Challenge #12

1. Design workflow

**1. Specification & Mathematical Modeling**

* **Target formula**:

Q(s,a)←(1−α)⋅Q(s,a)+α⋅(r+γ⋅max⁡Q(s′,a′))Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot \left( r + \gamma \cdot \max Q(s', a') \right)

* **Fixed-point format**: Choose Q4.12 or Q8.8 for hardware-friendliness.
* **Inputs**:
  + Q(s,a), max Q(s',a'), reward, alpha, gamma
* **Output**:
  + Updated Q(s,a)

**2. RTL Design (SystemVerilog)**

* Use:
  + 2 multipliers: γ \* maxQ, α \* target
  + 2 adders: r + (γ \* maxQ), final Q update
  + 1 subtractor: 1 - α (can be precomputed)

**3. Functional Simulation**

* Write a testbench to:
  + Apply sample values (e.g., Q=0, reward=-1, maxQ=0)
  + Compare output with Python/NumPy expected result
  + Simulate with tools like **EDA Playground**, **Vivado**, or **ModelSim**

**4. Timing Simulation & Debugging**

* Add a clock if using registers (pipelining)
* Include control signals (start, done) for integration
* Use waveform viewers (e.g., GTKWave or EPWave) to verify data flow

**5. Synthesis**

* Synthesize with tools like:
  + **Vivado**, **Synplify**, or **Yosys**
* Check:
  + Area (number of multipliers/adders)
  + Timing (max frequency)
  + Power (optional)

**6. Hardware Integration**

* Wrap inside a top-level FSM or accelerator
* Interface with:
  + State memory
  + Q-table memory
  + Learning rate registers

**7. Co-Simulation / Software-Hardware Validation**

* Match results with Python Q-learning output
* Sweep inputs (α, γ, rewards, Q-values) to test edge cases
* Optionally use Python-Cocotb or Verilator for co-simulation

**8. Optional: FPGA Prototyping**

* Deploy on board (e.g., Nexys A7 or ZedBoard)
* Use UART/JTAG to inject test values or stream episodes

1. Details of all PyMTL modules used.  
     
   QUpdateUnit:

This QUpdateUnit is a PyMTL3 hardware module that implements the core Q-learning update formula used in reinforcement learning. It calculates the updated Q-value Q(s,a) based on the current Q-value, reward, discount factor (gamma), and learning rate (alpha) using fixed-point arithmetic. The logic follows the standard Q-learning update rule:



The inputs include the current Q-value (q\_sa), maximum Q-value from the next state (max\_q\_sp), and the immediate reward. It computes an intermediate discounted\_q (γ·maxQ), then the target value, and finally blends the old and new Q-values to produce q\_sa\_new, which is output as the updated Q-value. The use of shift operations (e.g., >> 4) reflects scaling for fixed-point representation (Q4.12).

Test\_QUpdateUnit:

This PyMTL3 testbench function, test\_q\_update\_unit, verifies the functionality of the QUpdateUnit hardware module, which computes the Q-learning update formula used in reinforcement learning. It uses the run\_test\_vector\_sim utility to apply a sequence of test vectors to the module. Each test vector consists of input values: q\_sa (current Q-value), max\_q\_sp (maximum Q-value for next state), reward, alpha (learning rate), and gamma (discount factor), followed by the expected output q\_sa\_new. For example, in the first test, the inputs represent a case where the updated Q-value should be -0.5 (encoded as 0xF800 in fixed-point Q4.12 format). The test checks whether the module computes this value correctly, thereby confirming the correctness of the Q-update arithmetic logic in hardware.

QFSM:

This QFSM PyMTL module models a simplified finite state machine (FSM) that performs a single Q-value update step, as used in Q-learning reinforcement learning algorithms. It integrates a QUpdateUnit datapath, which computes the updated Q-value using the core formula.

The module maintains a 3D Q-table (Q\_table) indexed by state (s\_i, s\_j) and action a. The FSM cycles through three states: initialization (assigning a fixed state and action), computation (waiting for the Q-value update), and write-back (storing the updated Q-value back into the Q-table). This component emulates a single-cycle hardware-friendly update for one state-action pair and serves as a toy hardware design example of a learning step in FrozenLake-style environments.

QFSM\_test:

This Python script serves as a **testbench for a PyMTL hardware module** named QFSM, which simulates a small finite state machine (FSM) for performing a Q-value update in a reinforcement learning setting, such as FrozenLake. It imports PyMTL3 and uses the run\_sim infrastructure to instantiate the design, apply standard passes (like elaboration and simulation), and reset the hardware model. The FSM is then simulated over 6 clock cycles using sim\_tick(), and its internal state or output is printed each cycle using line\_trace(). This test checks that the FSM correctly performs the Q-learning update and transitions through the expected states.

Final Output:

Q\_new=0fff

Q\_new=0fff

Q\_new=0fff

Q\_new=17fe

Q\_new=1bfe

Q\_new=1bfe

Q-learning Benchmark: Python vs PyMTL

| **Metric** | **Python (Numba JIT)** | **PyMTL FSM (simulated)** |
| --- | --- | --- |
| Episodes/Steps | 10,000 | 10,000 cycles |
| Execution Time | **3.87 sec** | **~1.00 sec** (simulated) |
| Modeling Type | High-level Python | Cycle-accurate RTL |