. 16-bit Fibonacci Linear Feedback Shift Register (LFSR) [40].

The PRNG used to generate the required random values is implemented using a Fibonacci Linear-Feedback Shift Register (LFSR). Fibonacci LFSR has the advantage that the next L bits are immediately visible in the shift register. This is because all the shift register stages (except for the first one) get their inputs only from the previous stage. Thus, a random value can be generated every clock cycle.

To achieve a maximum period of random values series, the placement of taps required for the LFSR feedback needs to be configured with respect to the LFSR bit size. As shown in Fig. 7, we choose to implement a 16-bit LFSR. Accordingly, to achieve maximum period, the taps are placed at the 16th, 14th, 13th, and 11th bits [40].

3) ENVIRONMENT

The environment that we used consists of a 2-dimensional grid world with a total state of $I \times J$, where I denotes the first dimension and J denotes the second dimension of the grid world, resulting in a total N of states. There could be obstacles (depicted as walls) that separate each state as illustrated in Fig. 8. When the agent hits the wall, it will receive punishment in the form of a negative reward. On the other hand, when the agent reaches the goal, it will receive a reward and the episode will be reset. In this environment, the agent is targeted to reach a goal state. To encourage the agent to take the least amount of steps to reach the goal, every time the agent moves, it will receive a small negative reward. The reward function is implemented using these three rules:

1) If the agent moves to a new state, then $r_t = \mu_1$.
2) If the agent hits a wall, then $r_t = \mu_2$.
3) And if the agent reaches the terminal state, then $r_t = \mu_3$.

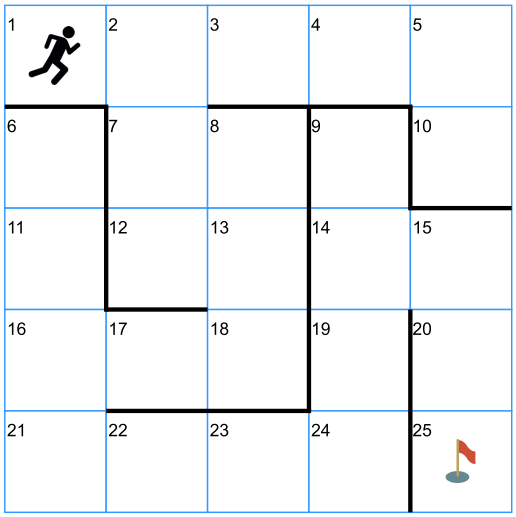


FIGURE 8. Grid-world illustration with N = 25, State 1 as Initial State, and State 25 as Terminal State.

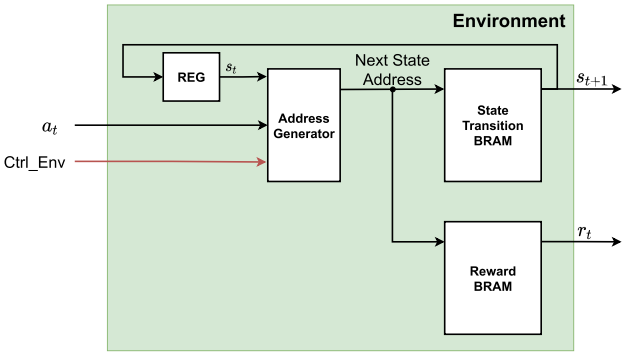


FIGURE 9. Environment module architecture.

State transition and reward matrix for the maze environment are stored in BRAM as shown in Fig. 9. State transition and reward matrix are generated by the PS and are written to the BRAM during the initialization of the learning process. For the terminal state, the next state will be the initial state and the reward is 0. The address generator module will generate the address which is used to retrieve the next state and reward data from the State Transition and Reward BRAMs.

The next state and the reward values for each state-action pair are stored in BRAM with the configuration illustrated in Fig. 10. The values are sorted based on the number of environment states (N) and by number of actions (Z). The first Z memories stores values for state 1, followed by the next Z memories for state 2 and so on until it reaches state N. Each Z memories stores a value for action 1, followed by the value for action 2 and so on until it reaches Z action. Based on this configuration, the BRAM size required for the environment module can be calculated using Eq. (5) with W as the bit width. Note that there are two values(r_t and s_{t+1}) stored in this BRAM so the calculation is multiplied by 2.

$$BRAM\ size = 2 \times W \times N \times Z \tag{5}$$

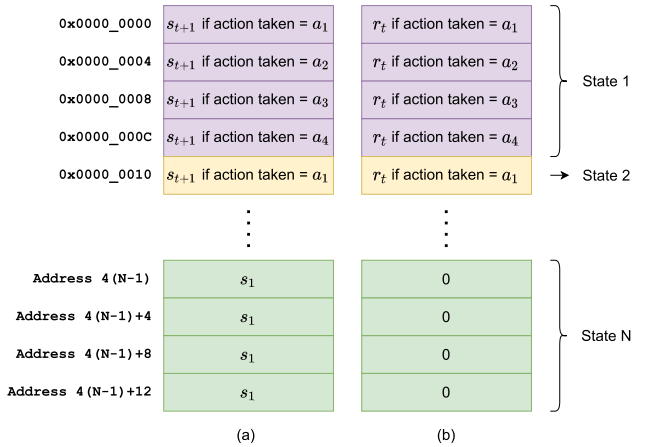


FIGURE 10. Address and value configurations for (a) Next State BRAM and (b) Reward BRAM for $Z = 4$, initial state of State 1, and terminal state of State N.

4) Q-VALUE BRAM

Fig. 11 shows how the Q-Values are stored in BRAM. We use Z numbers of BRAM to store each state-action. Each action BRAM stores Q-Values for every possible states. The depth of the BRAM depends on N. Q-Value BRAMs are also accessible from the PS through the BRAM Controller, which can be called from the main program.

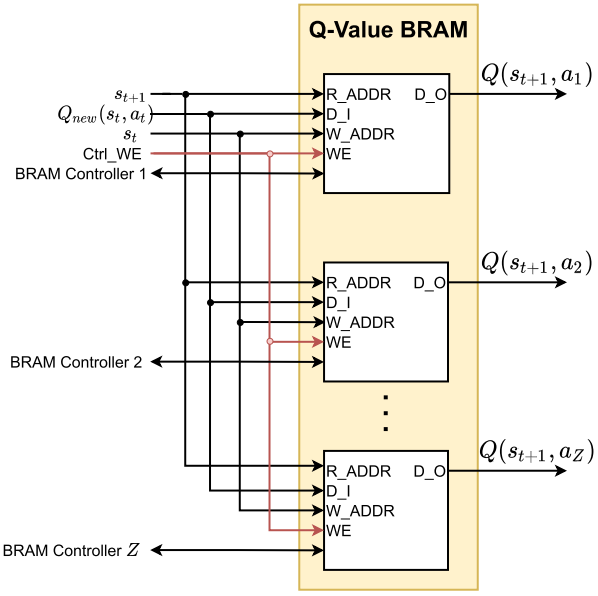


FIGURE 11. Q-Value BRAM module architecture.

5) CONTROL UNIT

Control Unit module arranges all blocks in the accelerator by providing control signals to the respected modules based on Finite State Machine (FSM) as shown in Fig. 12 and Fig. 13. In the Main FSM, there are 4 states, *IDLE*, *INIT*, *RUNNING*, and *DONE_LEARNING*. The system will enter the state *IDLE*

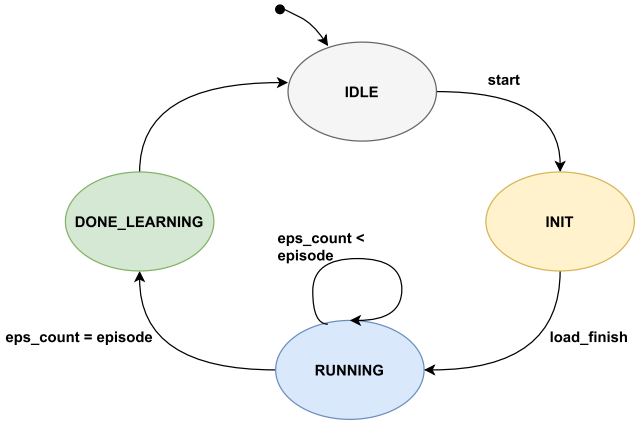


FIGURE 12. Main Finite State Machine of the Architecture.

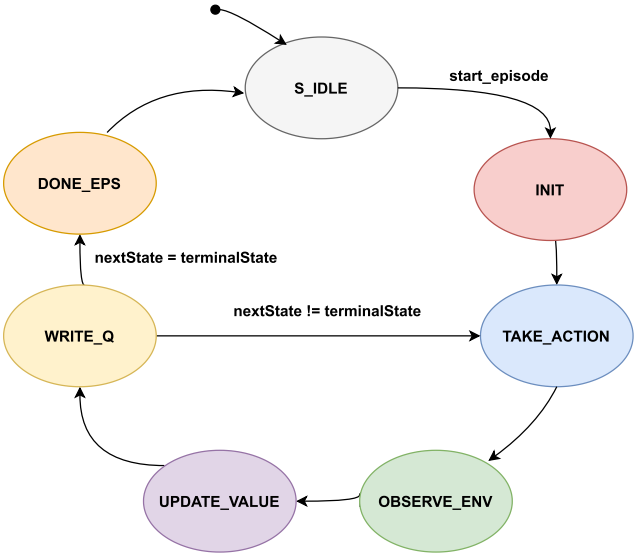


FIGURE 13. Level 1 Finite State Machine of the Architecture.

when it is reset. In *INIT* state, the system will wait for the PS to write the number of episodes, learning parameter (α), discount factor (γ), state transition matrix, and reward matrix.

In the *RUNNING* state, the system starts learning and remain learning until the number of episodes reached. The inner part of *RUNNING* state is shown in Fig. 13. The system takes an action, observes the reward and next state. After that, the system updates the Q-Value and writes it back. It keeps repeating these actions until it reaches the terminal state. Once it reaches the terminal state, the counter for the number of episodes will be incremented. Finally, it will repeat the steps until it reaches the number of episodes.

After finishes learning, the system enters the final state (*DONE_LEARNING*), where the PL inserts interrupt signal to the PS to indicate that the learning process has finished.

C. ARCHITECTURE IMPROVEMENT

In order to improve the performance of the baseline parallel architecture (referred as V1), specifically on the computation speed and learning efficiency, we proposed an improved design, referred as V2 (FARANE-Q). The improvements include: 1) employing pipeline architecture and 2) applying dynamic threshold (decreasing-ϵ) method in determining state action. The proposed optimizations are detailed in Fig. 14.

1) PIPELINE ARCHITECTURE

The design of V1 updates each Q-value in 4 clock cycles, as shown in timing chart in Fig. 15. Based on the V1 architecture, which again is created based on [12], a pipelining method is implemented to create V2. Thus, V2 updates the Q-value every clock cycle, as shown in Fig. 16.

The achieved throughput for V1 and V2 are formulated in Eq. (6) and Eq. (7), respectively.

$$\Gamma_{V1} = \frac{X}{(4X + 3E + 1)} \times F_s \tag{6}$$

$$\Gamma_{V2} = \frac{X}{(X + 2E)} \times F_s \tag{7}$$

where

- X is the number of times Q-value is updated,
- E is the number of episodes,
- and F_s is the operating clock frequency.

The total number of clock cycles required to complete one learning cycle also varies. V1 requires a total of $4X + 3E + 1$ clock cycles. The constant $3E$ is added because, for each episode, a transition time of three clock cycles is required. An additional constant of 1 is also added to the total clock cycles due to a setup time of one clock cycle at the beginning of the learning cycle. Meanwhile, V2 requires a total of $X + 2E$ clock cycles. The constant $2E$ is added because, after each episode, a transition time of two clock cycles is required. Fig. 14 shows the pipeline staging implemented to increase throughput in V2.

2) DYNAMIC EPSILON WITH DECREASING-ϵ METHOD

As stated in Section III-B3, V1 uses the ϵ-greedy algorithm to determine the agent's actions. The algorithm divides the agent's actions into exploration and exploitation. Exploration happens when the agent chooses one of the available actions randomly, while exploitation happens when the agent chooses an action with the maximum Q-value. In [41], decreasing the rate of exploration as the learning progress can be beneficial for the system performance. Decreasing the rate of exploration means the agent starts with a high exploration rate and as the learning episode increases, the exploration rate will decrease until it reaches zero. Thus, during the early stage of learning, the agent tends to explore many alternatives due to its high rate of exploration. As the rate of exploration decreases, the possibility of exploitation increases and becomes absolute at the latter stages of learning. Value-Difference Based Exploration (VDBE) is one of the available methods to implement decreasing-ϵ [42], which adapts the exploration parameter of ϵ-greedy in dependence with

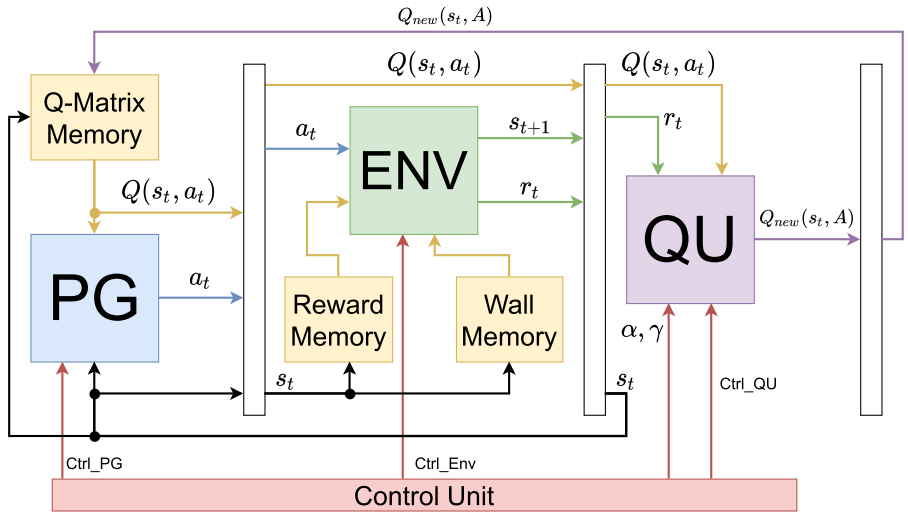


FIGURE 14. Three stage pipeline in V2 (FARANE-Q).

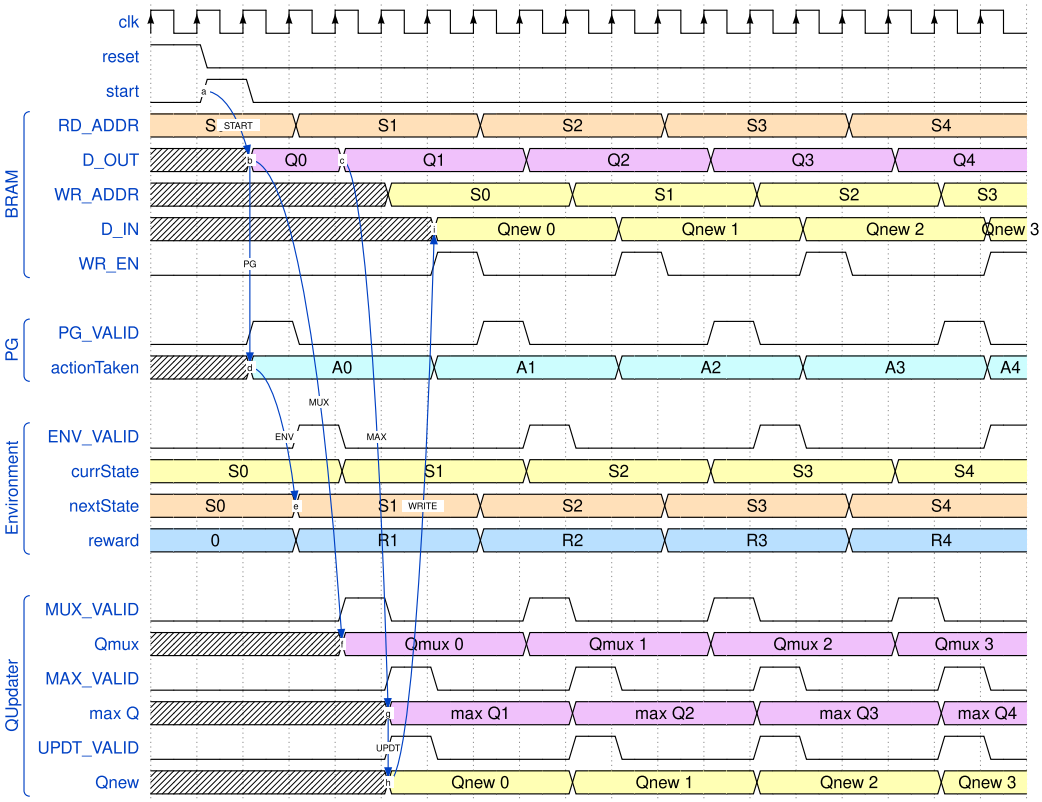


FIGURE 15. Timing diagram for RL-Parallel Architecture (Our's V1 (Parallel)).

the temporal-difference error observed from value-function backups.

This method uses a complex equation that uses many hardware resources. Instead, with the same concept of decreasing-ϵ, we implement a simpler method that uses the number of episodes to update the value of epsilon. Thus, ϵ in Eq. (4) is replaced with ϵ_δ as described in the following equation:

$$\sigma = \lfloor \epsilon_\delta \cdot (2^L - 1) \rfloor, \tag{8}$$

where L is the number of bits in LFSR and $\epsilon_\delta = \delta \cdot \epsilon$ is the dynamic epsilon with decreasing-ϵ method. Moreover, the ratio δ is calculated as follows:

$$\delta = \frac{\psi - \kappa}{\psi}.$$

where

- ψ is the maximum number of episodes,
- and κ is the number of current episodes or counted episodes.

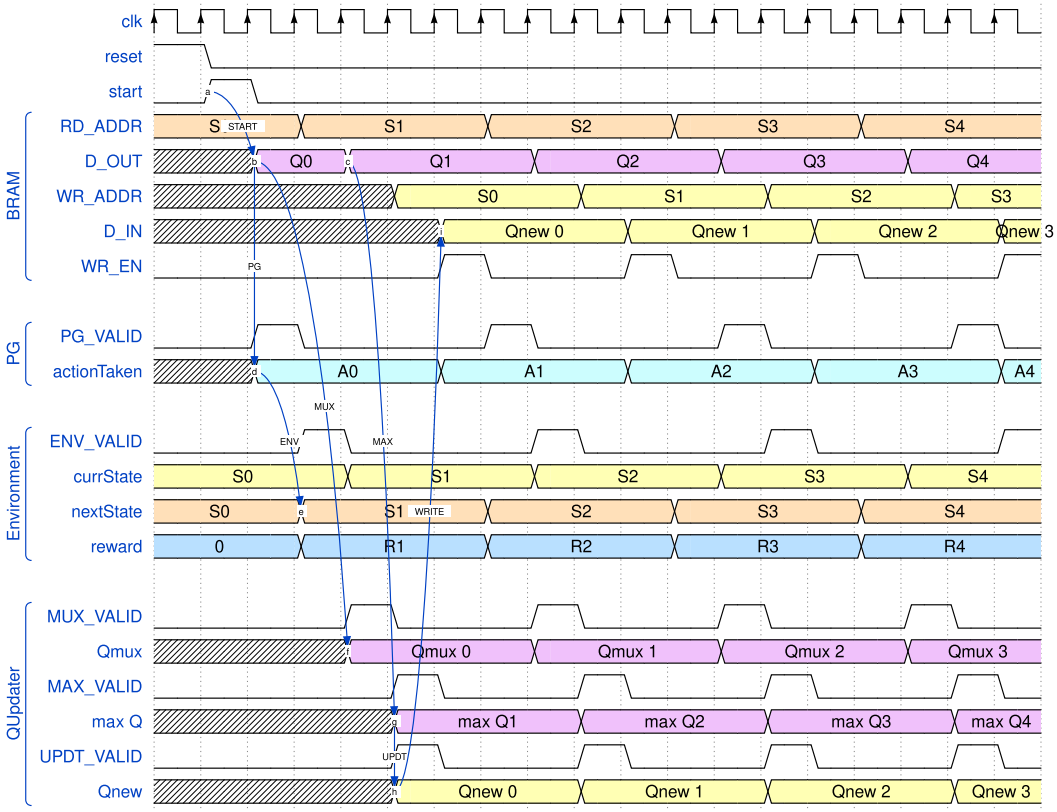


FIGURE 16. Timing diagram for RL with Parallel and Pipeline Architecture (Our's V2 (FARANE-Q)).

Note that the dynamic threshold depends on the value of ϵ_δ that changes according to the number of learning episodes κ. The threshold value σ decreases as κ increases until it reaches zero when the maximum episode is reached ($\psi = \kappa$). Since exploration is carried out when the generated random value is smaller than the threshold, the probability of exploration will continue to decrease as learning episodes are increasing. While in V1, as already stated, the probability of exploration is determined by the value of ϵ in Eq. (4) and the value is constant throughout the learning process.

D. ARCHITECTURE COMPARISON WITH STATE OF THE ART

For the sake of clarity, we summarize our proposed architecture and provide a comparison to the most recent architecture of Q-learning accelerator [12].

First, in terms of system-level design, our proposed architecture is intended as a System-on-Chip architecture instead of a hardware accelerator core. The proposed design does not only connect the HW accelerator to the AXI bus, but it also includes a design of instruction set, an FSM control system, and a memory map to provide configurability. In addition, the proposed architecture also supports HW/SW co-simulation. Hence, we can perform a comprehensive evaluation that covers algorithm performance evaluation, RTL functional simulation as well as FPGA verification. The proposed

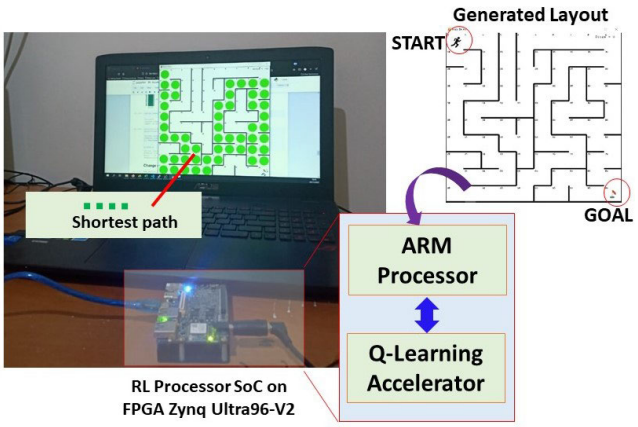


FIGURE 17. Hardware setup using Avnet Ultra96 V2 Board.

design is essentially different with the previously designed by Spanò et al. [12], where the previous design mainly focused on the discussion of the hardware accelerator core with no explanation details on the SoC architecture side.

Second, in the point of view of hardware accelerator, while the main block components are built on a common fundamental (an environment, a Q-updater, and a policy generator), our proposed design has several improvements on the modules as follows:

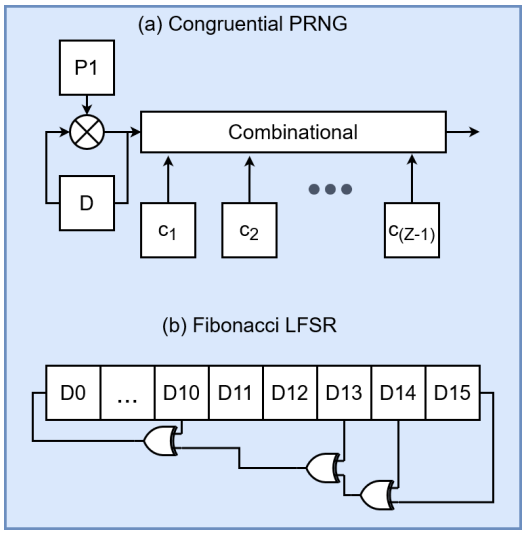


FIGURE 18. Random Number Generator Comparison between (a) Congruential PRNG [12], [13] versus (b) Proposed Fibonacci LFSR.

- **Data Flow Control:** Our design (FARANE-Q) supports both parallel and pipeline techniques on the architecture. The proposed design can maintain throughput at update sample in one clock cycle and hence it obtains a high throughput system. Moreover, we provide detail timing diagrams and their throughput formulations for both parallel and pipeline methods.
- **Random Number Generator:** Our design employs a Fibonacci LFSR based random generator that allows fast calculation and also requires low-complexity implementation (*e.g.* only XOR logic), while the random generator in [12] is implemented based on congruential PRNG, similar as initially described in [13]. This congruential PRNG based random generator is more complex since it involves more logics to implement multiplications and various combinational logics as shown in Figure 18.
- **Interpreter and Environment:** Our design is designed as a grid-wall environment. Our proposed design implements grid-wall environment which is different and more complex than the environment in [12]. In our proposed environment, a specific interpreter architecture is required in order to address this case. In terms of complexity, the wall is more complex than grid-based-only environment, since for the same value of state size, the grid-wall actually implements higher number of state spaces with irregular layout, and it actually increases the number of actual number of states. On the other hand, in the grid based, the obstacles have been considered as subset of states.
- **Policy Generator:** In our design, we proposed ϵ-greedy based policy generator, where a threshold value is calculated dynamically based on decreasing-ϵ criteria as described in Eq. (8). Meanwhile, the work in [12] uses ϵ-greedy with a constant value of ϵ.

To conclude the architecture comparison, we provide Table 1 which shows the comparisons between the Q-updater module in [12] and our proposed design. To obtain a fair comparison in terms of the resource utilization data and achievable maximum clock frequency, we implemented our design on the same target device, which is Xilinx Zynq UltraScale+ MPSoC ZCU106 XCZU7EV-FFVC1156-2-E FPGA Board. For the dynamic power consumption, we use an activity factor of 0.5 while the rest of the parameters remain the default parameters. Furthermore, we also show a comparison between the 16-bit and 32-bit variations of the architectures.

TABLE 1. Architecture comparison with state of the art.

Architecture Features	Spanò [12]		Proposed	
Design Level	Standalone		Standalone	
Pipeline	Yes		Yes	
Hardware Implementation Results				
Platform	Xilinx ZCU106			
N×Z	128×4			
Bit-Width	16	32	16	32
Clock (MHz)	158	112	158	158
LUT	333	682	217	427
Registers	258	606	135	275
LUTRAM	160	320	0	0
BRAM	NA	NA	2	2
DSP	3	5	0	0
Power (mW)	80	142	46	121

While we use the same high-level design of the Q-updater module, the implementation results show that our design uses less hardware resources compared to [12]. The implementation results of our design do not use any DSPs. Moreover, instead of using LUTRAM, we use BRAM to store the Q-Matrix. With lower resource utilization, our design was able to achieve lower power consumption despite achieving a clock frequency of 158 MHz for the 16-bit and 32-bit architectures.

IV. PERFORMANCE RESULTS AND DISCUSSION

A. EXPERIMENTAL SETUP

This section discusses the experimental setup we used to validate the proposed design. The designed system was implemented on the Xilinx Ultra96 V2 FPGA platform [17]. The experimental setup includes an FPGA, a display, and a host PC, as shown in Fig. 17.

To access the FPGA, we used the PYNQ platform as a hardware abstraction layer, which enables the use of Python programming language and libraries to program the FPGA [43]. The PYNQ platform is installed as a Linux boot image in the board's SD card memory. The platform then can be accessed through Jupyter Notebook which is run on a PC connected to the FPGA through a USB connection.

The host PC accesses the processing system on the FPGA by connecting to the Jupyter Notebook through the web browser and navigating to the available static IP address of the board [44]. The PS will then run a full software Python-based RL model which simulates the RL computations performed by the FPGA. Then, PS will initiate the data environment for learning on PL and start the hardware learning process. Finally, the performances of the model on PS and the architecture on PL can be evaluated together to get

the performances of the algorithm/architecture. This HW/SW co-simulation offers a comprehensive evaluation where the software simulation acts as a benchmark for comparing the hardware computation results.

For the simulation, we consider a maze navigation problem, where the agent has four possible directions ($Z = 4$) of actions (up, down, left, and right). While previous studies such as [12], [13], and [33], also perform simulations with eight possible actions, our maze simulation differs in terms of moving diagonally, which is the same as taking two of four possible actions (up/down and right/left) sequentially. Therefore, we only consider simulations with four possible actions. However, the proposed system essentially could be expanded to support a various number of states and actions by easily modifying the design. The general system parameters used for the experiments are presented in Table. 2.

TABLE 2. System parameters for experiment and simulation.

Parameter	Values
Learning Coefficient (α)	0.625
Discount Factor (γ)	0.875
Data Representation	Fixed-Point
Bit Width	$[n, b]$

B. HARDWARE RESOURCE

Table 3 shows the implementation results for V1 while Table 4 shows the implementation results for V2. Both designs are implemented with two different datapath (16 and 32 bits) and four different numbers of states (N). In addition, both architecture shown here are implemented at operating frequency of 125 MHz.

Since we employ approximate multipliers, our design does not require DSP at all to calculate the new Q-Value with an exception in V2 (FARANE-Q) which uses a single DSP to implement the multiplication required for threshold changes. The approximate multipliers are implemented as LUTs. Therefore, the usage of LUT and FF in the FPGA board increases along the bit-width. Additionally, as N increases, the depth of the required BRAM also increases. This is justified due to the increase of N increasing the memory usage to store the Q-Matrix, reward table, and state transition matrix.

It can be noted that the implementation results presented are shown only for an agent with 4 available actions ($Z = 4$). This is due to our experiments focused on solving maze navigation cases with 4 possible actions. However, our design can be modified to support other cases which required more than 4 actions by increasing the parallelization of the design. The modification will increase the amount of BRAM used, the number of binary tree stages in the MAX module, the number of inputs to the multiplexer used to select the Q-value, and the required bit to represent all available actions.

C. POWER CONSUMPTION

Table 5 shows the dynamic power consumption for V1 and V2. The data shown are obtained from the implementation report of both architectures. Variations of implementations are similar to the one used for IV-B.

TABLE 3. FPGA utilization of V1 for 16-bit and 32-bit Q-learning accelerator block for $Z = 4$ and 125 Mhz clock frequency.

# Bit	N	LUT	LUTRAM	FF	BRAM	DSP
16	512	2164	20	1930	17	0
	1024	2173	20	1932	21	0
	2048	2173	20	1938	32	0
	4096	2174	20	1944	62	0
32	512	2690	20	2074	17	0
	1024	2675	20	2076	21	0
	2048	2699	20	2107	32	0
	4096	2706	20	2088	62	0

TABLE 4. FPGA utilization of V2 (FARANE-Q) for 16-bit and 32-bit Q-learning accelerator block for $Z = 4$ and 125 Mhz clock frequency.

# Bit	N	LUT	LUTRAM	FF	BRAM	DSP
16	512	2111	25	2177	20	1
	1024	2114	25	2178	22	1
	2048	2106	25	2187	28	1
	4096	2125	25	2198	56	1
32	512	2493	24	2350	22	1
	1024	2497	24	2351	24	1
	2048	2486	24	2360	32	1
	4096	2513	24	2371	64	1

Similar to the resource utilization results, as N increases, the dynamic power consumption also rises. While compared to each other, V1 consumes more power than V2. This is justified due to the higher LUTs required for V1.

D. ACCURACY

To evaluate the accuracy of the proposed algorithm, we created a Python simulation as our benchmark to produce Q-Matrix with a double-precision floating-point. This simulation serves as a golden reference for comparison with the Fixed-Point representation of the Q-Matrix read from the BRAM. Each architecture has its software simulation due to a difference in action-policy implemented. In this evaluation, the accuracy metric is measured as an error rate, which can be calculated by using the following equation:

$$err = \frac{\sum_{i=0}^{N}\sum_{j=0}^{Z}\frac{|Q(s_i,a_j)-\hat{Q}(s_i,a_j)|}{|\hat{Q}(s_i,a_j)|}}{N \cdot Z}, \tag{9}$$

where N is the number of states, Z is the number of actions, $Q(s_i, a_j)$ is the Q-Matrix read from BRAM, and $\hat{Q}(s_i, a_j)$ is the Q-Matrix from simulation used as the reference.

The experiment is carried out with 2 different maze environments with a different total number of states (N). Each maze environment is run through each architecture with a randomized start and terminal state. Then, the resulting Q-Matrix is compared to Q-Matrix from each architecture's golden reference to calculate the error rate. The experiment is then repeated with another randomized start and terminal state and the whole is reiterated several times.

The experiment is also carried out with a different variation of fraction bits which are detailed in Table 6. Both V1 and V2 use a signed 32-bit fixed-point representation for this experiment. The fixed-point format is used to represent all

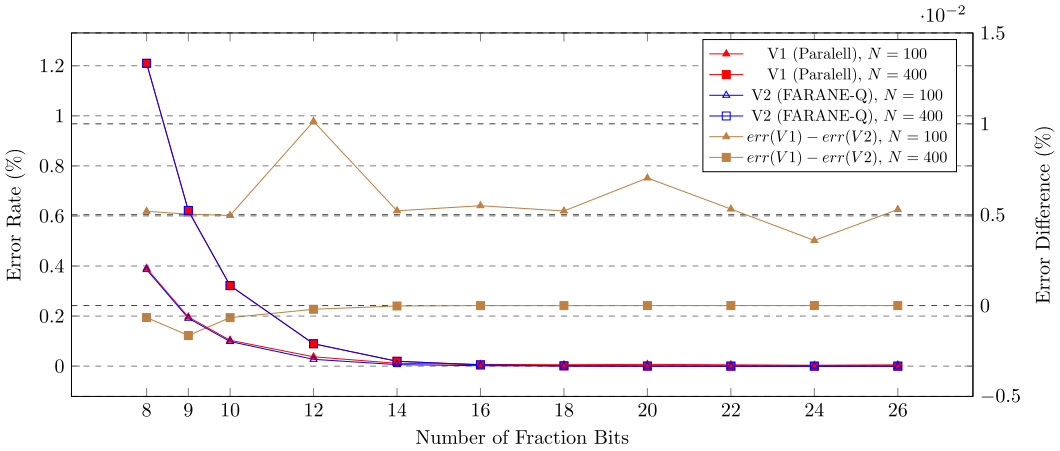


FIGURE 19. Average error rate for each variation of fraction bits throughout different maze environment sizes.

TABLE 5. Power consumption of V1 and V2 for 16-bit and 32-bit Q-learning accelerator block for $Z = 4$ and 125 Mhz clock frequency.

# Bit	N	Power (mW)	
		V1	V2
16	512	72	48
	1024	82	52
	2048	105	62
	4096	163	101
32	512	85	68
	1024	94	72
	2048	117	90
	4096	177	149

TABLE 6. Parameter variation for accuracy experiment.

Parameter	Variations
N × Z	[100×4, 400×4]
Fraction Bits	[8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26]

the variables directly used for Q-value calculation, which are $Q_{new}(s_t, a_t)$, $Q(s_t, A)$, $Q(s_{t+1}, A)$, and r_t. This format is the same throughout all of the modules in both architectures while other variables used separate bit representations which match each required range and value representation. With the described configuration, changing the fraction bits width only requires the user to re-initialize the reward matrix with the desired fraction bits width through the PYNQ interface. Thus, there is no need to re-synthesize both architectures. From our software simulation, we found out that the minimum and maximum value for the Q-Matrix is -16.89 and 10.0. Thus, the minimum integer bit needed is 6 bits (5 bits integer with a sign bit).

Fig. 19 shows the average error rate results for each fraction bits variations throughout several iterations of the mentioned experiment procedure. The same error rate percentage is obtained for V1 and V2. For testing on the maze with $N = 100$, the error rate is in the range of 0.4% to 0.005% with an error difference of about 0.5%. As for the results in the maze with $N = 400$, the error rate is in the range of 1.2% to almost 0% with an error difference of about 0.2%. Despite this, the error difference between the two results is still very small (<1%) for both maze cases.

The experimental results also show that the error rate decreases as the number of fraction bits increases. The change in error rate is not significant when the number of fraction bits exceeds 16. However, when the fraction bit is less than 16, it can be seen that the error rate for $N = 100$ is smaller than the result for $N = 400$. This is because, with a higher number of possible states, the number of states with a different number of visits is also higher, which leads to a higher difference in Q-values. However, despite the error rate, the produced Q-matrix is still optimal to guide the agent to reach the terminal state.

E. COMPUTATION TIME

An experiment was conducted to calculate the time required to carry out learning on both implemented designs. The RL algorithm was executed with several variations in the number of learning episodes. A software model of the Q-learning algorithm is also run on a 2.6 GHz Intel Core i7 CPU and an ARM CPU as standard for comparison. Fig. 20 shows the results of the computational time that has been measured. In comparison with the two software model calculations, a significant reduction in computation time was obtained. Compared to computation run on i7 CPU, an average reduction in computation time by 70× for V1 and 199× for V2 (FARANE-Q) was achieved. While compared to computation run on ARM CPU, an average reduction in computation time by 452× for V1 and 1277× for V2 (FARANE-Q) was achieved.

Meanwhile, when compared to each other, V2 (FARANE-Q) is faster than V1 (Parallel). Where the computation time of V2 (FARANE-Q) is 2.8× shorter than the computation time of V1. It's also worth noting that (FARANE-Q) computes faster with an average of 1.4× more Q-updates than V1, which is shown in Fig. 21.

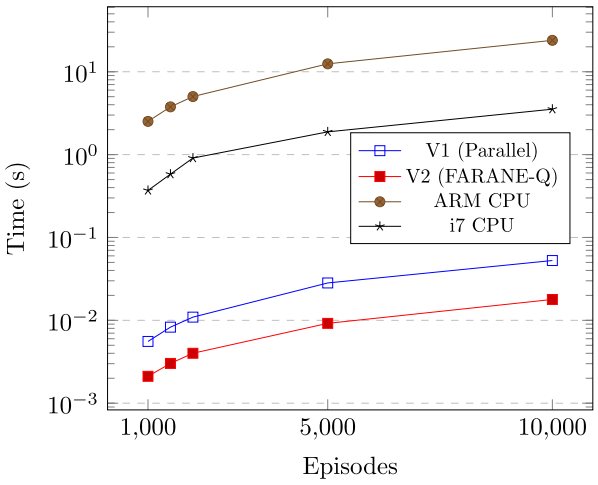


FIGURE 20. Execution time of Maze problem with only software and with accelerator.

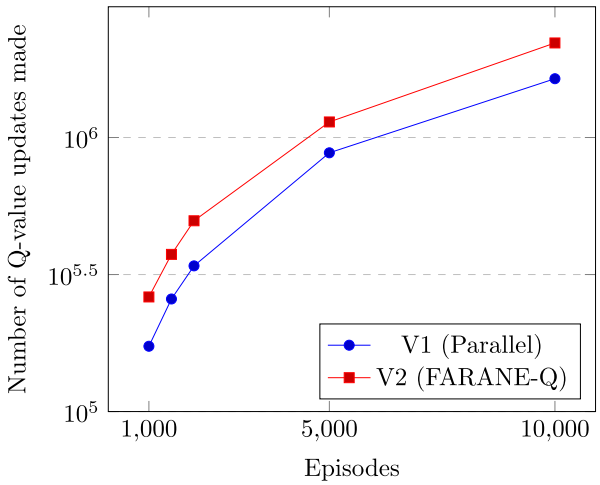


FIGURE 21. Number of Q-value updates made for various learning episodes.

TABLE 7. Throughput calculation for V1 for 125 Mhz clock frequency.

Learning Episode	Q-value Updates	Learning Time (ms)	Eq. (6) (MSps)	Throughput (MSps)
1000	173130	5.564	31.11	31.11
1500	257493	8.276	31.11	31.11
2000	340124	10.932	31.11	31.11
5000	879446	28.263	31.12	31.12
10000	1639043	52.690	31.11	31.11
Average			31.11	31.11

TABLE 8. Throughput calculation for V2 for 125 Mhz clock frequency.

Learning Episode	Q-value Updates	Learning Time (ms)	Eq. (7) (MSps)	Throughput (MSps)
1000	261946	2.112	124.052	124.052
1500	374635	3.021	124.006	124.006
2000	496753	4.006	124.001	124.000
5000	1139632	9.197	123.912	123.911
10000	2213146	17.865	123.880	123.879
Average			123.970	123.969

Although the two architectures are working on the same problem, the difference in the number of writes is due to the different action-policy used by each architecture. As explained in Section IV-D, V1 uses ϵ-greedy with an ϵ value of 0.3. The value is hard-coded directly in the architecture. Therefore, the probability of V1 performing exploration is less than its probability to perform exploitation. Although the probability of exploration stays the same throughout the learning process, the overall number of steps taken is less than in V2 (FARANE-Q). In contrast to V1, V2 (FARANE-Q) uses decreasing-ϵ, where the epsilon value decreases from 0.875 to 0 as the number of episodes increases. Because of this, the probability to perform exploration early in the learning process is almost absolute and requires significantly more steps. Therefore, V2 (FARANE-Q) updates more Q-values than V1 because the number of actions it takes during the learning process is significantly more than V1.

Additionally, from these two results, we can evaluate the throughput results, where the throughput is calculated as the number of updated Q-values within learning time (at a specified episode). At the same clock frequency of 125 MHz, the V1 can achieve an average throughput of 31.11 MSps, while the average throughput of V2 (FARANE-Q) can achieve up to 123.969 MSps. This achievable throughput of V2 (FARANE-Q) is approximately 3.98× improvement from V1. Table 7 and 8 shows the detailed throughput calculations for each learning episode.

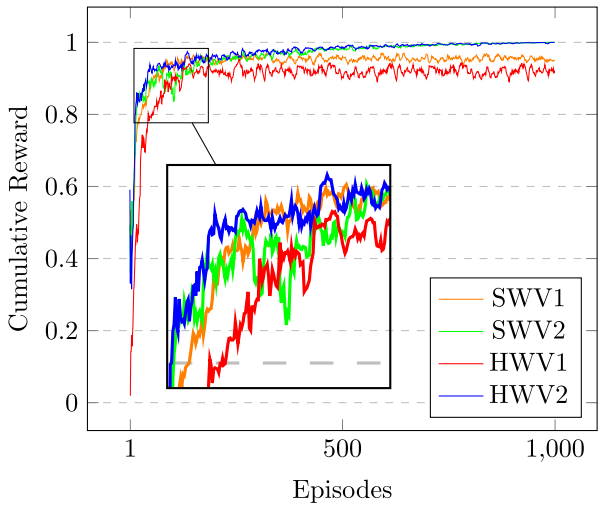


FIGURE 22. Cumulative rewards.

F. CUMULATIVE REWARD

To further show convergent learning outcomes, cumulative reward experiments were carried out. The cumulative reward is defined as [23]:

$$R = \sum_{i=1}^{T} \gamma^{i} r_i,$$

where T is the total time steps for an episode and γ is a discount factor ranged from 0 to 1. In this case, we take the discount factor $\gamma = 0.875$.

As we know, the agent receives rewards according to the actions taken and in one learning episode the agent will perform many actions. Thus, the accumulation of rewards received by the agent in one episode can show how the resulting policy performance is. A positive reward is only given to the agent when the selected action makes the agent reach its goal state while other actions will be given negative rewards. As the number of actions needed by the agent to reach its goal state increase, the cumulative reward value decreases. Therefore, a minimum action will show a maximum cumulative reward.

Fig. 22 shows how the cumulative reward changes as the learning episodes increase. The two graphs, in red and orange, show the learning processes performed in V1 (Parallel), while the green and blue graphs show the learning processes performed in V2 (FARANE-Q). A comparison is also shown with software-based learning that model each architecture, labeled as SWV1 and SWV2. Moreover, the hardware-based learning processes are labeled as HWV1 and HWV2. Both V1 and V2 (FARANE-Q) show similar results compared to their respective software models, even with the use of approximate multipliers. In the early learning episodes, the cumulative rewards are minimum as the agent tends to choose an action randomly resulting in many actions required to reach its terminal state. As the episode increases, the cumulative reward also increases and converges to the maximum cumulative reward possible.

When compared to each other, the effect of different threshold equations is shown in Eq. (4) for V1 (Parallel) and Eq. (8) for V2 (FARANE-Q). Since V1 uses a constant threshold, the probability of random actions stays constant throughout the learning process. This results in a relatively flat line after around 150 learning episodes were performed. A small fluctuation with a margin of $\pm 3\%$ is due to random actions still taken by the agent in V1, whereas V2 (FARANE-Q) uses a dynamic threshold which decreases the probability of random actions as the learning progress for V2. This results in a gradual increase in cumulative reward as the learning episode progress. Near the end of the learning process, the cumulative reward accumulated on V2 does not vary as much compared to the beginning of the learning process. This is due to only greedy actions being chosen at episodes nearing the end of the learning process.

G. COMPARISON WITH OTHERS

Table 9 shows the comparison of our two architectures with other works. To get a fair comparison from the available data sources for each work, we tried to select the results with the closest number of N and Z to each other. In addition, we also calculated area efficiency and energy efficiency for each work as a form of normalization. Eq. 10 shows how to calculate the area efficiency, and Eq. 11 shows how to calculate the energy efficiency [45] as follows:

$$A_{eff} = \frac{Throughput}{\frac{LUT}{1000} \times G}, \tag{10}$$

$$E_{eff} = \frac{Throughput}{Power \times G}, \tag{11}$$

where A_{eff} and E_{eff} are the area efficiency and energy efficiency for an agent, respectively, and G is the number of agents. The better performances are indicated by the larger values of the area and/or energy efficiencies. Note that the more agents are working in parallel, the more Q-values are produced.

It should be noted that through Section IV-B to IV-F, the V1 and V2 (FARANE-Q) use a clock frequency of 125 MHz. However, in this section, we implemented the V2 (FARANE-Q) with a clock frequency of 150 MHz for both 32 and 16 bits fixed precision. The detailed utilization values and the achieved throughput are shown in Table 9.

The work in [14] uses a neural network to estimate the Q-value and replace the Q-matrix. Their work shows results from two different architectures. One design uses a single neuron, while another uses a multi-layer perceptron (MLP). We chose to compare our design with the single neuron architecture because it is the only one with detailed hardware utilization data. Compared to ours, the design with single neuron architecture required more resources in LUT and achieved lower clock frequency.

The work in [13] is the first complete implementation of an accelerator for RL. Their work implements an architecture focusing on parallelization based on the number of available states. While the implementation allows the computation of Q-values for each available state simultaneously, only one will be updated to the Q-matrix. Thus, this parallelization seems redundant. The architecture uses only LUT, DSP, and register. The LUT and DSP are used to implement the processing module, while the register is mainly used to store the required values. For 30-bit width configuration, with $N = 132$ and $Z = 4$, the work in [13] requires a much higher amount of hardware resources. While there is no available information on the achieved clock frequency, the architecture achieves a throughput of 13.4 MSps, which is lower compared to both of our architectures. With its higher number of LUTs and lower throughput, the work area efficiency is much lower than ours. However, it should be noted that the architecture is implemented in a different platform than ours, which may affect its difference in power consumption and achieved throughput.

The work in [12] implements an architecture that focuses on parallelization based on the number of available actions. This work is also the base for both of our architectures. With the implementation of the approximated multiplier, [12] only requires a small amount of DSP compared to [13]. According to the presented data in [12], despite the exact configuration of 32-bit width, $N = 128$, and $Z = 4$, their design requires much smaller hardware resources compared to both

TABLE 9. Comparison with other works.

Reference	P. Gan. *et al.* [14]	Spanò, *et al.* [12]		Da Silva, *et al.* [13]	Y. Meng *et al.* [33]	Proposed V1 (Parallel)	**Proposed V2 (FARANE-Q)**	
Design Level	Standalone Core					System on Chip		
Platform	Xilinx Vertex-7	Xilinx ZCU106		Xilinx Virtex-6 ML60	Xilinx UltraScale+	Xilinx Ultra96-V2		
Technology (nm)	28	16		40	16	16		
Architecture	NNQL (Perceptron)	Approximate Computing		-	Pipeline Staging	Approximate Computing	**Approx. Computing, Parallel Staging**	
Parallelization	-	Z		N	Z	Z	**Z**	
Action Policy	-	NA		random	ϵ-greedy	ϵ-greedy	**decreasing-ϵ**	
Number of Agents (G)	single	single		single	**double**	single	single	
N×Z	24 × 9	128 × 4		132 × 4	64 × 8	128 × 4		
Clock Freq (MHz)	50	158	112	NA	189	125	150	
Bit Width	NA	16	32	30	16	32	32	16
DSP	17	3	5	370	4	0	1	1
LUT	12253	333	682	77574	172	2655	2490	2062
LUTRAM	NA	160	320	NA	NA	20	26	23
Registers	NA	258	606	13175	345	2076	2348	2179
BRAM	15	NA	NA	NA	NA	21	22	20
Throughput (MSps)	2.34	158	112	13.4	**189**	31.11	148.55	145.46
Power (mW)	1916	80	142	78	80	94	85	**60**
Area Eff. (MSps/1KLUT· Agent)	0.19	320.49	111.78	0.17	**549.42**	11.63	59.04	69.76
Energy Eff. (MSps/W · Agent)	1.22	1975	788.73	171.79	1180	330.96	1747.65	**2424.33**
Ratio (16-Bit FARANE-Q/other)	1985.06×	1.23×	3.08×	14.12×	2.06×	7.33×	1.39×	**1**×

of our architectures. Thus, their design can achieve an area efficiency of 111.78 MSps/1KLUT per agent, which is higher than ours. However, the energy efficiency of this architecture, which is 788.73 MSps/W per agent, is lower than V2 (FARANE-Q) due to its high dynamic power consumption of 142 mW, despite its low resource utilization. This also applies to the 16-bit variation of their design. The design achieves a higher clock frequency of 158 MHz. Due to the higher clock frequency and even the lower hardware resources than their 32-bit counterpart, their design achieves an area efficiency of 320.49 MSps/1KLUT per agent and an energy efficiency of 1975 MSps/W per agent.

Finally, we compare our proposed architectures with [33]. In [33], similar to V2 (FARANE-Q), pipeline architecture is also proposed but uses no approximate computing. It also uses the ϵ-greedy algorithm for determining the actions. However, in our proposed method V2 (FARANE-Q), we use dynamic ϵ-greedy with decreasing-ϵ. Another uniqueness of [33] is the use of multi-agent, in this case, double agents. Thus, the throughput is contributed by two agents that work in parallel. Compared to ours and other work presented here, [33] has the highest throughput of 189 MSps by using a double agent, and our V2 (FARANE-Q) has the second highest with a throughput of 148.55 MSps by only an agent. With its area efficiency of 549.42 MSps/1KLUT per agent, this architecture can achieve the highest area efficiency. This is possible due to its small LUT utilization and high throughput. We assume they did not include the overall SoC utilization, such as the processing system and memories with its connections. This is shown by the absence of BRAM and LUTRAM (Not-Available/NA) utilization in the paper. Thus, using Eq. (11) that considers the number of agents used, our 16-bit FARANE-Q has the highest energy efficiency, which is 2.06 × higher than [33].

Compared with other works in Table 9, we want to highlight that the data in our work shows more than just the resource utilization and the power consumption of the PL. It shows the AXI modules, memory blocks, and the PL combined resource utilization and power consumption. Tables 10 and 11 show the distribution of resource utilization and power consumption relative to the three modules. Based on the given distribution, the AXI modules use the most LUTs and Registers but they have a low dynamic power consumptions (19 mW for 16-bit and 20 mW for 32-bit). The resource utilization is then followed by the PL and the memory blocks. Meanwhile, the memory blocks have the highest dynamic power consumption (40 mW for 16-bit and 52 mW for 32-bit), while the PL has the lowest (8 mW for 16-bit and 15 mW for 32-bit).

The AXI modules interface the memories and the PL with the PS. Thus, these modules are active before the learning process begins when the PS initializes the hyperparameters and environment to the PL and memory blocks. These modules are also active after the learning process ends, specifically when the PS reads the final Q-Matrix from the memory blocks. The opposite happens for the PL, which is active only during the learning process. Meanwhile, the memory blocks are constantly being read and written before, during, and after the learning process. Due to these behaviors, it is justified that our V2 has lower power consumption despite its higher resource utilization compared to other works. The high resource utilization of the AXI modules does not translate to high power consumption because it is mostly idle or inactive compared to the memory blocks, which contribute the

TABLE 10. Resource and power distribution of V2 for 16-bit.

Block/Module	LUT	LUTRAM	FF	BRAM	Power (mW)
AXI modules	1447	0	1765	0	19
Memory blocks	33	13	57	20	40
PL	582	10	357	0	8

TABLE 11. Resource and power distribution of V2 for 32-bit.

Block/Module	LUT	LUTRAM	FF	BRAM	Power (mW)
AXI modules	1443	0	1765	0	20
Memory blocks	38	16	66	22	52
PL	994	10	517	0	15

highest dynamic power consumption despite its low resource utilization.

V. CONCLUSION

In this paper, we have proposed a scalable and configurable Reinforcement Learning system in SoC architecture. The designed system is capable of performing the RL algorithm in variable system parameters, such as the different sizes of state-action, a random layout of floorplan (state transition), and system configurations. Hence, it can be deployed for different applications of RL with various system configurations. Experimental results showed that our FARANE-Q achieves an acceleration ratio of up to around 1795× and 280× compared to embedded ARM on Zynq and Intel i7 processor, respectively. In addition, the employed fraction-bit optimization and approximate computing techniques provide a low complexity design, with lower LUTs and without DSPs implementation, and a low error rate of up to less than 0.4% for more than 8 fraction bits. Our parallel and pipeline architecture of FARANE-Q achieves a high throughput of 148.55 MSps with a higher clock frequency of 150 MHz. Finally, the proposed FARANE-Q outperforms others by at least 1.23× in energy efficiency.

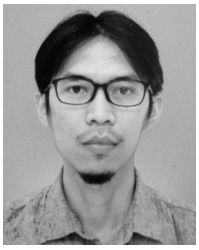
ACKNOWLEDGMENT

The authors acknowledge Handi Nugroho Setiawan for the early data for Version 1.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[2] T. Johannink, S. Bahl, A. Nair, J. Luo, A. Kumar, M. Loskyll, J. A. Ojea, E. Solowjow, and S. Levine, "Residual reinforcement learning for robot control," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, May 2019, pp. 6023–6029.
[3] B. Zuo, J. Chen, L. Wang, and Y. Wang, "A reinforcement learning based robotic navigation system," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2014, pp. 3452–3457.
[4] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2017, pp. 3389–3396.
[5] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2018, pp. 1–8.
[6] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3133–3174, 4th Quart., 2019.
[7] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2018, pp. 1871–1879.
[8] C. Yu, J. Liu, S. Nemati, and G. Yin, "Reinforcement learning in healthcare: A survey," *ACM Comput. Surv.*, vol. 55, no. 1, pp. 1–36, Jan. 2023.
[9] A. Coronato, M. Naeem, G. De Pietro, and G. Paragliola, "Reinforcement learning for intelligent healthcare applications: A survey," *Artif. Intell. Med.*, vol. 109, Sep. 2020, Art. no. 101964.
[10] T. P. Le, C. Rho, Y. Min, S. Lee, and D. Choi, "A2GAN: A deep reinforcement-based learning algorithm for risk-aware in finance," *IEEE Access*, vol. 9, pp. 137165–137175, 2021.
[11] G. Hinton and T. J. Sejnowski, *Unsupervised Learning: Foundations of Neural Computation*. Cambridge, MA, USA: MIT Press, 1999.
[12] S. Spanò, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, "An efficient hardware implementation of reinforcement learning: The *Q*-learning algorithm," *IEEE Access*, vol. 7, pp. 186340–186351, 2019.
[13] 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, 2018.
[14] P. R. Gankidi and J. Thangavelautham, "FPGA architecture for deep learning and its application to planetary robotics," in *Proc. IEEE Aerosp. Conf.*, Mar. 2017, pp. 1–9.
[15] S. S. Sahoo, A. R. Baranwal, S. Ullah, and A. Kumar, "MemOReL: A memory-oriented optimization approach to reinforcement learning on FPGA-based embedded systems," in *Proc. Great Lakes Symp. VLSI*, 2021, pp. 339–346.
[16] A. R. Baranwal, S. Ullah, S. S. Sahoo, and A. Kumar, "ReLAccS: A multi-level approach to accelerator design for reinforcement learning on FPGA-based systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 9, pp. 1754–1767, Sep. 2021.
[17] Avnet. (2021). *Ultra96-v2 Board*. Accessed: Nov. 3, 2021. [Online]. Available: https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/
[18] J. Lee, S. Kim, S. Kim, W. Jo, D. Han, J. Lee, and H.-J. Yoo, "OmniDRL: A 29.3 TFLOPS/W deep reinforcement learning processor with dualmode weight compression and on-chip sparse weight transposer," in *Proc. Symp. VLSI Circuits*, Jun. 2021, pp. 1–2.
[19] R. Berdan, T. Marukame, S. Kabuyanagi, K. Ota, M. Saitoh, S. Fujii, J. Deguchi, and Y. Nishi, "In-memory reinforcement learning with moderately-stochastic conductance switching of ferroelectric tunnel junctions," in *Proc. Symp. VLSI Technol.*, Jun. 2019, pp. T22–T23.
[20] A. Amravati, S. B. Nasir, S. Thangadurai, I. Yoon, and A. Raychowdhury, "A 55 nm time-domain mixed-signal neuromorphic accelerator with stochastic synapses and embedded reinforcement learning for autonomous micro-robots," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 124–126.
[21] S.-C. Kao, G. Jeong, and T. Krishna, "ConfuciuX: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 622–636.
[22] F. Tu, W. Wu, Y. Wang, H. Chen, F. Xiong, M. Shi, N. Li, J. Deng, T. Chen, L. Liu, S. Wei, Y. Xie, and S. Yin, "Evolver: A deep learning processor with on-device quantization–voltage–frequency tuning," *IEEE J. Solid-State Circuits*, vol. 56, no. 2, pp. 658–673, Feb. 2021.
[23] J. Yang, S. Hong, and J.-Y. Kim, "FIXAR: A fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 259–264.
[24] Y. Wang, M. Wang, B. Li, H. Li, and X. Li, "A many-core accelerator design for on-chip deep reinforcement learning," in *Proc. 39th Int. Conf. Computer-Aided Design*, Nov. 2020, pp. 1–7.
[25] Y. G. Kim and C.-J. Wu, "AutoScale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 1082–1096.
[26] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 499–513.

[27] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proc. ACM/IEEE 46th Int. Symp. Comput. Archit.*, Jun. 2019, pp. 279–291.
[28] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.
[29] T.-R. Lin, D. Penney, M. Pedram, and L. Chen, "A deep reinforcement learning framework for architectural exploration: A routerless NoC case study," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 99–110.
[30] Y. Fenjiro and H. Benbrahim, "Deep reinforcement learning overview of the state of the art," *J. Autom., Mobile Robot. Intell. Syst.*, vol. 12, no. 3, pp. 20–39, Dec. 2018.
[31] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, Learn., Optim.*, vol. 12, no. 3, p. 729, 2012.
[32] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Machine Learning Proceedings 1995*. Amsterdam, The Netherlands: Elsevier, 1995, pp. 30–37.
[33] 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," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 107–114.
[34] 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, 2019.
[35] L. Shoufeng, L. Ximin, and D. Shiqiang, "*Q*-learning for adaptive traffic signal control based on delay minimization strategy," in *Proc. IEEE Int. Conf. Netw., Sens. Control*, Apr. 2008, pp. 687–691.
[36] M. Abdoos, N. Mozayani, and A. L. C. Bazzan, "Traffic light control in non-stationary environments based on multi agent *Q*-learning," in *Proc. 14th Int. IEEE Conf. Intell. Transp. Syst. (ITSC)*, Oct. 2011, pp. 1580–1585.
[37] H. Wei, G. Zheng, H. Yao, and Z. Li, "IntelliLight: A reinforcement learning approach for intelligent traffic light control," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 2496–2505.
[38] Xilinx. (2017). *Ultra96-V2 Board*. Accessed: Jun. 7, 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide
[39] G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, M. Re, and S. Spanò, "An action-selection policy generator for reinforcement learning hardware accelerators," in *Proc. Int. Conf. Appl. Electron. Pervading Ind., Environ. Soc.* New York, NY, USA: Springer, 2020, pp. 267–272.
[40] A. K. Panda, P. Rajput, and B. Shukla, "FPGA implementation of 8, 16 and 32 bit LFSR with maximum length feedback polynomial using VHDL," in *Proc. Int. Conf. Commun. Syst. Netw. Technol.*, May 2012, pp. 769–773.
[41] M. A. Wiering, "Explorations in efficient reinforcement learning," Ph.D. dissertation, Fac. Sci. (FNWI), Inform. Inst. (IVI), Univ. Amsterdam, Amsterdam, The Netherlands, 1999.
[42] M. Tokic, "Adaptive ε-greedy exploration in reinforcement learning based on value differences," in *Proc. Annu. Conf. Artif. Intell.* New York, NY, USA: Springer, 2010, pp. 203–210.
[43] Xilinx. (2021). *PYNQ: Python Productivity*. Accessed: Jun. 8, 2022. [Online]. Available: http://www.pynq.io/
[44] AMD. (2018). *Getting Started With PYNQ*. Accessed: Jul. 22, 2022. [Online]. Available: https://pynq.readthedocs.io/en/latest/getting_started.html
[45] M. Rothmann and M. Porrmann, "A survey of domain-specific architectures for reinforcement learning," *IEEE Access*, vol. 10, pp. 13753–13767, 2022.



NANA SUTISNA (Member, IEEE) received the B.S. degree in electrical engineering and the M.S. degree in microelectronics from the Bandung Institute of Technology, Indonesia, in 2005 and 2011, respectively, and the Ph.D. degree in computer science and electronics from the Kyushu Institute of Technology, in 2017. From 2017 to 2020, he was a Postdoctoral Fellow at the Department of Computer Science and System Engineering, Kyushu Institute of Technology. He is currently a Lecturer with the Institut Teknologi Bandung. His research interests include VLSI design, baseband wireless system design, AI processor design, and HW/SW co-design and co-verification.

ANDI M. RIYADHUS ILMY received the B.Sc. degree in electrical engineering from the Institut Teknologi Bandung (ITB), Indonesia, in 2022. He is currently working as a Researcher with the Microelectronic Center, ITB. His research interests include computer architecture, artificial intelligence, VLSI design, and embedded systems.

INFALL SYAFALNI (Member, IEEE) received the B.Eng. degree in electrical engineering from the Institut Teknologi Bandung (ITB), Bandung, Indonesia, in 2008, the M.Sc. degree in electronic engineering from the University of Science Malaysia (USM), Penang, Malaysia, in 2011, and the Dr.Eng. degree in engineering from the Kyushu Institute of Technology (KIT), Iizuka, Fukuoka, Japan, in 2014. From 2014 to 2015, he held a research position at KIT. From 2015 to 2018, he held an ASIC engineer position at the ASIC Development Group, Logic Research Company Ltd., Fukuoka. In 2019, he joined the ITB, where he is currently an Assistant Professor with the School of Electrical Engineering and Informatics and a Researcher with the University Center of Excellence on Microelectronics, ITB. His current research interests include logic synthesis, logic design, VLSI design, efficient circuits, and algorithms.

RAHMAT MULYAWAN (Member, IEEE) received the B.Eng. degree in EE from the ITB, Indonesia, in 2008, and the M.Sc. degree in EE from the TU Delft, The Netherlands, in 2011. He is currently a member of the Microelectronics Centre, ITB. His research interests include intelligent signal processing, MIMO systems, and transceiver design for optical wireless communications.

TRIO ADIONO (Senior Member, IEEE) received the B.Eng. degree in electrical engineering and the M.Eng. degree in microelectronics from the Institut Teknologi Bandung, Indonesia, in 1994 and 1996, respectively, and the Ph.D. degree in VLSI design from the Tokyo Institute of Technology, Japan, in 2002. He is currently a Professor with the School of Electrical Engineering and Informatics, and also works as the Head of the IC Design Laboratory, Microelectronics Center, Institut Teknologi Bandung. He holds a Japanese patent on a high-quality video compression system. His research interests include VLSI design, signal and image processing, VLC, smart cards, and electronics solution design and integration.