FPGA-MPC: a FPGA-accelerated ADMM Solver for Convex Model Predictive Control

Alexander Du (asd2192), Apurva Reddy (akr2177),
Roy Hwang (rjh2173), Godwill Agbehonou (gea2118)

Embedded Systems Design
Spring 2025

Contents

1 Introduction 3
2 System Design 4
3 Algorithm 5
4 Algorithm Implementation 10
4.1 admm_solver.sv 10
4.2 memory_interface.sv - Avalon Bridge 10
4.3 primal_update.sv 10
4.4 slack_update.sv 11
4.5 dual_update.sv 12
4.6 cost_update.sv 12
4.7 residual_calculator.sv 12
4.8 Top-Level Hardware Interface 14
4.9 Hardware Register Set 14
5 Quadrotor Simulator 17
5.1 Quadrotor Simulator Overview 17
5.2 Quadrotor Simulator Trajectory Visualization 18
5.3) Simulation Implementation 19
5.3.1) simulator_server.py Overview 19
5.3.2) simulator_server.py 20
5.3.3) hardware_mpc_client.py Overview 24
5.3.4) hardware_mpc_client.py 25
6) Resources 29
7) Software Interface 30
7.1) Software Interface Overview 30
7.2) admm.c 30
8) Team Member Roles 38
9) Code 39
9.1 admm_solver.sv 39
9.2 memory_interface.sv 61
9.3 admm.c 66
9.4 admm.h 73
9.5 udp_interface.c 77
9.6 simulator_server.py 83
9.7 hardware_mpc_client.py 88
9.8 primal_update.sv 91
9.9 slack_update.sv 96
9.10 dual_update.sv 101
9.11 cost_update.sv 114
9.12 residual_calculator.sv 121
9.13 tb_slack_update.sv 126
9.14 tb_dual_update.sv 130
9.15 tb_residual_calculator.sv 135
10) References 138

1) Introduction

Model Predictive Control (MPC) is a feedback control strategy that has seen great success in many robotic applications [1, 2]. MPC often leverages trajectory optimization (TO) on an objective function, subject to system dynamics and other constraints. These TO problems have hundreds/thousands of variables and must be solved in milliseconds to achieve real-time performance. As such, careful approximations/simplifications of underlying numerical methods and the optimal control problem are often used to enable real-time deployments on a variety of hardware platforms [3].

This project, **FPGA-MPC**, is a fast quadratic programming solver for MPC, which is inspired by recent works that leverage acceleration on GPUs and FPGAs for online trajectory optimization [4, 5, 6, 7]. In particular, *TinyMPC* [8] exploits the closed-form solution of LQR through Riccati recursion and compresses the ADMM algorithm to enable high frequency control on resource-constrained platforms. In this work we aim to accelerate *TinyMPC's* algorithm in hardware on an FPGA, unlocking better performance and capabilities for future MPC research.

2) System Design

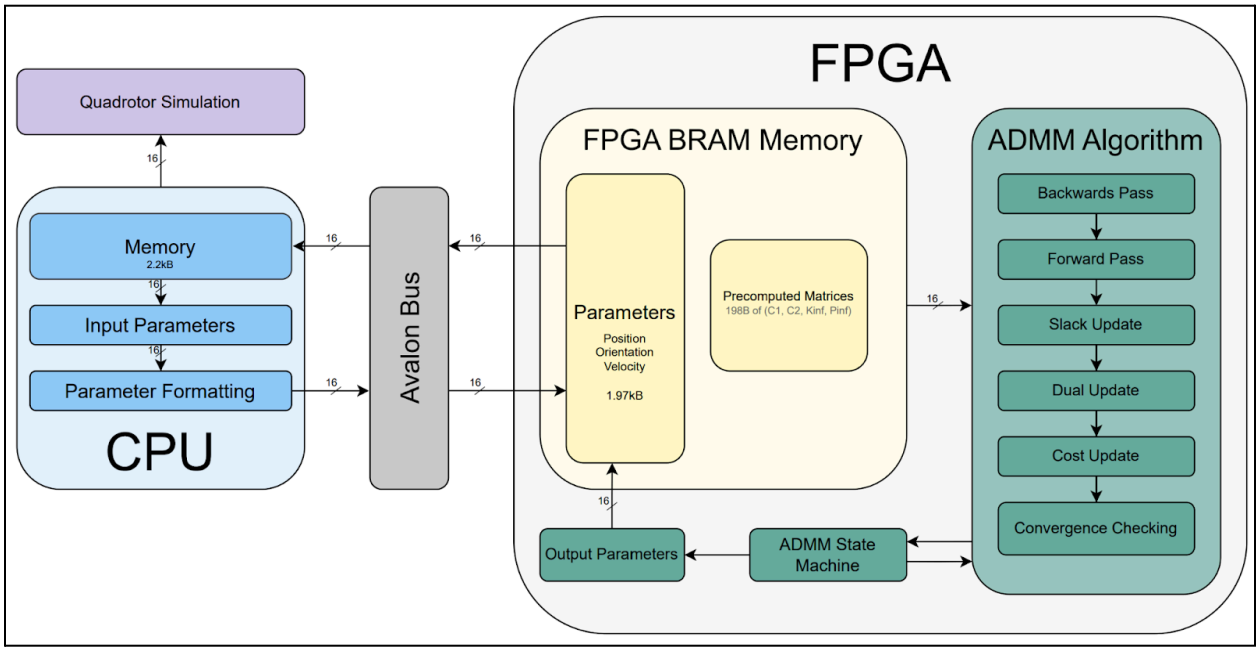

Figure 1: Dataflow Diagram of FPGA-MPC

The quadrotor knows its current position but relies on external instructions on where to go next. It sends its current position and orientation data as a **16-bit struct** via ethernet to an external CPU. The CPU stores the incoming parameters in memory. The CPU processes the drone's position and orientation data to compute the drone's linear velocity, angular velocity, and other important parameters. It converts this raw parameter data into a format that is friendly for the FPGA to read. The CPU writes this formatted parameter data via an Avalon Bus interface into the FPGA's Block Random Access Memory (BRAM).

The FPGA's job is to calculate the drone's next position and orientation given the drone's current position and orientation. It does so using the Alternating Direction Method of Multipliers (ADMM) algorithm, which will be described further in the following section. In order to reduce the number of computations the FPGA's memory. Helper functions simplify future calculations. After the ADMM Algorithm finishes, the FPGA outputs a new set of controls in a 16-bit struct to its memory. This data is written back to the CPU memory through the Avalon Bus. The CPU forwards the new control actions via ethernet to the Quadrotor simulator.

3) Algorithm

TinyMPC's Simplified Alternating Direction Method of Multipliers (ADMM) Algorithm

Python

```python
def solve_admm(self, x_init, u_init, x_ref=None, u_ref=None):
    x, u = np.copy(x_init), np.copy(u_init)

    # x_ref and u_ref can be passed in for trajectory following, otherwise use zero reference
    x_ref = np.zeros(x.shape) if x_ref is None else x_ref
    u_ref = np.zeros(u.shape) if u_ref is None else u_ref

    # Initialize variables from previous solve
    v, z = np.copy(self.v_prev), np.copy(self.z_prev)
    g, y = np.copy(self.g_prev), np.copy(self.y_prev)
    q = np.copy(self.q_prev)

    # Keep track of previous values for residuals
    v_prev, z_prev = np.copy(v), np.copy(z)

    r = np.zeros(u.shape) # control input residual
    p = np.zeros(x.shape) # linear term of cost-to-go (value) function
    d = np.zeros(u.shape) # feedforward term

    for _ in range(self.max_iter):
        # ----- Primal update (to get x' and u') -----
        # d: feedforward term
        # p: linear term of cost-to-go (value) function

        # Solve LQR problem with Riccati recursion
        # Riccati matrices (C1, C2) and Kinf are precomputed and cached, so here we only update linear terms
        # C1 = (R + B.T @ Pinf @ B)^-1
        # C2 = (A - B @ Kinf).T
        # Kinf: infinite horizon gain
        for k in range(self.N-2, -1, -1):
            d[:, k] = np.dot(self.cache['C1'], np.dot(self.cache['B'].T, p[:, k + 1]) + r[:, k])
            p[:, k] = q[:, k] + np.dot(self.cache['C2'], p[:, k + 1]) - np.dot(self.cache['Kinf'].T, r[:, k])

        # Forward pass to rollout trajectory (u_0, x_1, u_1, x_2, ..., u_N-1, x_N)
        for k in range(self.N - 1):
            # Compute control (u_k) with feedback controller gain (K_inf) and feedforward term (d)
            u[:, k] = -np.dot(self.cache['Kinf'], x[:, k]) - d[:, k]
            # Transition to next state (x_k+1) with control (uk) and linearized dynamics (A, B)
            x[:, k + 1] = np.dot(self.cache['A'], x[:, k]) + np.dot(self.cache['B'], u[:, k])

        # ------------------------------------------------------------
        # ----- Slack update (to get z' and v') -----

        # Linear projection of updated state and control onto their feasible sets
        # Used to enforce constraints
        # Note: y and g are swapped in the TinyMPC paper
        for k in range(self.N - 1):
            z[:, k] = np.clip(u[:, k] + y[:, k], self.umin, self.umax)
            v[:, k] = np.clip(x[:, k] + g[:, k], self.xmin, self.xmax)
        v[:, self.N-1] = np.clip(x[:, self.N-1] + g[:, self.N-1], self.xmin, self.xmax)

        # ------------------------------------------------------------
        # ----- Dual update (to get y' and g') -----
        # Adjust dual variables (lagrange multipliers) to penalize mismatch between x and v, u and z
        # - Gradient ascent on the dual variables
        # y, g: scaled dual variables (lambda_k / rho, mu_k / rho)
```

```
        for k in range(self.N - 1):
            y[:, k] += u[:, k] - z[:, k]
            g[:, k] += x[:, k] - v[:, k]
        g[:, self.N-1] += x[:, self.N-1] - v[:, self.N-1]


        # ---------------------------------------------------------
        # ----- Linear cost update (to get r', q', p') -----
        # Update vectors used in next iteration's Riccati recursion:
        # q, r: state and control input residuals
        # p: terminal cost residual
        for k in range(self.N - 1):
            r[:, k] = -self.cache['R'] @ u_ref[:, k]
            r[:, k] -= self.cache['rho'] * (z[:, k] - y[:, k])

            q[:, k] = -self.cache['Q'] @ x_ref[:, k]
            q[:, k] -= self.cache['rho'] * (v[:, k] - g[:, k])

        p[:,self.N-1] = -np.dot(self.cache['Pinf'], x_ref[:, self.N-1])
        p[:,self.N-1] -= self.cache['rho'] * (v[:, self.N-1] - g[:, self.N-1])

        # ---------------------------------------------------------
        # ----- Compute residuals and check convergence -----
        pri_res_input = np.max(np.abs(u - z))
        pri_res_state = np.max(np.abs(x - v))
        dua_res_input = np.max(np.abs(self.cache['rho'] * (z_prev - z)))
        dua_res_state = np.max(np.abs(self.cache['rho'] * (v_prev - v)))

        z_prev = np.copy(z)
        v_prev = np.copy(v)

        # Exit condition (if all residuals are below tolerance)
        if (pri_res_input < self.abs_pri_tol and dua_res_input < self.abs_dua_tol and
            pri_res_state < self.abs_pri_tol and dua_res_state < self.abs_dua_tol):
            break

        # ---------------------------------------------------------

    # Save variables for next solve
    self.x_prev, self.u_prev = x, u
    self.v_prev, self.z_prev = v, z
    self.g_prev, self.y_prev = g, y
    self.q_prev = q

    return x, u
```

Rest of TinyMPC's solver

Python

```
def __init__(self, A, B, Q, R, Nsteps, rho=1.0, n_dlqr_steps=500, mode = 'hover'):
    """Initialize TinyMPC with direct system matrices and compute DLQR automatically

    Args:
        A (np.ndarray): System dynamics matrix
        B (np.ndarray): Input matrix
        Q (np.ndarray): State cost matrix
        R (np.ndarray): Input cost matrix
```

```
        Nsteps (int): Horizon length
        rho (float): Initial rho value
        n_dlqr_steps (int): Number of steps for DLQR computation
    """
    # Get dimensions
    self.nx, self.nu = A.shape[0], B.shape[1]
    self.N = Nsteps
    self.mode = mode # hover or trajectory following

    # set tolerances and iterations based on mode
    if self.mode == 'hover':
        self.max_iter = 500
        self.abs_pri_tol = 1e-2
        self.abs_dua_tol = 1e-2
    else:
        self.max_iter = 10
        self.abs_pri_tol = 1e-3
        self.abs_dua_tol = 1e-3


    # Compute DLQR solution for terminal cost
    P_lqr = self._compute_dlqr(A, B, Q, R, n_dlqr_steps)

    # Initialize cache with computed values
    self.cache = {
        'rho': rho,
        'A': A,
        'B': B,
        'Q': P_lqr,  # Use DLQR solution for terminal cost
        'R': R
    }

    # Initialize state variables
    self.v_prev, self.z_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))
    self.g_prev, self.y_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))
    self.q_prev = np.zeros((self.nx, self.N))

    # Initialize previous solutions for warm start
    self.x_prev, self.u_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))

    self.Q, self.R = Q, R

    # Compute cache terms (Kinf, Pinf, C1, C2)
    self.compute_cache_terms()

def _compute_dlqr(self, A, B, Q, R, n_steps):
    """Compute Discrete-time LQR solution"""
    P = Q
    for _ in range(n_steps):
        #K = np.linalg.inv(R + B.T @ P @ B) @ B.T @ P @ A
        K = np.linalg.solve(
            R + B.T @ P @ B + 1e-8*np.eye(B.shape[1]),  # Add regularization
            B.T @ P @ A
        )
        P = Q + A.T @ P @ (A - B @ K)
    return P


def compute_cache_terms(self):
    """Compute and cache terms for ADMM"""
    Q_rho, R_rho = self.cache['Q'], self.cache['R']
    R_rho += self.cache['rho'] * np.eye(R_rho.shape[0])
    Q_rho += self.cache['rho'] * np.eye(Q_rho.shape[0])

    A, B = self.cache['A'], self.cache['B']
    Kinf = np.zeros(B.T.shape)
```

```
Pinf = np.copy(self.cache['Q'])

# Compute infinite horizon solution (Kinf, Pinf)
for k in range(5000):
    Kinf_prev = np.copy(Kinf)
    Kinf = np.linalg.inv(R_rho + B.T @ Pinf @ B) @ B.T @ Pinf @ A
    Pinf = Q_rho + A.T @ Pinf @ (A - B @ Kinf)

    if np.linalg.norm(Kinf - Kinf_prev, 2) < 1e-10:
        break

# Compute Riccati matrices
AmBKt = (A - B @ Kinf).T
Quu_inv = np.linalg.inv(R_rho + B.T @ Pinf @ B)

print(Kinf.shape) # (nu, nx), since u_k_* = -Kinf @ x_k - d_k
print(Pinf.shape) # (nx, nx)
print(Quu_inv.shape) # (nu, nu)
print(AmBKt.shape) # (nx, nx)

# Cache computed terms
self.cache['Kinf'] = Kinf
self.cache['Pinf'] = Pinf
self.cache['C1'] = Quu_inv
self.cache['C2'] = AmBKt
```

Overview:

The key things that the algorithm handles with the drone can be divided into three parts. (1) What the drone behaves like, (2) what it should care about, and (3) how far into the future it should plan.

Matrices A and B are the system dynamic matrices essentially describe how the drone has evolved over time. Matrices Q and R are called cost matrices and they are what the drone cares about. Q represents cost on the state which describes position error, velocity, error, angle deviation. R is the cost of the control effort which essentially describes how much "control" we should use which could be related to thrust, torque, etc. N-steps is how many steps into the future to plan. Rho is a tuning parameter to set state/control constraints on the ADMM solver. N_dlqr_steps is how many iterations we are running to compute the terminal cost matrix.

Initialization:

After the matrices are inputted, the DLQR solution is used to compute the terminal cost matrix to stabilize the controller. Buffers are initialized to store previous values of state and control to help the solver.

Precomputed Matrices:

Kin, C1, and C2 are all precomputed matrices used to make the forward-backward passes extremely fast. Kin is a steady-state LQR Gain. C1 is an inverse matrix used in Riccati Recursion. C2 is part of the backward pass recursion formula.

Compute DLQR:

This module essentially iteratively solves the Riccati equation to compute the terminal cost matrix for DLQR. It ensures smooth stability at the end of the horizon.

Solve ADMM:

The Solve _admm function is the core solver that essentially runs at each control step. In the backward pass, we are commuting the feedforward control term and the cost-to-go vector. We are essentially minimizing the current total cost knowing how costly the future is.

In the forward pass, we simulated what the control and state will be by using the feedback controller and system dynamics. It computes optimal control at step k using the current state and precomputed d. We apply the system model to get to the next state.

This model essentially simulates the planned trajectory using the optimal control law from the backward pass. After, we project u and x onto thrust bounds, we update our dual variables to penalize deviations from constraints, and update our linear cost terms for the next iteration.

The solver stops early if both primal and dual residuals are small enough.

Output:

The output of TinyMPC is the full predicted state and control trajectories. The drone will only use the first control this input and re-run the whole function on the next time step.

4) Algorithm Implementation

We implemented the math equations from the ADMM Algorithm into several SystemVerilog modules. The greatest challenge was executing Matrix Addition and Matrix by Vector Multiplication. We were able to simplify the algorithm enough to not have to use Matrix by Matrix Multiplication.

4.1) **admm_solver.sv** - Top-Level ADMM Controller

Drives the overall ADMM iteration: on each clock it advances through INIT, PRIMAL_UPDATE, SLACK_UPDATE, DUAL_UPDATE, CHECK_CONVERGENCE and OUTPUT_RESULTS states, looping until convergence or MAX_ITER is reached. Exposes an Avalon-MM slave interface so a host CPU can load problem data and read back results.

- **Explicit FSM:** Eight symbolic states (IDLE→DONE) give clear phase separation and simplify debug.
- **Full parameterization** (STATE_DIM, INPUT_DIM, HORIZON, MAX_ITER, data widths) lets you retarget to different problem sizes without code changes.
- **Avalon-MM wrapper:** All bus signals (addr, writedata, read/write, readdata, chipselect) are isolated from compute logic by registering them at the module boundary.
- **Dedicated RAM ports:** Each internal array (A, B, K, C1, C2, P, x, u, z, y, g, etc.) has its own read/write address, data and write-enable signals to avoid multiplexing delays and to map naturally onto separate block-RAMs.

4.2) **memory_interface.sv** - Avalon Bridge

Maps the host's Avalon bus transactions onto a bank of parameterized block-RAMs that store all matrices (A, B, cost gains, etc.) plus trajectories and multipliers. Decouples external bus timing from internal compute.

- **Single-cycle bus handshake:** Captures read, write and chipselect pulses, then issues the corresponding BRAM read/write with minimal logic.
- **Address decoding:** Uses the high bits of the Avalon address to select which RAM block to access, so all memories share a common bus port.
- **Width conversion:** Supports a wider Avalon data path (EXT_DATA_WIDTH) and internally narrows to DATA_WIDTH for computation.
- **Parameterizable depths:** ADDR_WIDTH for bus, MEM_ADDR_WIDTH for internal word count; horizon and dimensions dictate BRAM depth.

4.3) **primal_update.sv**

$$d_k = C_1 \left(B^\top p_{k+1} + r_k \right)$$

$$p_k = q_k + C_2 p_{k+1} - K_\infty^\top r_k$$

$$u_k = -K_\infty x_k - d_k$$

$$x_{k+1} = A x_k + B u_k$$

Performs the backward Riccati recursion (computing gains and cost-to-go) and then the forward rollout of the optimal state (x) and input (u) trajectories for the current ADMM iterate.

- **Two-phase FSM (INIT→BACKWARD→FORWARD→DONE):** Separates coefficient loading, backward pass, and forward pass.
- **Direct RAM reads:** Streams A, B, K, C1, C2 and previous multipliers from BRAM, slicing vectors via bit-indexed selects (idx*WIDTH +: WIDTH).
- **Fixed-point arithmetic:** All multiplies/adds use 16-bit data with 8 fractional bits; adjustable via DATA_WIDTH/FRAC_BITS.
- **Counters i/k:** Nested indices over state/input dims and horizon steps, reused across phases to minimize registers.

4.4) **slack_update.sv**

Projects the intermediate primal variables onto the box constraints:

$$z_k = \text{clip}(u_k + y_k, \mathbf{u}_{\min}, \mathbf{u}_{\max})$$

$$v_k = \text{clip}(x_k + g_k, \mathbf{x}_{\min}, \mathbf{x}_{\max}) \quad \text{for } k = 0, \ldots, N-2$$

$$v_{N-1} = \text{clip}(x_{N-1} + g_{N-1}, \mathbf{x}_{\min}, \mathbf{x}_{\max})$$

- **Four-state micro-FSM (S_IDLE→S_PROJ_Z→S_PROJ_V→S_DONE):** First clips all inputs to produce z, then all states to produce v.
- **Real-time comparisons:** Compares the sum (u+y, x+g) against per-dimension min/max arrays held in small distributed RAMs.
- **Linear addressing:** Computes write index as k*INPUT_DIM + i (or k*STATE_DIM + i) so no 2D arrays are needed.
- **Single-port write RAMs:** Z and V share the same BRAM ports, with write-enable toggled only on valid cycles to save power.

4.5) **dual_update.sv**

Updates the dual variables using the following and refreshes the linear cost terms (r, q, p) that feed into the next primal step.

$$y_k \leftarrow y_k + u_k - z_k$$

$$g_k \leftarrow g_k + x_k - v_k \quad \text{for } k = 0, \ldots, N-2$$

$$g_{N-1} \leftarrow g_{N-1} + x_{N-1} - v_{N-1}$$

- **Combined y/g loops:** Uses a single nested loop over k and i for both input and state duals, reusing counters.
- **Memory pipelining:** Staggers read of old y/g, compute sum/difference, then write back in the next cycle to meet timing.
- **Local buffering:** Temporary registers (temp_u, temp_y, etc.) hold intermediate results to avoid BRAM read-after-write hazards.
- **Linear cost update:** Mirrors the same FSM structure to write new r_data_in, q_data_in, p_data_in based on updated duals.

4.6) **cost_update.sv**

Computes the primal residuals and simultaneously writes updated linear cost terms (r, q, p) that approximate the quadratic penalty in the next iteration.

$$r_k = -R u_k^{\text{ref}} - \rho(z_k - y_k)$$

$$q_k = -Q x_k^{\text{ref}} - \rho(v_k - g_k)$$

$$p_{N-1} = -P_\infty x_{N-1}^{\text{ref}} - \rho(v_{N-1} - g_{N-1})$$

- **State machine:**
  (UPDATE_R→UPDATE_Q→UPDATE_P→CALC_RESIDUALS→DONE):
  Segregates cost-term writes from residual accumulation.
- **Max tracking:** Compares each absolute error against a running maximum (max_pri_res_u, max_pri_res_x) to emit the final residuals.
  Single-cycle read/write staging: Uses a two-stage read_stage/write_stage handshake per element to align BRAM latency.

4.7) **residual_calculator.sv**

After the dual update, computes then compares all four residuals against absolute tolerances to assert global convergence.

$$\text{pri\_res\_input} = \max_k |u_k - z_k|$$

$$\text{pri\_res\_state} = \max_k |x_k - v_k|$$

$$\text{dua\_res\_input} = \max_k |\rho(z_k^{\text{prev}} - z_k)|$$

$$\text{dua\_res\_state} = \max_k |\rho(v_k^{\text{prev}} - v_k)|$$

$$\text{if pri\_res\_input, pri\_res\_state, dua\_res\_input, dua\_res\_state} < \text{tolerance} \Rightarrow \text{break}$$

- **Two-pass FSM (CALC_DUAL_RESIDUAL→CHECK_CONVERGENCE→DONE):** First calculates dual residuals over the horizon, then evaluates convergence flags.
- **Stored previous arrays:** Reads z_prev/v_prev from BRAM before the latest z/v to compute differences.
- **Parametric multiplier:** Hard-wired to factor in ϱ (from solver cache) via fixed-point multiply.
- **Early exit:** If all four norms fall below their tolerances, the done output fires immediately to terminate ADMM.

4.8) Top-Level Hardware Interface

(from admm_solver.sv)

```
None
module admm_solver #(
       parameter STATE_DIM       = 12,   // Dimension of state vector (nx)
       parameter INPUT_DIM       = 4,   // Dimension of input vector (nu)
       parameter HORIZON         = 30,   // Maximum MPC horizon length (N)
       parameter MAX_ITER        = 100,  // Maximum ADMM iterations
       parameter EXT_DATA_WIDTH = 32,   // Avalon bus data width
       parameter DATA_WIDTH      = 16,   // Internal fixed-point width
       parameter FRAC_BITS       = 8,   // Fractional bits in fixed point
       parameter ADDR_WIDTH      = 16,   // Avalon address width
       parameter MEM_ADDR_WIDTH = 9     // Internal BRAM address width
)(
       input  logic                      clk,
       input  logic                      rst,
       input  logic [EXT_DATA_WIDTH-1:0] writedata,
       input  logic                      read,
       input  logic                      write,
       input  logic [ADDR_WIDTH-1:0]   addr,
       input  logic                      chipselect,
       output logic [EXT_DATA_WIDTH-1:0] readdata
);

       // … internal declarations, BRAM instantiations, FSM, sub-module
instantiations …

endmodule
```

4.9) Hardware Register Set

(from memory_interface.sv)

```
None
always_ff @(posedge clk or posedge rst) begin
       if (rst) begin
       // reset all registers
       readdata            <= 0;
       start_solving       <= 0;
       A_wren              <= 0;
       B_wren              <= 0;
       Q_wren              <= 0;
```

```
R_wren              <= 0;
x_ref_wren          <= 0;
u_ref_wren          <= 0;
pri_tol_new         <= 16'h0019;
dual_tol_new        <= 16'h0019;
rho_new             <= 16'h0100;
end else if (chipselect) begin
// default: clear write enables each cycle
A_wren              <= 0;
B_wren              <= 0;
Q_wren              <= 0;
R_wren              <= 0;
x_ref_wren          <= 0;
u_ref_wren          <= 0;

if (read) begin
       case (addr[15:12])
       4'h0: begin
              // Control & status registers
              case (addr[11:0])
                     12'h000: readdata <= {31'b0, solver_done};
                     12'h004: readdata <= current_iter;
                     12'h008: readdata <= active_horizon;
                     12'h00C: readdata <= converged;
                     12'h010: readdata <= pri_res_u;
                     12'h018: readdata <= pri_res_x;
                     12'h020: readdata <= dual_res;
                     default: readdata <= 0;
              endcase
       end

       4'h1: begin
              // Read state trajectory x[k]
              x_rdaddress <= addr[7:0] * STATE_DIM + addr[11:8];
              readdata     <= x_data_out;
       end

       4'h2: begin
              // Read input trajectory u[k]
              u_rdaddress <= addr[7:0] * INPUT_DIM + addr[11:8];
              readdata     <= u_data_out;
       end

       default: readdata <= 0;
```

```
            endcase

      end else if (write) begin
            case (addr[15:12])
            4'h0: begin
                  // Write control & tuning registers
                  case (addr[11:0])
                        12'h000: start_solving     <= writedata[0];
                        12'h004: begin
                        active_horizon_new <= writedata;
                        active_horizon_wren<= 1;
                        end
                        12'h008: pri_tol_new       <= writedata[15:0];
                        12'h010: dual_tol_new      <= writedata[15:0];
                        12'h018: rho_new           <= writedata[15:0];
                        default: ;
                  endcase
            end

            4'h1: begin
                  // Write input bounds u_min/u_max
                  if (addr[7])
                        u_max[addr[11:8]] <= writedata[15:0];
                  else
                        u_min[addr[11:8]] <= writedata[15:0];
            end

            // … additional write-cases for x_min, x_max, references, etc. …

            default: ;
            endcase
      end
      end
end

endmodule
```

We created testbench files for each module, passing in sample inputs and confirming that the module was executing the math equations correctly.

5) Quadrotor Simulator

5.1) Quadrotor Simulator Overview

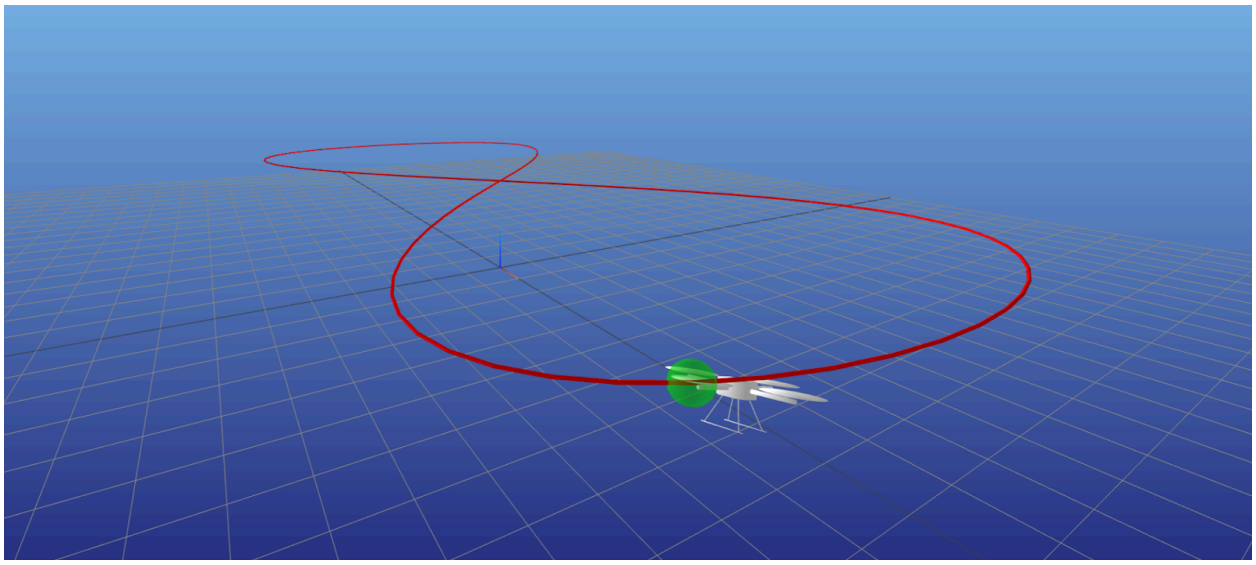

Figure 2: Photo of Quadrotor Simulation In Flight

System Architecture:

1. FPGA
    a. Hosts the ADMM-based MPC controller implemented in Verilog.
2. CPU
    a. Connected to the FPGA via an Avalon HPS-to-FPGA bridge.
    b. Transfers computed control inputs from the FPGA to the drone simulator via Ethernet.
3. External Computer
    a. Runs the drone simulator.
    b. Receives control inputs via UDP/TCP over Ethernet.
    c. Sends back the updated quadrotor state after each integration step.

Dynamics Modeling

The quadrotor attempts to follow a figure 8 trajectory. A physics engine simulates an imperfect drone, one that is weighed down by gravity and affected by wind.

- **Output Vector:** The quadrotor outputs its current state through 12 values: x, y, z, φ, θ, ψ, dx, dy, dz, dφ, dθ, dψ. These consist of position, orientation, linear velocity, and angular velocity in the x, y, and z dimensions. Each value is a 16-bit fixed-point integer, creating a total of 16 * 12 = 192 bits.

- **Input Vector:** The quadrotor accepts 4 values: u_1, u_2, u_3, u_4. These are motor thrust commands to the quadrotor's 4 motors. Thus there is a total of 4 * 16 = 64 bits.

5.2) Quadrotor Simulator Trajectory Visualization

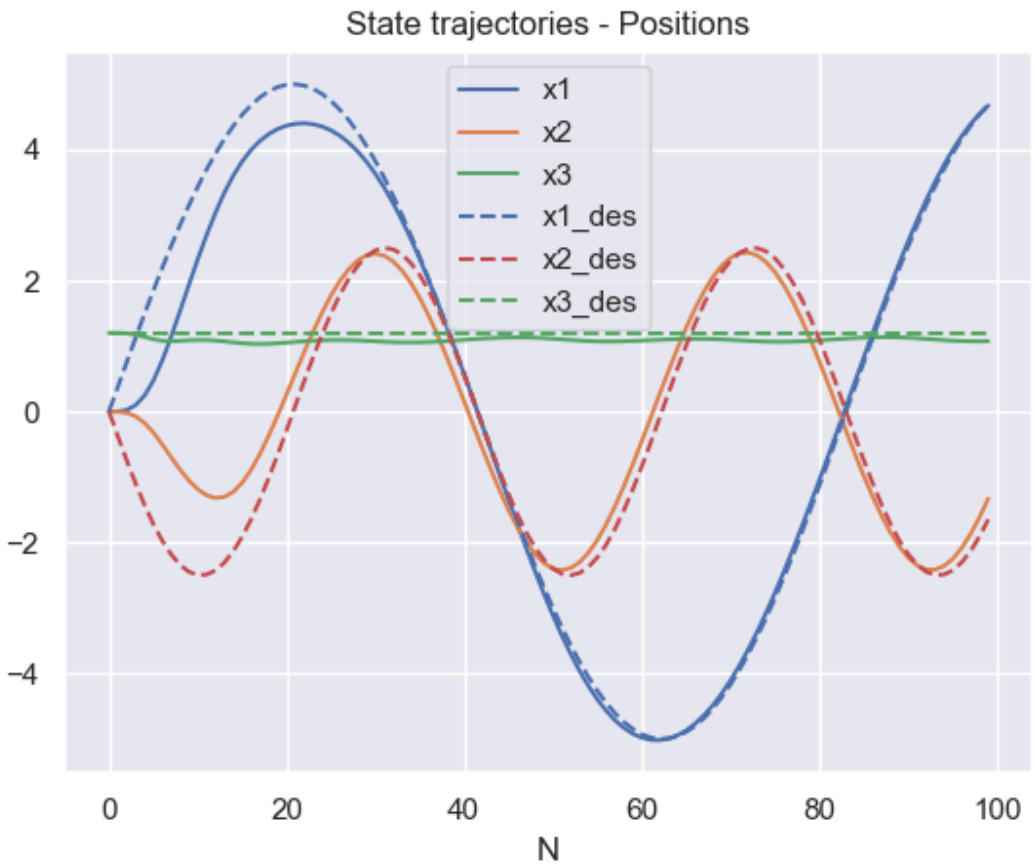

Figure 5.1

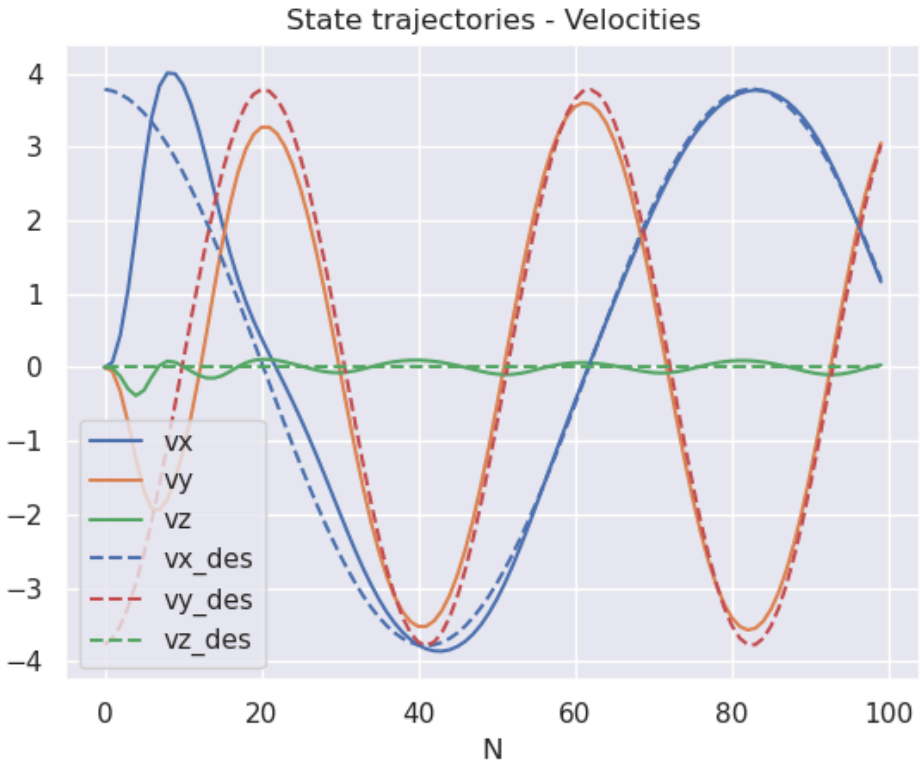

Figure 5.3

5.3) Simulation Implementation

5.3.1) simulator_server.py Overview:

The QuadroterSimulatorServer class provides a UDP-based simulation of the quadroter dynamics which seamlessly integrates with a client that sends motor commands and receives updated state estimates. On initialization, the server:

1. Creates and binds a Datagram socket on the user-specified host and port.
2. Stores a client endpoint (client_host:client_port) when first received, so all subsequent state packets are directed correctly.
3. Instantiates the QuadroterDynamics model, sets the simulation rate (simulator_hz) and timestep (dt = 1/simulator_hz), and seeds the state vector self.x with a small nonzero quaternion and zeroed position/velocities.
4. Also can launch the URDFQuadroter Visualizer, which opens a browser window and loads the provided UDRF (and mesh directory) for real-time 3D rendering of the vehicle's pose.

Once run() is called, the server enters a real-time loop at the desired frequency:

Receive Control: It attempts to recvfrom a 4-float packet (motor commands) with a short timeout. If no packet arrives, it falls back to hover thrust from QuadroterDynamics.

Simulate: It advances the state via quad.dynamics_rk4(self.x, u, dt), applying fourth-order Runge–Kutta integration. If visualization is enabled, it normalizes the quaternion, extracts the position and attitude, and updates the URDF visualizer.
Send State: It normalizes the quaternion again, packs the 12 state values (position, quaternion, velocity, angular velocity) into a big-endian struct.pack('!12f',...), and sends the binary packet back to the stored client address.
Timing: The loop enforces the fixed timestep by measuring elapsed time and sleeping for the remainder of dt. On keyboard interrupt, the loop breaks and the UDP socket is closed cleanly.

5.3.2) simulator_server.py:

```python
#!/usr/bin/env python3
# simulator_server.py
import socket
import struct
import time
import numpy as np
import argparse
import os
from quadrotor import QuadrotorDynamics
from visualizer_urdf import URDFQuadrotorVisualizer

class QuadrotorSimulatorServer:
    def __init__(self, host='0.0.0.0', port=12345, client_host='127.0.0.1',
client_port=12346, simulator_hz=50.0, visualization=True,
urdf_path="drone.urdf", mesh_dir="/home/alex/a2r/fpga-mpc/main-repo/python"):
        self.host = host
        self.port = port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind((self.host, self.port))


        if client_host and client_port:
            self.client_address = (client_host, client_port)
            print(f"Client address set to {client_host}:{client_port}")
        else:
            self.client_address = None


        self.quad = QuadrotorDynamics()
        self.simulator_hz = simulator_hz
        self.dt = 1.0 / simulator_hz
```

```
        self.x = 0.00001 * np.ones(13)
        self.x[0] = 1.0


        self.uhover = self.quad.hover_thrust


        self.visualization = visualization
        if self.visualization:

            self.visualizer = URDFQuadrotorVisualizer(urdf_path=urdf_path,
mesh_dir=mesh_dir, auto_open_browser=True)
            print("3D visualization started - browser window should open
automatically")
            if urdf_path:
                print(f"Using URDF model from: {urdf_path}")
            else:
                print("Using default quadrotor model")

        print(f"Simulator server initialized on {host}:{port}")
        print(f"Running at {simulator_hz} Hz (dt = {self.dt})")

    def send_state(self):

        if self.client_address is None:
            return


        fmt = '!12f'


        q_norm = self.x[4:8] / np.linalg.norm(self.x[4:8])


        state_to_send = np.concatenate([
            self.x[0:3],
            q_norm,
            self.x[7:10],
            self.x[10:12]
        ])


        binary_data = struct.pack(fmt, *state_to_send)
```

```
        self.socket.sendto(binary_data, self.client_address)

    def receive_control(self, timeout=0.01):

        self.socket.settimeout(timeout)
        try:

            fmt = '!4f'
            expected_size = struct.calcsize(fmt)

            data, addr = self.socket.recvfrom(expected_size)
            self.client_address = addr

            if len(data) == expected_size:

                controls = struct.unpack(fmt, data)
                return np.array(controls)
            else:
                print(f"Warning: Received packet of unexpected size
{len(data)}")
                return None

        except socket.timeout:
            return None
        except Exception as e:
            print(f"Error receiving control: {e}")
            return None

    def step_simulation(self, u):


        self.x = self.quad.dynamics_rk4(self.x, u, dt=self.dt)

        if self.visualization:

            position = self.x[0:3]
            quaternion = self.x[3:7] / np.linalg.norm(self.x[3:7])


            self.visualizer.update_quadrotor_state(position, quaternion)

    def run(self):

        print("Starting simulator server. Press Ctrl+C to stop.")
```

```
        if self.client_address is not None:
            print(f"Sending initial state to {self.client_address}")
            self.send_state()

        last_step_time = time.time()

        try:
            while True:

                current_time = time.time()
                elapsed = current_time - last_step_time


                if elapsed < self.dt:
                    time.sleep(self.dt - elapsed)


                u = self.receive_control()


                if u is None:
                    u = self.uhover


                self.step_simulation(u)


                self.send_state()


                last_step_time = time.time()

        except KeyboardInterrupt:
            print("\nSimulator server stopped.")
        finally:
            self.socket.close()

if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="Quadrotor Simulator Server")
    parser.add_argument("--host", type=str, default="0.0.0.0",
                        help="Host to bind the server to")
```

```
    parser.add_argument("--port", type=int, default=12345,
                        help="Port to bind the server to")
    parser.add_argument("--client-host", type=str, default="127.0.0.1",
                        help="Client host address")
    parser.add_argument("--client-port", type=int, default=12346,
                        help="Client port")
    parser.add_argument("--hz", type=float, default=50.0,
                        help="Simulation frequency in Hz")
    parser.add_argument("--no-viz", action="store_true",
                        help="Disable visualization")
    parser.add_argument("--urdf", type=str, default=None,
                        help="Path to URDF file for visualization")
    parser.add_argument("--mesh-dir", type=str, default=None,
                        help="Path to directory containing mesh files")

    args = parser.parse_args()

    server = QuadrotorSimulatorServer(
        host=args.host,
        port=args.port,
        client_host=args.client_host,
        client_port=args.client_port,
        simulator_hz=args.hz,
        visualization=not args.no_viz
    )

    server.run()
```

5.3.3) hardware_mpc_client.py Overview:

The HardwareMPCClient provides a simple UDP-based interface to send motor commands to and receive state updates from the quadrotor simulator or hardware accelerator. On initialization, the client takes a simulator IP address and port along with a local port for receiving replies. It constructs and binds a UDP socket to 0.0.0.0:client_port, printing its peer (server_host;server_port) and local bind port. All communication uses big-endian binary packing and the receive_state(timeout) method waits for a 12-float packet, unpacks it via struct.unpack, and returns a length-13 Numpy array representing position, quaternion, velocity, and angular velocity. If no data arrives or a malformed packet is received, the method logs a warning or timeout and returns None. The send_control(control) method validates that exactly four motor commands are provided, packs them, and transmits the binary frame to the stored simulator address.

The main() function wires these primitives together into a continuous control loop. After parsing command-line arguments, it creates the HardwareMPCClient and waits to receive a valid state. When a valid

state is received, the client extracts and logs each component (position, quaternion, velocity, angular velocity) for debugging. It then invokes example_hover_controller(state), a placeholder routine that computes equal hover thrust for each roter based on vehicle mass, gravity, and motor constant-standing in for a future hardware MPC call. The resulting 4-element control vector is sent back via send_control(). A brief sleep enforces a modest loop rate, and on KeyboardInterrupt the client cleanly closes its UDP socket before exiting.

5.3.4) hardware_mpc_client.py:

```python
#!/usr/bin/env python3
# hardware_mpc_client.py
import socket
import struct
import time
import numpy as np
import argparse

class HardwareMPCClient:
    def __init__(self, server_host='127.0.0.1', server_port=12345,
client_port=12346):


        self.server_address = (server_host, server_port)


        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind(('0.0.0.0', client_port))

        print(f"HardwareMPC client initialized, connecting to
{server_host}:{server_port}")
        print(f"Listening for responses on port {client_port}")

    def receive_state(self, timeout=1.0):

        self.socket.settimeout(timeout)
        try:

            fmt = '!12f'
            expected_size = struct.calcsize(fmt)

            data, addr = self.socket.recvfrom(expected_size)

            if len(data) == expected_size:
```

```
                state_tuple = struct.unpack(fmt, data)


                state = np.array(state_tuple)
                return state
            else:
                print(f"Warning: Received packet of unexpected size
{len(data)}")
                return None

        except socket.timeout:
            print("Timeout waiting for state from simulator")
            return None
        except Exception as e:
            print(f"Error receiving state: {e}")
            return None

    def send_control(self, control):

        if len(control) != 4:
            raise ValueError("Control must have exactly 4 elements")


        fmt = '!4f'
        binary_data = struct.pack(fmt, *control)


        self.socket.sendto(binary_data, self.server_address)

    def close(self):

        self.socket.close()

def example_hover_controller(state):

    mass = 0.035
    g = 9.81
    scale = 65535
    kt = 2.245365e-6 * scale


    hover_thrust = (mass * g / kt / 4.0) * np.ones(4)
```

```
    return hover_thrust

def main():
    parser = argparse.ArgumentParser(description="Hardware MPC Client")
    parser.add_argument("--server", type=str, default="127.0.0.1",
                        help="Simulator server IP address")
    parser.add_argument("--server-port", type=int, default=12345,
                        help="Simulator server port")
    parser.add_argument("--client-port", type=int, default=12346,
                        help="Local port to bind for receiving responses")
    args = parser.parse_args()

    client = HardwareMPCClient(
        server_host=args.server,
        server_port=args.server_port,
        client_port=args.client_port
    )

    try:
        while True:
            state = client.receive_state()

            if state is not None:

                position = state[0:3]
                quaternion = state[3:7]
                velocity = state[7:10]
                angular_velocity = state[10:13]

                print(f"Position: {position}")
                print(f"Quaternion: {quaternion}")
                print(f"Velocity: {velocity}")
                print(f"Angular Velocity: {angular_velocity}")


                u = example_hover_controller(state)
                print(f"Sending control: {u}")


                client.send_control(u)

                time.sleep(0.01)
```

```
    except KeyboardInterrupt:
        print("\nClient stopped.")
    finally:
        client.close()

if __name__ == "__main__":
    main()
```

We would like to cite Elena Oikonomou for providing a substantial code base to run the simulation.

6) Resources

Matrix Storage:

We assume Fixed-Point 16 Bit Entry:

Matrix	Size	Elements	Bytes
A	12×12	144	288 B
B	12×4	48	96 B
Q	12×12	144	288 B
R	4×4	16	32 B
Kinf	4×12 (gain matrix)	48	96 B
Pinf	12×12	144	288 B
C1	4×4	16	32 B
C2	12×12	144	288 B

Subtotal: 1048 B = **1.04 KB**

Table 6.1

Trajectory Matrices:

Variable	Size	Elements	Bytes (16-bit)
x	N_x X N	12×30 = 360	720 B
u	N_u X (N_1)	4×29 = 116	232 B
v, g, p, q	same as x	4 × 720 B	2880 B
z, y, d, r	same as u	4 × 232 B	928 B

Subtotal = **4.76 KB**

Table 6.2

Thus, our total On-Chip Memory will be approximately **5.8 KB** which falls well below the limit of 512 KB on the DE1.

7) Software Interface

7.1) Software Interface Overview

The ADMM accelerator is connected to the userspace through a character-device interface registered via the Linux "misc" subsystem, creating /dev/admm. When the module loads, the driver's problem function registers this misc device, gets the FPGA's registers from the device tree and requests access to that memory region, and maps it to the kernel's address space. In this phase, default solver parameters like horizon and primal/dual tolerances are initialized. When the driver unloads, it cleanly maps the I/O region, releases the memory region, and deregisters the misc device.

The connection to the hardware accelerator is essentially performed via three IOCTL commands defined in admm.h.

ADMM_WRITE_CONFIG: Userspace populates an admm_config_arg_t struct with solver included with solver parameters, initial, and reference states/input vectors. The driver copes the struct in from copy_from_user, then writes each field to the corresponding memory-mapped register (e.g., HORIZON, PRI_TOL, X_INIT, U_REF, etc.) via iowrite 32 which updates its internal dev.config.

ADMM_START: Writing this command issues a single iowrite32(1, START_SOLVER(...)), triggering the FPGA to begin ADMM iterations.

ADMM_READ_RESULT: After computation, userspace issues this IOCTL to copy back an admm_result_arg_t. The driver's read_result helper reads status registers-iteration count, convergence flag, primal and dual residuals-and then streams out the computed state/control trajectory over the configured horizon via repeated ioread32 calls, filling the result struct before copying it back with copy_to_user.

7.2) admm.c

```C/C++
/*
 * Device driver for the ADMM FPGA Accelerator
 *
 * Adapted from vga_ball.c
 */

#include "admm.h"
#include <linux/errno.h>
#include <linux/fs.h>
```

```c
#include <linux/init.h>
#include <linux/io.h>
#include <linux/kernel.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/platform_device.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>

#define DRIVER_NAME "admm"

struct admm_dev {
  struct resource res;
  void __iomem *virtbase;
  admm_config_t config;
  admm_result_t result;
} dev;



static void set_initial_state(const int16_t x_init[STATE_DIM],
                              const int16_t u_init[INPUT_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_init[i], X_INIT(dev.virtbase, i));
    dev.config.initial_state.x[i] = x_init[i];
  }

  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_init[i], U_INIT(dev.virtbase, i));
    dev.config.initial_state.u[i] = u_init[i];
  }
}

static void set_reference_trajectory(const int16_t x_ref[STATE_DIM],
                                     const int16_t u_ref[INPUT_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_ref[i], X_REF(dev.virtbase, i));
    dev.config.reference.x[i] = x_ref[i];
  }
```

```
  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_ref[i], U_REF(dev.virtbase, i));
    dev.config.reference.u[i] = u_ref[i];
  }
}

static void read_result(admm_result_t *result) {

  result->iterations = ioread32(CUR_ITER(dev.virtbase));
  result->converged = ioread32(CONVERGED(dev.virtbase));
  result->pri_res_u = ioread32(PRI_RES_U(dev.virtbase));
  result->pri_res_x = ioread32(PRI_RES_X(dev.virtbase));
  result->dual_res = ioread32(DUAL_RES(dev.virtbase));


  int i, j;
  for (i = 0; i < dev.config.horizon; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      result->trajectory[i].x[j] = ioread32(X_READ(dev.virtbase, i, j));
    }
    for (j = 0; j < INPUT_DIM; j++) {
      result->trajectory[i].u[j] = ioread32(U_READ(dev.virtbase, i, j));
    }
  }

  dev.result = *result;
}


static void set_solver_params(uint32_t horizon, int16_t pri_tol,
                              int16_t dual_tol, int16_t rho) {
  iowrite32(horizon, HORIZON(dev.virtbase));
  iowrite32(pri_tol, PRI_TOL(dev.virtbase));
  iowrite32(dual_tol, DUAL_TOL(dev.virtbase));
  iowrite32(rho, RHO(dev.virtbase));

  dev.config.horizon = horizon;
  dev.config.pri_tol = pri_tol;
  dev.config.dual_tol = dual_tol;
  dev.config.rho = rho;
}

static void set_state_bounds(const int16_t x_min[STATE_DIM],
```

```
                              const int16_t x_max[STATE_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_min[i], X_MIN(dev.virtbase, i));
    iowrite32(x_max[i], X_MAX(dev.virtbase, i));
  }
}

static void set_input_bounds(const int16_t u_min[INPUT_DIM],
                              const int16_t u_max[INPUT_DIM]) {
  int i;
  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_min[i], U_MIN(dev.virtbase, i));
    iowrite32(u_max[i], U_MAX(dev.virtbase, i));
  }
}

static void set_system_matrix_A(const int16_t A[STATE_DIM][STATE_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      iowrite32(A[i][j], MATRIX_A(dev.virtbase, i, j));
    }
  }
}

static void set_system_matrix_B(const int16_t B[STATE_DIM][INPUT_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < INPUT_DIM; j++) {
      iowrite32(B[i][j], MATRIX_B(dev.virtbase, i, j));
    }
  }
}

static void set_cost_matrix_Q(const int16_t Q[STATE_DIM][STATE_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      iowrite32(Q[i][j], MATRIX_Q(dev.virtbase, i, j));
    }
  }
}
```

```
static void set_cost_matrix_R(const int16_t R[INPUT_DIM][INPUT_DIM]) {
  int i, j;
  for (i = 0; i < INPUT_DIM; i++) {
    for (j = 0; j < INPUT_DIM; j++) {
      iowrite32(R[i][j], MATRIX_R(dev.virtbase, i, j));
    }
  }
}

static void write_config(admm_config_t *config) {
  set_solver_params(config->horizon, config->pri_tol, config->dual_tol,
                    config->rho);
  set_initial_state(config->initial_state.x, config->initial_state.u);
  set_reference_trajectory(config->reference.x, config->reference.u);

  dev.config = *config;
}

static long admm_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
  admm_config_arg_t config_arg;
  admm_result_arg_t result_arg;

  switch (cmd) {
  case ADMM_WRITE_CONFIG:
    if (copy_from_user(&config_arg, (admm_config_arg_t *)arg,
                       sizeof(admm_config_arg_t)))
      return -EACCES;
    write_config(&config_arg.config);
    break;

  case ADMM_READ_RESULT:
    read_result(&dev.result);
    result_arg.result = dev.result;
    if (copy_to_user((admm_result_arg_t *)arg, &result_arg,
                     sizeof(admm_result_arg_t)))
      return -EACCES;
    break;

  case ADMM_START:
    iowrite32(1, START_SOLVER(dev.virtbase));
    break;

  default:
    return -EINVAL;
```

```
  }

  return 0;
}


static const struct file_operations admm_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = admm_ioctl,
};


static struct miscdevice admm_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &admm_fops,
};

static int __init admm_probe(struct platform_device *pdev) {
  int ret;


  ret = misc_register(&admm_misc_device);


  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
    ret = -ENOENT;
    goto out_deregister;
  }


  if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME)
==
      NULL) {
    ret = -EBUSY;
    goto out_deregister;
  }

  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
  }
```

```
  memset(&dev.config, 0, sizeof(dev.config));

  dev.config.horizon = 10;
  dev.config.pri_tol = TO_FIXED(0.01);
  dev.config.dual_tol = TO_FIXED(0.01);
  dev.config.rho = TO_FIXED(1.0);

  return 0;

out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&admm_misc_device);
  return ret;
}

static int admm_remove(struct platform_device *pdev) {
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&admm_misc_device);
  return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id admm_of_match[] = {
    {.compatible = "csee4840,admm-1.0"},
    {},
};
MODULE_DEVICE_TABLE(of, admm_of_match);
#endif


static struct platform_driver admm_driver = {
    .driver =
        {
            .name = DRIVER_NAME,
            .owner = THIS_MODULE,
            .of_match_table = of_match_ptr(admm_of_match),
        },
    .remove = __exit_p(admm_remove),
};
```

```
static int __init admm_init(void) {
  pr_info(DRIVER_NAME ": init\n");
  return platform_driver_probe(&admm_driver, admm_probe);
}


static void __exit admm_exit(void) {
  platform_driver_unregister(&admm_driver);
  pr_info(DRIVER_NAME ": exit\n");
}

module_init(admm_init);
module_exit(admm_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alex Du");
MODULE_DESCRIPTION("Driver for FPGA-MPC's ADMM Accelerator");
```

8) Team Member Roles

Alex: Had the idea to work on TinyMPC on the FPGA, wrote software version of algorithm, C software interface, and external simulator. Designed the high level system and did some integration.

Apurva: Wrote the working SystemVerilog modules that can run on Quartus and take into account memory management.

Godwill: Integrated the SystemVerilog modules into Quartus and the FPGA.

Roy: Designed the skeleton of the SystemVerilog modules that execute each step of the ADMM Algorithm. Created test bench files and verified success in ModelSim.

9) Code

9.1) admm_solver.sv

```
None
module admm_solver #(
    parameter STATE_DIM     = 12,        // Dimension of state vector (nx)
    parameter INPUT_DIM     = 4,         // Dimension of input vector (nu)
    parameter HORIZON       = 30,        // Maximum MPC horizon length (N)
    parameter MAX_ITER      = 100,       // Maximum ADMM iterations
    parameter EXT_DATA_WIDTH = 32,       // Bus data width
    parameter DATA_WIDTH    = 16,        // Internal calculation width (16-bit
fixed point)
    parameter FRAC_BITS     = 8,         // Number of fractional bits for fixed
point
    parameter ADDR_WIDTH    = 16,        // Address width for bus interface
    parameter MEM_ADDR_WIDTH = 9         // Internal memory address width
)(
    // Clock and reset
    input logic clk,
    input logic rst,

    // Avalon memory-mapped slave interface
    input logic [EXT_DATA_WIDTH-1:0] writedata,
    input logic read,
    input logic write,
    input logic [ADDR_WIDTH-1:0] addr,
    input logic chipselect,
    output logic [EXT_DATA_WIDTH-1:0] readdata
);

    // State machine states
    typedef enum logic [2:0] {
        IDLE = 3'd0,
        INIT = 3'd1,
        PRIMAL_UPDATE = 3'd2,  // X-Update (Riccati recursion & trajectory
rollout)
        SLACK_UPDATE = 3'd3,  // Z-Update (Projection)
        DUAL_UPDATE = 3'd4,  // Y-Update (Dual variables)
        CHECK_CONVERGENCE = 3'd5,
        OUTPUT_RESULTS = 3'd6,
        DONE = 3'd7
```

```
} solver_state_t;

solver_state_t state;

// Control signals
logic start_solving;
logic solver_done;
logic [31:0] current_iter;
logic [31:0] active_horizon_reg; // INTERNAL REG ONLY
wire [31:0] active_horizon = active_horizon_reg;
logic [31:0] active_horizon_new;
logic active_horizon_wren;
logic converged;

// Memory signals for system matrices
logic [MEM_ADDR_WIDTH-1:0] A_rdaddress, A_wraddress;
logic [DATA_WIDTH-1:0] A_data_out, A_data_in;
logic A_wren;

logic [MEM_ADDR_WIDTH-1:0] B_rdaddress, B_wraddress;
logic [DATA_WIDTH-1:0] B_data_out, B_data_in;
logic B_wren;

logic [MEM_ADDR_WIDTH-1:0] Q_rdaddress, Q_wraddress;
logic [DATA_WIDTH-1:0] Q_data_out, Q_data_in;
logic Q_wren;

logic [MEM_ADDR_WIDTH-1:0] R_rdaddress, R_wraddress;
logic [DATA_WIDTH-1:0] R_data_out, R_data_in;
logic R_wren;

// Memory signals for precomputed terms
logic [MEM_ADDR_WIDTH-1:0] K_rdaddress, K_wraddress;
logic [DATA_WIDTH-1:0] K_data_out, K_data_in;
logic K_wren;

logic [MEM_ADDR_WIDTH-1:0] C1_rdaddress, C1_wraddress;
logic [DATA_WIDTH-1:0] C1_data_out, C1_data_in;
logic C1_wren;

logic [MEM_ADDR_WIDTH-1:0] C2_rdaddress, C2_wraddress;
logic [DATA_WIDTH-1:0] C2_data_out, C2_data_in;
logic C2_wren;
```

```
logic [MEM_ADDR_WIDTH-1:0] P_rdaddress, P_wraddress;
logic [DATA_WIDTH-1:0] P_data_out, P_data_in;
logic P_wren;

// Memory signals for trajectories
logic [MEM_ADDR_WIDTH-1:0] x_rdaddress, x_wraddress;
logic [DATA_WIDTH-1:0] x_data_out, x_data_in;
logic x_wren;

logic [MEM_ADDR_WIDTH-1:0] u_rdaddress, u_wraddress;
logic [DATA_WIDTH-1:0] u_data_out, u_data_in;
logic u_wren;

// Memory signals for auxiliary variables
logic [MEM_ADDR_WIDTH-1:0] z_rdaddress, z_wraddress;
logic [DATA_WIDTH-1:0] z_data_out, z_data_in;
logic z_wren;

logic [MEM_ADDR_WIDTH-1:0] v_rdaddress, v_wraddress;
logic [DATA_WIDTH-1:0] v_data_out, v_data_in;
logic v_wren;

// Memory signals for previous values (for residuals)
logic [MEM_ADDR_WIDTH-1:0] z_prev_rdaddress, z_prev_wraddress;
logic [DATA_WIDTH-1:0] z_prev_data_out, z_prev_data_in;
logic z_prev_wren;

logic [MEM_ADDR_WIDTH-1:0] v_prev_rdaddress, v_prev_wraddress;
logic [DATA_WIDTH-1:0] v_prev_data_out, v_prev_data_in;
logic v_prev_wren;

// Memory signals for dual variables
logic [MEM_ADDR_WIDTH-1:0] y_rdaddress, y_wraddress;
logic [DATA_WIDTH-1:0] y_data_out, y_data_in;
logic y_wren;

logic [MEM_ADDR_WIDTH-1:0] g_rdaddress, g_wraddress;
logic [DATA_WIDTH-1:0] g_data_out, g_data_in;
logic g_wren;

// Memory signals for linear cost terms
logic [MEM_ADDR_WIDTH-1:0] q_rdaddress, q_wraddress;
logic [DATA_WIDTH-1:0] q_data_out, q_data_in;
logic q_wren;
```

```
logic [MEM_ADDR_WIDTH-1:0] r_rdaddress, r_wraddress;
logic [DATA_WIDTH-1:0] r_data_out, r_data_in;
logic r_wren;

logic [MEM_ADDR_WIDTH-1:0] p_rdaddress, p_wraddress;
logic [DATA_WIDTH-1:0] p_data_out, p_data_in;
logic p_wren;

logic [MEM_ADDR_WIDTH-1:0] d_rdaddress, d_wraddress;
logic [DATA_WIDTH-1:0] d_data_out, d_data_in;
logic d_wren;

logic [MEM_ADDR_WIDTH-1:0] x_ref_rdaddress, x_ref_wraddress;
logic [DATA_WIDTH-1:0] x_ref_data_out, x_ref_data_in;
logic x_ref_wren;

logic [MEM_ADDR_WIDTH-1:0] u_ref_rdaddress, u_ref_wraddress;
logic [DATA_WIDTH-1:0] u_ref_data_out, u_ref_data_in;
logic u_ref_wren;

// Stage control signals
logic primal_update_start, primal_update_done;
logic slack_update_start, slack_update_done;
logic dual_update_start, dual_update_done;
logic residual_calc_start, residual_calc_done;

// Residuals
logic [DATA_WIDTH-1:0] pri_res_u, pri_res_x, dual_res;
logic [DATA_WIDTH-1:0] pri_tol, dual_tol;

logic [DATA_WIDTH-1:0] pri_tol_new, dual_tol_new;

// Bounds
logic [DATA_WIDTH-1:0] u_min [INPUT_DIM];
logic [DATA_WIDTH-1:0] u_max [INPUT_DIM];
logic [DATA_WIDTH-1:0] x_min [STATE_DIM];
logic [DATA_WIDTH-1:0] x_max [STATE_DIM];

// Initial state
logic [DATA_WIDTH*STATE_DIM-1:0] x_init;

// Reference trajectories
logic [STATE_DIM*HORIZON*DATA_WIDTH-1:0] x_ref;
```

```
logic [INPUT_DIM*(HORIZON-1)*DATA_WIDTH-1:0] u_ref;

// ADMM parameter
logic [DATA_WIDTH-1:0] rho;
logic [DATA_WIDTH-1:0] rho_new;

// Memory blocks for system matrices and trajectories using RAM modules

// System matrices - A matrix (STATE_DIM x STATE_DIM)
// A_ram (already provided)
soc_system_RAM A_ram (
    .clk(clk),
    .writedata(A_data_in),
    .address(A_wren ? A_wraddress : A_rdaddress),
    .write(A_wren),
    .readdata(A_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// System matrices - B matrix (STATE_DIM x INPUT_DIM)
soc_system_RAM B_ram (
    .clk(clk),
    .writedata(B_data_in),
    .address(B_wren ? B_wraddress : B_rdaddress),
    .write(B_wren),
    .readdata(B_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// System matrices - Q matrix (STATE_DIM x STATE_DIM)
soc_system_RAM Q_ram (
    .clk(clk),
    .writedata(Q_data_in),
    .address(Q_wren ? Q_wraddress : Q_rdaddress),
```

```
        .write(Q_wren),
        .readdata(Q_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // System matrices - R matrix (INPUT_DIM x INPUT_DIM)
    soc_system_RAM R_ram (
        .clk(clk),
        .writedata(R_data_in),
        .address(R_wren ? R_wraddress : R_rdaddress),
        .write(R_wren),
        .readdata(R_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Precomputed cache terms - K matrix (Kinf) (INPUT_DIM x STATE_DIM)
    soc_system_RAM K_ram (
        .clk(clk),
        .writedata(K_data_in),
        .address(K_wren ? K_wraddress : K_rdaddress),
        .write(K_wren),
        .readdata(K_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Precomputed cache terms - C1 matrix ((R + B'*P*B)^-1) (INPUT_DIM x
INPUT_DIM)
    soc_system_RAM C1_ram (
        .clk(clk),
```

```
        .writedata(C1_data_in),
        .address(C1_wren ? C1_wraddress : C1_rdaddress),
        .write(C1_wren),
        .readdata(C1_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Precomputed cache terms - C2 matrix ((A - B*K)') (STATE_DIM x STATE_DIM)
    soc_system_RAM C2_ram (
        .clk(clk),
        .writedata(C2_data_in),
        .address(C2_wren ? C2_wraddress : C2_rdaddress),
        .write(C2_wren),
        .readdata(C2_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Precomputed cache terms - P matrix (Pinf) (STATE_DIM x STATE_DIM)
    soc_system_RAM P_ram (
        .clk(clk),
        .writedata(P_data_in),
        .address(P_wren ? P_wraddress : P_rdaddress),
        .write(P_wren),
        .readdata(P_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Trajectories - x trajectory (STATE_DIM x HORIZON)
    soc_system_RAM x_ram (
```

```
        .clk(clk),
        .writedata(x_data_in),
        .address(x_wren ? x_wraddress : x_rdaddress),
        .write(x_wren),
        .readdata(x_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Trajectories - u trajectory (INPUT_DIM x (HORIZON-1))
    soc_system_RAM u_ram (
        .clk(clk),
        .writedata(u_data_in),
        .address(u_wren ? u_wraddress : u_rdaddress),
        .write(u_wren),
        .readdata(u_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Trajectories - z trajectory (INPUT_DIM x (HORIZON-1))
    soc_system_RAM z_ram (
        .clk(clk),
        .writedata(z_data_in),
        .address(z_wren ? z_wraddress : z_rdaddress),
        .write(z_wren),
        .readdata(z_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    // Trajectories - v trajectory (STATE_DIM x HORIZON)
```

```
soc_system_RAM v_ram (
    .clk(clk),
    .writedata(v_data_in),
    .address(v_wren ? v_wraddress : v_rdaddress),
    .write(v_wren),
    .readdata(v_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Previous values for residuals - z_prev (INPUT_DIM x (HORIZON-1))
soc_system_RAM z_prev_ram (
    .clk(clk),
    .writedata(z_prev_data_in),
    .address(z_prev_wren ? z_prev_wraddress : z_prev_rdaddress),
    .write(z_prev_wren),
    .readdata(z_prev_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Previous values for residuals - v_prev (STATE_DIM x HORIZON)
soc_system_RAM v_prev_ram (
    .clk(clk),
    .writedata(v_prev_data_in),
    .address(v_prev_wren ? v_prev_wraddress : v_prev_rdaddress),
    .write(v_prev_wren),
    .readdata(v_prev_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);
```

```
// Dual variables - y (INPUT_DIM x (HORIZON-1))
soc_system_RAM y_ram (
    .clk(clk),
    .writedata(y_data_in),
    .address(y_wren ? y_wraddress : y_rdaddress),
    .write(y_wren),
    .readdata(y_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Dual variables - g (STATE_DIM x HORIZON)
soc_system_RAM g_ram (
    .clk(clk),
    .writedata(g_data_in),
    .address(g_wren ? g_wraddress : g_rdaddress),
    .write(g_wren),
    .readdata(g_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Linear cost terms - q (STATE_DIM x HORIZON)
soc_system_RAM q_ram (
    .clk(clk),
    .writedata(q_data_in),
    .address(q_wren ? q_wraddress : q_rdaddress),
    .write(q_wren),
    .readdata(q_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);
```

```
// Linear cost terms - r (INPUT_DIM x (HORIZON-1))
soc_system_RAM r_ram (
    .clk(clk),
    .writedata(r_data_in),
    .address(r_wren ? r_wraddress : r_rdaddress),
    .write(r_wren),
    .readdata(r_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Linear cost terms - p (STATE_DIM x HORIZON)
soc_system_RAM p_ram (
    .clk(clk),
    .writedata(p_data_in),
    .address(p_wren ? p_wraddress : p_rdaddress),
    .write(p_wren),
    .readdata(p_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
);

// Linear cost terms - d (INPUT_DIM x (HORIZON-1))
soc_system_RAM d_ram (
    .clk(clk),
    .writedata(d_data_in),
    .address(d_wren ? d_wraddress : d_rdaddress),
    .write(d_wren),
    .readdata(d_data_out),
    .reset(rst),
    .chipselect(1'b1),
    .clken(1'b1),
    .freeze(1'b0),
    .reset_req(1'b0),
    .byteenable(2'b11)
```

```
    );

    soc_system_RAM x_ref_ram (
        .clk(clk),
        .writedata(x_ref_data_in),
        .address(x_ref_wren ? x_ref_wraddress : x_ref_rdaddress),
        .write(x_ref_wren),
        .readdata(x_ref_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );

    soc_system_RAM u_ref_ram (
        .clk(clk),
        .writedata(u_ref_data_in),
        .address(u_ref_wren ? u_ref_wraddress : u_ref_rdaddress),
        .write(u_ref_wren),
        .readdata(u_ref_data_out),
        .reset(rst),
        .chipselect(1'b1),
        .clken(1'b1),
        .freeze(1'b0),
        .reset_req(1'b0),
        .byteenable(2'b11)
    );


    // Instantiate solver stage 1 - X-Update (Riccati recursion & trajectory
rollout)
    primal_update #(
        .STATE_DIM(STATE_DIM),
        .INPUT_DIM(INPUT_DIM),
        .HORIZON(HORIZON),
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(MEM_ADDR_WIDTH)
    ) primal_update_inst (
        .clk(clk),
        .rst(rst),
        .start(primal_update_start),
```

```
// System matrices
.A_rdaddress(A_rdaddress),
.A_data_out(A_data_out),
.B_rdaddress(B_rdaddress),
.B_data_out(B_data_out),
.K_rdaddress(K_rdaddress),
.K_data_out(K_data_out),
.C1_rdaddress(C1_rdaddress),
.C1_data_out(C1_data_out),
.C2_rdaddress(C2_rdaddress),
.C2_data_out(C2_data_out),

// Initial state
.x_init(x_init),

// Linear cost terms
.p_rdaddress(p_rdaddress),
.p_data_out(p_data_out),
.r_rdaddress(r_rdaddress),
.r_data_out(r_data_out),
.q_rdaddress(q_rdaddress),
.q_data_out(q_data_out),

// Output trajectories
.x_wraddress(x_wraddress),
.x_data_in(x_data_in),
.x_wren(x_wren),

.u_wraddress(u_wraddress),
.u_data_in(u_data_in),
.u_wren(u_wren),

.d_wraddress(d_wraddress),
.d_data_in(d_data_in),
.d_wren(d_wren),

// Added p vector write ports
.p_wraddress(p_wraddress),
.p_data_in(p_data_in),
.p_wren(p_wren),

// Configuration
.active_horizon(active_horizon_reg),
```

```
    .done(primal_update_done)
);

// Instantiate solver stage 2 - Z-Update (Projection)
slack_update #(
    .STATE_DIM(STATE_DIM),
    .INPUT_DIM(INPUT_DIM),
    .HORIZON(HORIZON),
    .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(MEM_ADDR_WIDTH)
) slack_update_inst (
    .clk(clk),
    .rst(rst),
    .start(slack_update_start),

    // Trajectory inputs
    .x_rdaddress(x_rdaddress),
    .x_data_out(x_data_out),
    .u_rdaddress(u_rdaddress),
    .u_data_out(u_data_out),
    .y_rdaddress(y_rdaddress),
    .y_data_out(y_data_out),
    .g_rdaddress(g_rdaddress),
    .g_data_out(g_data_out),
    // Bounds
    .u_min(u_min),
    .u_max(u_max),
    .x_min(x_min),
    .x_max(x_max),

    // Output auxiliary variables
    .z_wraddress(z_wraddress),
    .z_data_in(z_data_in),
    .z_wren(z_wren),
    .v_wraddress(v_wraddress),
    .v_data_in(v_data_in),
    .v_wren(v_wren),

    // Previous values for residuals
    //.z_prev_wraddress(z_prev_wraddress),
    //.z_prev_data_in(z_prev_data_in),
    //.z_prev_wren(z_prev_wren),

    // Configuration
```

```
        .active_horizon(active_horizon_reg),

        .done(slack_update_done)
    );

    // Instantiate solver stage 3 - Y-Update (Dual variables)
    dual_update #(
        .STATE_DIM(STATE_DIM),
        .INPUT_DIM(INPUT_DIM),
        .HORIZON(HORIZON),
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(MEM_ADDR_WIDTH)
    ) dual_update_inst (
        .clk(clk),
        .rst(rst),
        .start(dual_update_start),

        // Trajectory inputs
        .x_rdaddress(x_rdaddress),
        .x_data_out(x_data_out),
        .u_rdaddress(u_rdaddress),
        .u_data_out(u_data_out),
        .z_rdaddress(z_rdaddress),
        .z_data_out(z_data_out),
        .v_rdaddress(v_rdaddress),
        .v_data_out(v_data_out),

        // Dual variables
        .y_rdaddress(y_rdaddress),
        .y_data_out(y_data_out),
        .y_wraddress(y_wraddress),
        .y_data_in(y_data_in),
        .y_wren(y_wren),

        .g_rdaddress(g_rdaddress),
        .g_data_out(g_data_out),
        .g_wraddress(g_wraddress),
        .g_data_in(g_data_in),
        .g_wren(g_wren),

        // Linear cost terms
        .r_rdaddress(r_rdaddress),
        .r_data_out(r_data_out),
        .r_wraddress(r_wraddress),
```

```
        .r_data_in(r_data_in),
        .r_wren(r_wren),

        .q_rdaddress(q_rdaddress),
        .q_data_out(q_data_out),
        .q_wraddress(q_wraddress),
        .q_data_in(q_data_in),
        .q_wren(q_wren),

        .p_rdaddress(p_rdaddress),
        .p_data_out(p_data_out),
        .p_wraddress(p_wraddress),
        .p_data_in(p_data_in),
        .p_wren(p_wren),

        // Reference trajectories
        //.x_ref(x_ref),
        //.u_ref(u_ref),


        // Cost matrices
        .R_rdaddress(R_rdaddress),
        .R_data_out(R_data_out),
        .Q_rdaddress(Q_rdaddress),
        .Q_data_out(Q_data_out),
        .P_rdaddress(P_rdaddress),
        .P_data_out(P_data_out),

        // ADMM parameter
        .rho(rho),

        // Residuals
        .pri_res_u(pri_res_u),
        .pri_res_x(pri_res_x),

        // Configuration
        .active_horizon(active_horizon_reg),

        .done(dual_update_done)
    );

    // Instantiate residual calculator for convergence check
    residual_calculator #(
        .STATE_DIM(STATE_DIM),
```

```
        .INPUT_DIM(INPUT_DIM),
        .HORIZON(HORIZON),
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(MEM_ADDR_WIDTH)
    ) residual_calc_inst (
        .clk(clk),
        .rst(rst),
        .start(residual_calc_start),

        // Residual inputs from dual_update
        .pri_res_u(pri_res_u),
        .pri_res_x(pri_res_x),

        .z_rdaddress(z_rdaddress),
        .z_data_out(z_data_out),
        .z_prev_rdaddress(z_prev_rdaddress),
        .z_prev_data_out(z_prev_data_out),
        .pri_tol(pri_tol),
        .dual_tol(dual_tol),
        .active_horizon(active_horizon_reg),

        // Outputs
        .dual_res(dual_res),
        .converged(converged),
        .done(residual_calc_done)
    );

    // Main state machine to control the ADMM solver
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            current_iter <= 0;
            active_horizon_reg <= HORIZON;
            solver_done <= 0;

            // Initialize control signals
            primal_update_start <= 0;
            slack_update_start <= 0;
            dual_update_start <= 0;
            residual_calc_start <= 0;

            // Initialize tolerances
            pri_tol <= 16'h0080; // 0.001 in 16-bit fixed point
            dual_tol <= 16'h0080;
```

```
        rho <= 16'h1000; // 1.0

    end else begin
        // Default values for control signals
        primal_update_start <= 0;
        slack_update_start <= 0;
        dual_update_start <= 0;
        residual_calc_start <= 0;
       active_horizon_reg <= HORIZON;
        pri_tol <= 16'h0080; // 0.001 in 16-bit fixed point
        dual_tol <= 16'h0080;

        rho <= 16'h1000; // 1.0

        case (state)
            IDLE: begin
                if (start_solving) begin
                    state <= INIT;
                    current_iter <= 0;
                    solver_done <= 0;
                end
            end

            INIT: begin
                // TODO: INIT
                state <= PRIMAL_UPDATE;
            end

            PRIMAL_UPDATE: begin
                // X-Update stage (Riccati recursion and forward rollout)
                if (!primal_update_start && !primal_update_done) begin
                    primal_update_start <= 1;
                end else if (primal_update_done) begin
                    state <= SLACK_UPDATE;
                end
            end

            SLACK_UPDATE: begin
                // Z-Update stage (Projection)
                if (!slack_update_start && !slack_update_done) begin
                    slack_update_start <= 1;
                end else if (slack_update_done) begin
                    state <= DUAL_UPDATE;
```

```
                end
            end

            DUAL_UPDATE: begin
                // Y-Update stage (Dual variables)
                if (!dual_update_start && !dual_update_done) begin
                    dual_update_start <= 1;
                end else if (dual_update_done) begin
                    state <= CHECK_CONVERGENCE;
                end
            end

            CHECK_CONVERGENCE: begin
                // Check convergence criteria
                if (!residual_calc_start && !residual_calc_done) begin
                    residual_calc_start <= 1;
                end else if (residual_calc_done) begin
                    if (converged || current_iter >= MAX_ITER-1) begin
                        state <= OUTPUT_RESULTS;
                    end else begin
                        current_iter <= current_iter + 1;
                        state <= PRIMAL_UPDATE; // Start next iteration
                    end
                end
            end

            OUTPUT_RESULTS: begin
                //TODO: FINISH
                state <= DONE;
            end

            DONE: begin
                solver_done <= 1;
                if (!start_solving) begin
                    state <= IDLE; // Return to IDLE when start is
deasserted
                end
            end

            default: state <= IDLE;
        endcase
    end
end
```

```
// Instantiate memory interface module
memory_interface #(
    .STATE_DIM(STATE_DIM),
    .INPUT_DIM(INPUT_DIM),
    .HORIZON(HORIZON),
    .EXT_DATA_WIDTH(EXT_DATA_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .FRAC_BITS(FRAC_BITS),
    .ADDR_WIDTH(ADDR_WIDTH),
    .MEM_ADDR_WIDTH(MEM_ADDR_WIDTH)
) mem_if (
    // Clock and reset
    .clk(clk),
    .rst(rst),

    // Avalon memory-mapped slave interface
    .writedata(writedata),
    .read(read),
    .write(write),
    .addr(addr),
    .chipselect(chipselect),
    .readdata(readdata),

    // Control signals
    .solver_done(solver_done),
    .current_iter(current_iter),
    .active_horizon(active_horizon_reg),
    .active_horizon_new(active_horizon_new),
   .active_horizon_wren(active_horizon_wren),
    .converged(converged),
    .start_solving(start_solving),

    // Residuals
    .pri_res_u(pri_res_u),
    .pri_res_x(pri_res_x),
    .dual_res(dual_res),
    .pri_tol(pri_tol),
   .pri_tol_new(pri_tol_new),
    .dual_tol(dual_tol),
   .dual_tol_new(dual_tol_new),

    // ADMM parameter
    .rho(rho),
```

```
 // Bounds
 .u_min(u_min),
 .u_max(u_max),
 .x_min(x_min),
 .x_max(x_max),

 // Initial state
 .x_init(x_init),

 // Reference trajectories
 //.x_ref(x_ref),
 //.u_ref(u_ref),

.x_ref_wraddress(x_ref_wraddress),
 .x_ref_data_in(x_ref_data_in),
.x_ref_wren(x_ref_ren),
.u_ref_wdaddress(x_wraddress),
 .u_ref_data_in(u_ref_data_in),
.u_ref_wren(u_ref_ren),

 // Memory signals for system matrices
 .A_rdaddress(A_rdaddress),
 .A_wraddress(A_wraddress),
 .A_data_in(A_data_in),
 .A_data_out(A_data_out),
 .A_wren(A_wren),

 .B_rdaddress(B_rdaddress),
 .B_wraddress(B_wraddress),
 .B_data_in(B_data_in),
 .B_data_out(B_data_out),
 .B_wren(B_wren),

 .Q_rdaddress(Q_rdaddress),
 .Q_wraddress(Q_wraddress),
 .Q_data_in(Q_data_in),
 .Q_data_out(Q_data_out),
 .Q_wren(Q_wren),

 .R_rdaddress(R_rdaddress),
 .R_wraddress(R_wraddress),
 .R_data_in(R_data_in),
 .R_data_out(R_data_out),
 .R_wren(R_wren),
```

```
        // Memory signals for trajectories
        .x_rdaddress(x_rdaddress),
        .x_data_out(x_data_out),

        .u_rdaddress(u_rdaddress),
        .u_data_out(u_data_out)
    );

endmodule
```

9.2) memory_interface.sv

```
None
// Optimized memory_interface using RAM blocks only
module memory_interface #(
    parameter STATE_DIM       = 12,
    parameter INPUT_DIM       = 4,
    parameter HORIZON         = 30,
    parameter EXT_DATA_WIDTH  = 32,
    parameter DATA_WIDTH      = 16,
    parameter FRAC_BITS       = 8,
    parameter ADDR_WIDTH      = 16,
    parameter MEM_ADDR_WIDTH  = 9
)(
    input  logic clk,
    input  logic rst,

    // Avalon memory-mapped slave interface
    input  logic [EXT_DATA_WIDTH-1:0] writedata,
    input  logic read,
    input  logic write,
    input  logic [ADDR_WIDTH-1:0] addr,
    input  logic chipselect,
    output logic [EXT_DATA_WIDTH-1:0] readdata,

    // Solver control
    input  logic solver_done,
    input  logic [31:0] current_iter,
    input  logic [31:0] active_horizon,
    output logic [31:0] active_horizon_new,
    output logic active_horizon_wren,
    input  logic converged,
    output logic start_solving,

    // Residuals
    input  logic [DATA_WIDTH-1:0] pri_res_u,
    input  logic [DATA_WIDTH-1:0] pri_res_x,
    input  logic [DATA_WIDTH-1:0] dual_res,
    input logic [DATA_WIDTH-1:0] pri_tol,
    input logic [DATA_WIDTH-1:0] dual_tol,
    output logic [DATA_WIDTH-1:0] pri_tol_new,
    output logic [DATA_WIDTH-1:0] dual_tol_new,
    input logic [DATA_WIDTH-1:0] rho,
    output logic [DATA_WIDTH-1:0] rho_new,
```

```
    // Bounds
    output logic [DATA_WIDTH-1:0] u_min [INPUT_DIM],
    output logic [DATA_WIDTH-1:0] u_max [INPUT_DIM],
    output logic [DATA_WIDTH-1:0] x_min [STATE_DIM],
    output logic [DATA_WIDTH-1:0] x_max [STATE_DIM],

    // Initial state and references
    output logic [STATE_DIM*DATA_WIDTH-1:0] x_init,

    output logic [STATE_DIM*HORIZON*DATA_WIDTH-1:0] x_ref,
    output logic [INPUT_DIM*(HORIZON-1)*DATA_WIDTH-1:0] u_ref,


    // RAM control ports
    output logic [MEM_ADDR_WIDTH-1:0] A_rdaddress, A_wraddress,
    output logic [DATA_WIDTH-1:0] A_data_in,
    input  logic [DATA_WIDTH-1:0] A_data_out,
    output logic A_wren,

    output logic [MEM_ADDR_WIDTH-1:0] B_rdaddress, B_wraddress,
    output logic [DATA_WIDTH-1:0] B_data_in,
    input  logic [DATA_WIDTH-1:0] B_data_out,
    output logic B_wren,

    output logic [MEM_ADDR_WIDTH-1:0] Q_rdaddress, Q_wraddress,
    output logic [DATA_WIDTH-1:0] Q_data_in,
    input  logic [DATA_WIDTH-1:0] Q_data_out,
    output logic Q_wren,

    output logic [MEM_ADDR_WIDTH-1:0] R_rdaddress, R_wraddress,
    output logic [DATA_WIDTH-1:0] R_data_in,
    input  logic [DATA_WIDTH-1:0] R_data_out,
    output logic R_wren,

    output logic [MEM_ADDR_WIDTH-1:0] x_rdaddress,
    input  logic [DATA_WIDTH-1:0] x_data_out,

    output logic [MEM_ADDR_WIDTH-1:0] u_rdaddress,
    input  logic [DATA_WIDTH-1:0] u_data_out
);

    logic [DATA_WIDTH-1:0] x_init_array [STATE_DIM];
    logic [DATA_WIDTH-1:0] x_ref_array [STATE_DIM][HORIZON];
    logic [DATA_WIDTH-1:0] u_ref_array [INPUT_DIM][HORIZON-1];
```

```
    always_comb begin
       for(int i = 0; i<STATE_DIM; i++) begin

             x_init[i*DATA_WIDTH +: DATA_WIDTH] = x_init_array[i];

       end
       for(int i = 0; i<STATE_DIM; i++) begin
          for(int j = 0; j < HORIZON; j++) begin

                x_ref[(i*HORIZON+j)*DATA_WIDTH +: DATA_WIDTH] =
x_ref_array[i][j];

           end
       end
       for(int i = 0; i<INPUT_DIM; i++) begin
          for(int j = 0; j < HORIZON-1; j++) begin

                u_ref[(i*(HORIZON-1)+j)*DATA_WIDTH +: DATA_WIDTH] =
u_ref_array[i][j];

           end
       end
    end

    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            readdata <= 0;
            start_solving <= 0;
            A_wren <= 0;
            B_wren <= 0;
            Q_wren <= 0;
            R_wren <= 0;
            pri_tol_new  <= 16'h0019;
            dual_tol_new <= 16'h0019;
            rho_new      <= 16'h0100;
        end else if (chipselect) begin
            A_wren <= 0;
            B_wren <= 0;
            Q_wren <= 0;
            R_wren <= 0;

            if (read) begin
                case (addr[15:12])
```

```
            4'h0: begin
                case (addr[11:0])
                    12'h000: readdata <= {31'b0, solver_done};
                    12'h004: readdata <= current_iter;
                    12'h008: readdata <= active_horizon;
                    12'h00C: readdata <= converged;
                    12'h010: readdata <= pri_res_u;
                    12'h018: readdata <= pri_res_x;
                    12'h020: readdata <= dual_res;
                    default: readdata <= 0;
                endcase
            end
            4'h1: begin
                x_rdaddress <= addr[7:0] * STATE_DIM + addr[11:8];
                readdata <= x_data_out;
            end
            4'h2: begin
                u_rdaddress <= addr[7:0] * INPUT_DIM + addr[11:8];
                readdata <= u_data_out;
            end
            default: readdata <= 0;
        endcase
    end else if (write) begin
        case (addr[15:12])
            4'h0: begin
                case (addr[11:0])
                    12'h000: start_solving <= writedata[0];
                    12'h004: begin
                   active_horizon_new <= writedata;
                   active_horizon_wren <= 1;
                end
                    12'h008: pri_tol_new <= writedata[15:0];
                    12'h010: dual_tol_new <= writedata[15:0];
                    12'h018: rho_new <= writedata[15:0];
                endcase
            end
            4'h1: begin
                A_wraddress <= addr[11:8] * STATE_DIM + addr[7:4];
                A_data_in <= writedata[15:0];
                A_wren <= 1;
            end
            4'h2: begin
                B_wraddress <= addr[11:8] * INPUT_DIM + addr[7:4];
                B_data_in <= writedata[15:0];
```

```
                    B_wren <= 1;
                end
                4'h3: begin
                    Q_wraddress <= addr[11:8] * STATE_DIM + addr[7:4];
                    Q_data_in <= writedata[15:0];
                    Q_wren <= 1;
                end
                4'h4: begin
                    R_wraddress <= addr[11:8] * INPUT_DIM + addr[7:4];
                    R_data_in <= writedata[15:0];
                    R_wren <= 1;
                end
                4'h5: x_init_array[addr[11:8]] <= writedata[15:0];
                4'h6: x_ref_array[addr[11:8]][addr[7:0]] <=
writedata[15:0];
                4'h7: u_ref_array[addr[11:8]][addr[7:0]] <=
writedata[15:0];
                4'h8: begin
                    if (addr[7]) x_max[addr[11:8]] <= writedata[15:0];
                    else x_min[addr[11:8]] <= writedata[15:0];
                end
                4'h9: begin
                    if (addr[7]) u_max[addr[11:8]] <= writedata[15:0];
                    else u_min[addr[11:8]] <= writedata[15:0];
                end
            endcase
        end
    end
endmodule
```

9.3) admm.c

```C/C++
/*
 * Device driver for the ADMM FPGA Accelerator
 *
 * Adapted from vga_ball.c
 */

#include "admm.h"
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/io.h>
#include <linux/kernel.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/platform_device.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>

#define DRIVER_NAME "admm"

struct admm_dev {
  struct resource res;
  void __iomem *virtbase; // memory-mapped base address of registers
  admm_config_t config;
  admm_result_t result;
} dev;

// ******************** RUNTIME FUNCTIONS *********************

static void set_initial_state(const int16_t x_init[STATE_DIM],
                              const int16_t u_init[INPUT_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_init[i], X_INIT(dev.virtbase, i));
    dev.config.initial_state.x[i] = x_init[i];
  }

  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_init[i], U_INIT(dev.virtbase, i));
```

```
    dev.config.initial_state.u[i] = u_init[i];
  }
}

static void set_reference_trajectory(const int16_t x_ref[STATE_DIM],
                                     const int16_t u_ref[INPUT_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_ref[i], X_REF(dev.virtbase, i));
    dev.config.reference.x[i] = x_ref[i];
  }

  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_ref[i], U_REF(dev.virtbase, i));
    dev.config.reference.u[i] = u_ref[i];
  }
}

static void read_result(admm_result_t *result) {
  // status
  result->iterations = ioread32(CUR_ITER(dev.virtbase));
  result->converged = ioread32(CONVERGED(dev.virtbase));
  result->pri_res_u = ioread32(PRI_RES_U(dev.virtbase));
  result->pri_res_x = ioread32(PRI_RES_X(dev.virtbase));
  result->dual_res = ioread32(DUAL_RES(dev.virtbase));

  // trajectory
  int i, j;
  for (i = 0; i < dev.config.horizon; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      result->trajectory[i].x[j] = ioread32(X_READ(dev.virtbase, i, j));
    }
    for (j = 0; j < INPUT_DIM; j++) {
      result->trajectory[i].u[j] = ioread32(U_READ(dev.virtbase, i, j));
    }
  }

  dev.result = *result;
}

// ******************** SETUP/CONFIG FUNCTIONS **********************

static void set_solver_params(uint32_t horizon, int16_t pri_tol,
                              int16_t dual_tol, int16_t rho) {
```

```
  iowrite32(horizon, HORIZON(dev.virtbase));
  iowrite32(pri_tol, PRI_TOL(dev.virtbase));
  iowrite32(dual_tol, DUAL_TOL(dev.virtbase));
  iowrite32(rho, RHO(dev.virtbase));

  // Update the local copy
  dev.config.horizon = horizon;
  dev.config.pri_tol = pri_tol;
  dev.config.dual_tol = dual_tol;
  dev.config.rho = rho;
}

static void set_state_bounds(const int16_t x_min[STATE_DIM],
                             const int16_t x_max[STATE_DIM]) {
  int i;
  for (i = 0; i < STATE_DIM; i++) {
    iowrite32(x_min[i], X_MIN(dev.virtbase, i));
    iowrite32(x_max[i], X_MAX(dev.virtbase, i));
  }
}

static void set_input_bounds(const int16_t u_min[INPUT_DIM],
                             const int16_t u_max[INPUT_DIM]) {
  int i;
  for (i = 0; i < INPUT_DIM; i++) {
    iowrite32(u_min[i], U_MIN(dev.virtbase, i));
    iowrite32(u_max[i], U_MAX(dev.virtbase, i));
  }
}

static void set_system_matrix_A(const int16_t A[STATE_DIM][STATE_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      iowrite32(A[i][j], MATRIX_A(dev.virtbase, i, j));
    }
  }
}

static void set_system_matrix_B(const int16_t B[STATE_DIM][INPUT_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < INPUT_DIM; j++) {
      iowrite32(B[i][j], MATRIX_B(dev.virtbase, i, j));
```

```
    }
  }
}

static void set_cost_matrix_Q(const int16_t Q[STATE_DIM][STATE_DIM]) {
  int i, j;
  for (i = 0; i < STATE_DIM; i++) {
    for (j = 0; j < STATE_DIM; j++) {
      iowrite32(Q[i][j], MATRIX_Q(dev.virtbase, i, j));
    }
  }
}

static void set_cost_matrix_R(const int16_t R[INPUT_DIM][INPUT_DIM]) {
  int i, j;
  for (i = 0; i < INPUT_DIM; i++) {
    for (j = 0; j < INPUT_DIM; j++) {
      iowrite32(R[i][j], MATRIX_R(dev.virtbase, i, j));
    }
  }
}

static void write_config(admm_config_t *config) {
  set_solver_params(config->horizon, config->pri_tol, config->dual_tol,
                    config->rho);
  set_initial_state(config->initial_state.x, config->initial_state.u);
  set_reference_trajectory(config->reference.x, config->reference.u);

  dev.config = *config;
}

// ******************** IOCTL **********************

// Handle ioctl() calls from userspace
static long admm_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
  admm_config_arg_t config_arg;
  admm_result_arg_t result_arg;

  switch (cmd) {
  case ADMM_WRITE_CONFIG:
    if (copy_from_user(&config_arg, (admm_config_arg_t *)arg,
                       sizeof(admm_config_arg_t)))
      return -EACCES;
    write_config(&config_arg.config);
```

```
    break;

  case ADMM_READ_RESULT:
    read_result(&dev.result);
    result_arg.result = dev.result;
    if (copy_to_user((admm_result_arg_t *)arg, &result_arg,
                     sizeof(admm_result_arg_t)))
      return -EACCES;
    break;

  case ADMM_START:
    iowrite32(1, START_SOLVER(dev.virtbase));
    break;

  default:
    return -EINVAL;
  }

  return 0;
}

/* The operations our device knows how to do */
static const struct file_operations admm_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = admm_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice admm_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &admm_fops,
};

static int __init admm_probe(struct platform_device *pdev) {
  int ret;

  /* Register ourselves as a misc device: creates /dev/admm */
  ret = misc_register(&admm_misc_device);

  /* Get the address of our registers from the device tree */
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
    ret = -ENOENT;
```

```
    goto out_deregister;
  }

  /* Make sure we can use these registers */
  if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME)
==
      NULL) {
    ret = -EBUSY;
    goto out_deregister;
  }

  /* Arrange access to our registers */
  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
  }

  /* Setup default configuration */
  memset(&dev.config, 0, sizeof(dev.config));

  /* Set default solver parameters */
  dev.config.horizon = 10;
  dev.config.pri_tol = TO_FIXED(0.01);
  dev.config.dual_tol = TO_FIXED(0.01);
  dev.config.rho = TO_FIXED(1.0);

  return 0;

out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&admm_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int admm_remove(struct platform_device *pdev) {
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&admm_misc_device);
  return 0;
}
```

```c
/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id admm_of_match[] = {
    {.compatible = "csee4840,admm-1.0"},
    {},
};
MODULE_DEVICE_TABLE(of, admm_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver admm_driver = {
    .driver =
        {
            .name = DRIVER_NAME,
            .owner = THIS_MODULE,
            .of_match_table = of_match_ptr(admm_of_match),
        },
    .remove = __exit_p(admm_remove),
};

/* Called when the module is loaded: set things up */
static int __init admm_init(void) {
  pr_info(DRIVER_NAME ": init\n");
  return platform_driver_probe(&admm_driver, admm_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit admm_exit(void) {
  platform_driver_unregister(&admm_driver);
  pr_info(DRIVER_NAME ": exit\n");
}

module_init(admm_init);
module_exit(admm_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alex Du");
MODULE_DESCRIPTION("Driver for FPGA-MPC's ADMM Accelerator");
```

9.4) admm.h

```C/C++
#ifndef _ADMM_H
#define _ADMM_H

#include <linux/ioctl.h>

#ifdef __KERNEL__
#include <linux/kernel.h>
#include <linux/types.h>
#else
#include <stdint.h>
#endif

// System dimensions
#define STATE_DIM 12
#define INPUT_DIM 4
#define MAX_HORIZON 30

typedef struct {
  int16_t x[STATE_DIM];
  int16_t u[INPUT_DIM];
} state_input_t;

typedef struct {
  int16_t x_min[STATE_DIM];
  int16_t x_max[STATE_DIM];
} state_bounds_t;

typedef struct {
  int16_t u_min[INPUT_DIM];
  int16_t u_max[INPUT_DIM];
} input_bounds_t;

typedef struct {
  int16_t A[STATE_DIM][STATE_DIM];  // State transition matrix
} system_matrix_a_t;
typedef struct {
  int16_t B[STATE_DIM][INPUT_DIM];  // Input matrix
} system_matrix_b_t;

typedef struct {
  int16_t Q[STATE_DIM][STATE_DIM];  // State cost matrix
} cost_matrix_q_t;
```

```
typedef struct {
  int16_t R[INPUT_DIM][INPUT_DIM];  // Input cost matrix
} cost_matrix_r_t;

typedef struct {
  uint32_t horizon;
  int16_t pri_tol;
  int16_t dual_tol;
  int16_t rho;
} solver_params_t;

// Fixed-point configuration
#define DATA_WIDTH 16
#define FRAC_BITS 8

typedef struct {
    int16_t x[STATE_DIM];  // State vector
    int16_t u[INPUT_DIM];   // Input vector
} admm_trajectory_point_t;

// Configuration structure
typedef struct {
    uint32_t horizon;
    int16_t pri_tol;
    int16_t dual_tol;
    int16_t rho;
    state_input_t initial_state;
    state_input_t reference;
} admm_config_t;

// Solver results
typedef struct {
    admm_trajectory_point_t trajectory[MAX_HORIZON];  // Optimal trajectory
    uint32_t iterations;    // Number of iterations
    int16_t pri_res_u;      // Primal residual (inputs)
    int16_t pri_res_x;      // Primal residual (states)
    int16_t dual_res;       // Dual residual
    uint8_t converged;      // Converged flag
} admm_result_t;

typedef struct {
    admm_config_t config;
} admm_config_arg_t;
```

```
typedef struct {
    admm_result_t result;
} admm_result_arg_t;

// IOCTL commands
#define ADMM_MAGIC 'q'

// IOCTls
#define ADMM_WRITE_CONFIG _IOW(ADMM_MAGIC, 1, admm_config_arg_t)
#define ADMM_READ_RESULT _IOR(ADMM_MAGIC, 2, admm_result_arg_t)
#define ADMM_START       _IO(ADMM_MAGIC, 3)

// ************ OFFSETS ************
#define STATUS_REG_BASE       0x0000
#define START_SOLVER(x)       ((x) + 0x00)
#define SOLVER_DONE(x)        ((x) + 0x04)
#define CUR_ITER(x)           ((x) + 0x08)
#define CONVERGED(x)          ((x) + 0x0C)

#define CONFIG_REG_BASE       0x0100
#define HORIZON(x)            ((x) + 0x00)
#define PRI_TOL(x)            ((x) + 0x04)
#define DUAL_TOL(x)           ((x) + 0x08)
#define RHO(x)                ((x) + 0x0C)

#define RESIDUAL_REG_BASE     0x0200
#define PRI_RES_U(x)          ((x) + 0x00)
#define PRI_RES_X(x)          ((x) + 0x04)
#define DUAL_RES(x)           ((x) + 0x08)

#define TRAJECTORY_BASE       0x1000
#define X_READ(x, i, j)       ((x) + 0x0000 + (i * STATE_DIM + j) *
sizeof(int16_t))
#define U_READ(x, i, j)       ((x) + 0x1000 + (i * INPUT_DIM + j) *
sizeof(int16_t))
#define X_INIT(x, i)          ((x) + 0x2000 + (i * sizeof(int16_t)))
#define U_INIT(x, i)          ((x) + 0x2100 + (i * sizeof(int16_t)))
#define X_REF(x, i)           ((x) + 0x2200 + (i * sizeof(int16_t)))
#define U_REF(x, i)           ((x) + 0x2300 + (i * sizeof(int16_t)))

#define MATRIX_A(x, i, j)      ((x) + 0x3000 + (i << 8) + (j << 4))
#define MATRIX_B(x, i, j)      ((x) + 0x4000 + (i << 8) + (j << 4))

#define MATRIX_Q(x, i, j)      ((x) + 0x5000 + (i << 8) + (j << 4))
```

```
#define MATRIX_R(x, i, j)      ((x) + 0x6000 + (i << 8) + (j << 4))

#define X_MIN(x, i)            ((x) + 0x7000 + (i << 4))
#define X_MAX(x, i)            ((x) + 0x7080 + (i << 4))
#define U_MIN(x, i)            ((x) + 0x8000 + (i << 4))
#define U_MAX(x, i)            ((x) + 0x8080 + (i << 4))

// Fixed-point conversion
#define TO_FIXED(x)   ((int16_t)((x) * (1 << FRAC_BITS)))
#define FROM_FIXED(x) (((float)(x)) / (1 << FRAC_BITS))

#endif // _ADMM_H
```

9.5) udp_interface.c

```C/C++
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <errno.h>

#define SIMULATOR_PORT 12345
#define MPC_PORT 12346
#define BUFFER_SIZE 1024

// Structure to hold the quadrotor state
typedef struct {
    float position[3];    // x, y, z
    float quaternion[4];  // w, x, y, z
    float velocity[3];    // vx, vy, vz
    float angular_vel[3]; // wx, wy, wz
} QuadState;

// Structure to hold control inputs
typedef struct {
    float motor_cmd[4];  // Motor commands
} ControlInput;

// Function prototypes
int setup_udp_socket(int port, struct sockaddr_in *addr, const char *ip);
int receive_state(int sock_fd, struct sockaddr_in *server_addr, QuadState
*state);
int send_control(int sock_fd, struct sockaddr_in *server_addr, const
ControlInput *control);
void compute_mpc_control(const QuadState *state, ControlInput *control);

int main(int argc, char *argv[]) {
    // Default simulator IP
    char *simulator_ip = "127.0.0.1";

    // Parse command line arguments
    if (argc > 1) {
        simulator_ip = argv[1];
```

```
}

// Setup socket and address structures
int sock_fd;
struct sockaddr_in server_addr, client_addr;

// Initialize UDP socket
sock_fd = setup_udp_socket(MPC_PORT, &client_addr, "0.0.0.0");
if (sock_fd < 0) {
    fprintf(stderr, "Failed to setup UDP socket\n");
    return 1;
}

// Setup server address
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SIMULATOR_PORT);

if (inet_pton(AF_INET, simulator_ip, &server_addr.sin_addr) <= 0) {
    fprintf(stderr, "Invalid simulator IP address\n");
    close(sock_fd);
    return 1;
}

printf("UDP Interface initialized - connecting to simulator at %s:%d\n",
       simulator_ip, SIMULATOR_PORT);

// Initialize state and control structures
QuadState state;
ControlInput control;

// Main control loop
while (1) {
    // Receive current state from simulator
    if (receive_state(sock_fd, &server_addr, &state) > 0) {
        // Print received state for debugging
        printf("Position: [%.2f, %.2f, %.2f]\n",
               state.position[0], state.position[1], state.position[2]);

        // Compute control input using MPC
        compute_mpc_control(&state, &control);

        // Send control back to simulator
        if (send_control(sock_fd, &server_addr, &control) < 0) {
```

```
                fprintf(stderr, "Failed to send control\n");
            } else {
                printf("Sent control: [%.2f, %.2f, %.2f, %.2f]\n",
                       control.motor_cmd[0], control.motor_cmd[1],
                       control.motor_cmd[2], control.motor_cmd[3]);
            }
        }

        // Short sleep to prevent tight loop
        usleep(10000); // 10ms
    }

    close(sock_fd);
    return 0;
}

int setup_udp_socket(int port, struct sockaddr_in *addr, const char *ip) {
    int sock_fd;

    // Create UDP socket
    if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        return -1;
    }

    // Set socket to non-blocking
    int flags = fcntl(sock_fd, F_GETFL, 0);
    fcntl(sock_fd, F_SETFL, flags | O_NONBLOCK);

    // Configure address structure
    memset(addr, 0, sizeof(*addr));
    addr->sin_family = AF_INET;
    addr->sin_port = htons(port);

    if (inet_pton(AF_INET, ip, &addr->sin_addr) <= 0) {
        perror("Invalid address");
        close(sock_fd);
        return -1;
    }

    // Bind socket to the specified port
    if (bind(sock_fd, (const struct sockaddr *)addr, sizeof(*addr)) < 0) {
        perror("bind failed");
        close(sock_fd);
```

```
        return -1;
    }

    return sock_fd;
}

int receive_state(int sock_fd, struct sockaddr_in *server_addr, QuadState
*state) {
    char buffer[BUFFER_SIZE];
    socklen_t len = sizeof(*server_addr);

    // Receive data from the socket
    int n = recvfrom(sock_fd, buffer, BUFFER_SIZE, 0,
                     (struct sockaddr *)server_addr, &len);

    if (n < 0) {
        if (errno != EAGAIN && errno != EWOULDBLOCK) {
            perror("recvfrom failed");
        }
        return -1;
    }

    // Expected size: 12 floats (3 position + 4 quaternion + 3 velocity + 3
angular velocity)
    if (n != 12 * sizeof(float)) {
        fprintf(stderr, "Received invalid data size: %d bytes\n", n);
        return -1;
    }

    // Parse received data - convert from network byte order
    float *float_data = (float *)buffer;
    for (int i = 0; i < 3; i++) {
        state->position[i] = ntohf(float_data[i]);
    }

    for (int i = 0; i < 4; i++) {
        state->quaternion[i] = ntohf(float_data[3 + i]);
    }

    for (int i = 0; i < 3; i++) {
        state->velocity[i] = ntohf(float_data[7 + i]);
    }

    for (int i = 0; i < 3; i++) {
```

```
        state->angular_vel[i] = ntohf(float_data[10 + i]);
    }

    return n;
}

int send_control(int sock_fd, struct sockaddr_in *server_addr, const
ControlInput *control) {
    // Prepare buffer for sending
    float buffer[4];

    // Convert to network byte order
    for (int i = 0; i < 4; i++) {
        buffer[i] = htonf(control->motor_cmd[i]);
    }

    // Send data to server
    int n = sendto(sock_fd, buffer, sizeof(buffer), 0,
                   (struct sockaddr *)server_addr, sizeof(*server_addr));

    if (n < 0) {
        perror("sendto failed");
        return -1;
    }

    return n;
}

// Utility functions for float network byte order conversion
// Note: These are not standard functions and would need to be implemented
float ntohf(float netf) {
    uint32_t netl;
    memcpy(&netl, &netf, sizeof(netl));
    uint32_t hostl = ntohl(netl);
    float hostf;
    memcpy(&hostf, &hostl, sizeof(hostf));
    return hostf;
}

float htonf(float hostf) {
    uint32_t hostl;
    memcpy(&hostl, &hostf, sizeof(hostl));
    uint32_t netl = htonl(hostl);
    float netf;
```

```
    memcpy(&netf, &netl, sizeof(netf));
    return netf;
}

// Example implementation - in real use, this would call your MPC solver
void compute_mpc_control(const QuadState *state, ControlInput *control) {
    // Example hover controller using constants from the Python implementation
    const float mass = 0.035;
    const float g = 9.81;
    const float scale = 65535;
    const float kt = 2.245365e-6 * scale;

    // Calculate hover thrust
    float hover_thrust = (mass * g / kt / 4.0);

    // Set hover thrust for all motors
    for (int i = 0; i < 4; i++) {
        control->motor_cmd[i] = hover_thrust;
    }

    // In a real implementation, this would call your hardware MPC solver
    // This might involve:
    // 1. Converting state to the format needed by your MPC solver
    // 2. Calling the solver (e.g., admm.c functions)
    // 3. Converting the output to motor commands
}
```

9.6) simulator_server.py:

```python
#!/usr/bin/env python3
# simulator_server.py
import socket
import struct
import time
import numpy as np
import argparse
import os
from quadrotor import QuadrotorDynamics
from visualizer_urdf import URDFQuadrotorVisualizer

class QuadrotorSimulatorServer:
    def __init__(self, host='0.0.0.0', port=12345, client_host='127.0.0.1',
client_port=12346, simulator_hz=50.0, visualization=True,
urdf_path="drone.urdf", mesh_dir="/home/alex/a2r/fpga-mpc/main-repo/python"):
        self.host = host
        self.port = port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind((self.host, self.port))


        if client_host and client_port:
            self.client_address = (client_host, client_port)
            print(f"Client address set to {client_host}:{client_port}")
        else:
            self.client_address = None


        self.quad = QuadrotorDynamics()
        self.simulator_hz = simulator_hz
        self.dt = 1.0 / simulator_hz


        self.x = 0.00001 * np.ones(13)
        self.x[0] = 1.0


        self.uhover = self.quad.hover_thrust


        self.visualization = visualization
        if self.visualization:
```

```
            self.visualizer = URDFQuadrotorVisualizer(urdf_path=urdf_path,
mesh_dir=mesh_dir, auto_open_browser=True)
            print("3D visualization started - browser window should open
automatically")
            if urdf_path:
                print(f"Using URDF model from: {urdf_path}")
            else:
                print("Using default quadrotor model")

        print(f"Simulator server initialized on {host}:{port}")
        print(f"Running at {simulator_hz} Hz (dt = {self.dt})")

    def send_state(self):

        if self.client_address is None:
            return


        fmt = '!12f'


        q_norm = self.x[4:8] / np.linalg.norm(self.x[4:8])


        state_to_send = np.concatenate([
            self.x[0:3],
            q_norm,
            self.x[7:10],
            self.x[10:12]
        ])


        binary_data = struct.pack(fmt, *state_to_send)
        self.socket.sendto(binary_data, self.client_address)

    def receive_control(self, timeout=0.01):

        self.socket.settimeout(timeout)
        try:

            fmt = '!4f'
            expected_size = struct.calcsize(fmt)
```

```
            data, addr = self.socket.recvfrom(expected_size)
            self.client_address = addr

            if len(data) == expected_size:

                controls = struct.unpack(fmt, data)
                return np.array(controls)
            else:
                print(f"Warning: Received packet of unexpected size
{len(data)}")
                return None

        except socket.timeout:
            return None
        except Exception as e:
            print(f"Error receiving control: {e}")
            return None

    def step_simulation(self, u):


        self.x = self.quad.dynamics_rk4(self.x, u, dt=self.dt)

        if self.visualization:

            position = self.x[0:3]
            quaternion = self.x[3:7] / np.linalg.norm(self.x[3:7])


            self.visualizer.update_quadrotor_state(position, quaternion)

    def run(self):

        print("Starting simulator server. Press Ctrl+C to stop.")


        if self.client_address is not None:
            print(f"Sending initial state to {self.client_address}")
            self.send_state()

        last_step_time = time.time()

        try:
            while True:
```

```
                current_time = time.time()
                elapsed = current_time - last_step_time

                if elapsed < self.dt:
                    time.sleep(self.dt - elapsed)

                u = self.receive_control()

                if u is None:
                    u = self.uhover

                self.step_simulation(u)

                self.send_state()

                last_step_time = time.time()

        except KeyboardInterrupt:
            print("\nSimulator server stopped.")
        finally:
            self.socket.close()

if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="Quadrotor Simulator Server")
    parser.add_argument("--host", type=str, default="0.0.0.0",
                        help="Host to bind the server to")
    parser.add_argument("--port", type=int, default=12345,
                        help="Port to bind the server to")
    parser.add_argument("--client-host", type=str, default="127.0.0.1",
                        help="Client host address")
    parser.add_argument("--client-port", type=int, default=12346,
                        help="Client port")
    parser.add_argument("--hz", type=float, default=50.0,
                        help="Simulation frequency in Hz")
    parser.add_argument("--no-viz", action="store_true",
                        help="Disable visualization")
```

```
    parser.add_argument("--urdf", type=str, default=None,
                        help="Path to URDF file for visualization")
    parser.add_argument("--mesh-dir", type=str, default=None,
                        help="Path to directory containing mesh files")

    args = parser.parse_args()

    server = QuadrotorSimulatorServer(
        host=args.host,
        port=args.port,
        client_host=args.client_host,
        client_port=args.client_port,
        simulator_hz=args.hz,
        visualization=not args.no_viz
    )

    server.run()
```

9.7) hardware_mpc_client.py:

```python
#!/usr/bin/env python3
# hardware_mpc_client.py
import socket
import struct
import time
import numpy as np
import argparse

class HardwareMPCClient:
    def __init__(self, server_host='127.0.0.1', server_port=12345,
client_port=12346):


        self.server_address = (server_host, server_port)


        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind(('0.0.0.0', client_port))

        print(f"HardwareMPC client initialized, connecting to
{server_host}:{server_port}")
        print(f"Listening for responses on port {client_port}")

    def receive_state(self, timeout=1.0):

        self.socket.settimeout(timeout)
        try:

            fmt = '!12f'
            expected_size = struct.calcsize(fmt)

            data, addr = self.socket.recvfrom(expected_size)

            if len(data) == expected_size:

                state_tuple = struct.unpack(fmt, data)


                state = np.array(state_tuple)
                return state
            else:
```

```
                print(f"Warning: Received packet of unexpected size
{len(data)}")
                return None

        except socket.timeout:
            print("Timeout waiting for state from simulator")
            return None
        except Exception as e:
            print(f"Error receiving state: {e}")
            return None

    def send_control(self, control):

        if len(control) != 4:
            raise ValueError("Control must have exactly 4 elements")


        fmt = '!4f'
        binary_data = struct.pack(fmt, *control)


        self.socket.sendto(binary_data, self.server_address)

    def close(self):

        self.socket.close()

def example_hover_controller(state):

    mass = 0.035
    g = 9.81
    scale = 65535
    kt = 2.245365e-6 * scale


    hover_thrust = (mass * g / kt / 4.0) * np.ones(4)


    return hover_thrust

def main():
    parser = argparse.ArgumentParser(description="Hardware MPC Client")
    parser.add_argument("--server", type=str, default="127.0.0.1",
                        help="Simulator server IP address")
```

```
    parser.add_argument("--server-port", type=int, default=12345,
                        help="Simulator server port")
    parser.add_argument("--client-port", type=int, default=12346,
                        help="Local port to bind for receiving responses")
    args = parser.parse_args()

    client = HardwareMPCClient(
        server_host=args.server,
        server_port=args.server_port,
        client_port=args.client_port
    )

    try:
        while True:
            state = client.receive_state()

            if state is not None:

                position = state[0:3]
                quaternion = state[3:7]
                velocity = state[7:10]
                angular_velocity = state[10:13]

                print(f"Position: {position}")
                print(f"Quaternion: {quaternion}")
                print(f"Velocity: {velocity}")
                print(f"Angular Velocity: {angular_velocity}")


                u = example_hover_controller(state)
                print(f"Sending control: {u}")


                client.send_control(u)

                time.sleep(0.01)

    except KeyboardInterrupt:
        print("\nClient stopped.")
    finally:
        client.close()

if __name__ == "__main__":
    main()
```

9.8) primal_update.sv

```
None
// Riccati backward pass and forward rollout for X-update step using RAM
// Based on TinyMPC algorithm

module primal_update #(
    parameter STATE_DIM = 12,
    parameter INPUT_DIM = 4,
    parameter HORIZON = 30,
    parameter DATA_WIDTH = 16,
    parameter FRAC_BITS = 8,
    parameter ADDR_WIDTH = 9
)(
    input logic clk,
    input logic rst,
    input logic start,

    // RAM ports for all matrices and vectors
    output logic [ADDR_WIDTH-1:0] A_rdaddress,
    input  logic [DATA_WIDTH-1:0] A_data_out,

    output logic [ADDR_WIDTH-1:0] B_rdaddress,
    input  logic [DATA_WIDTH-1:0] B_data_out,

    output logic [ADDR_WIDTH-1:0] K_rdaddress,
    input  logic [DATA_WIDTH-1:0] K_data_out,

    output logic [ADDR_WIDTH-1:0] C1_rdaddress,
    input  logic [DATA_WIDTH-1:0] C1_data_out,

    output logic [ADDR_WIDTH-1:0] C2_rdaddress,
    input  logic [DATA_WIDTH-1:0] C2_data_out,

    output logic [ADDR_WIDTH-1:0] q_rdaddress,
    input  logic [DATA_WIDTH-1:0] q_data_out,

    output logic [ADDR_WIDTH-1:0] r_rdaddress,
    input  logic [DATA_WIDTH-1:0] r_data_out,

    output logic [ADDR_WIDTH-1:0] p_rdaddress,
    input  logic [DATA_WIDTH-1:0] p_data_out,

    output logic [ADDR_WIDTH-1:0] d_rdaddress,
    input  logic [DATA_WIDTH-1:0] d_data_out,
```

```
    output logic [ADDR_WIDTH-1:0] x_rdaddress,
    input  logic [DATA_WIDTH-1:0] x_data_out,

    output logic [ADDR_WIDTH-1:0] u_rdaddress,
    input  logic [DATA_WIDTH-1:0] u_data_out,

    output logic [ADDR_WIDTH-1:0] x_wraddress,
    output logic [DATA_WIDTH-1:0] x_data_in,
    output logic x_wren,

    output logic [ADDR_WIDTH-1:0] u_wraddress,
    output logic [DATA_WIDTH-1:0] u_data_in,
    output logic u_wren,

    output logic [ADDR_WIDTH-1:0] d_wraddress,
    output logic [DATA_WIDTH-1:0] d_data_in,
    output logic d_wren,

    output logic [ADDR_WIDTH-1:0] p_wraddress,
    output logic [DATA_WIDTH-1:0] p_data_in,
    output logic p_wren,

    input logic [31:0] active_horizon,
    input logic [STATE_DIM*DATA_WIDTH-1:0] x_init, //modify this 2d
    output logic done
);

    // FSM state definitions
    localparam IDLE         = 3'd0;
    localparam INIT         = 3'd1;
    localparam BACKWARD     = 3'd2;
    localparam FORWARD      = 3'd3;
    localparam DONE_STATE   = 3'd4;

    localparam FP_COMPUTE_X = 2'd0;
    localparam FP_STORE     = 2'd1;

    logic [2:0] state;
    logic [1:0] substate;
    logic [2:0] fp_state;
    logic [31:0] i, j, k;
    logic [DATA_WIDTH-1:0] temp_sum;
    logic [31:0] cycle_counter;
```

```
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            substate <= 0;
            fp_state <= 0;
            x_wren <= 0;
            u_wren <= 0;
            d_wren <= 0;
            p_wren <= 0;
            done <= 0;
            i <= 0;
            j <= 0;
            k <= 0;
            temp_sum <= 0;
            cycle_counter <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (start) begin
                        i <= 0;
                        state <= INIT;
                    end
                end

                INIT: begin
                    if (i < STATE_DIM) begin
                        x_wraddress <= i;
                        x_data_in <= x_init[i*DATA_WIDTH +: DATA_WIDTH];
//change this 2d
                        x_wren <= 1;
                        i <= i + 1;
                    end else begin
                        x_wren <= 0;
                        i <= 0;
                        k <= 0;
                        state <= FORWARD;
                        substate <= FP_COMPUTE_X;
                    end
                end

                FORWARD: begin
                    case (substate)
                        FP_COMPUTE_X: begin
```

```
                        case (fp_state)
                            0: begin
                                temp_sum <= 0;
                                j <= 0;
                                fp_state <= 1;
                            end
                            1: begin
                                if (j < STATE_DIM) begin
                                    A_rdaddress <= i * STATE_DIM + j;
                                    x_rdaddress <= k * STATE_DIM + j;
                                    fp_state <= 2;
                                end else begin
                                    j <= 0;
                                    fp_state <= 4;
                                end
                            end
                            2: fp_state <= 3;
                            3: begin
                                temp_sum <= temp_sum + A_data_out *
x_data_out;
                                j <= j + 1;
                                fp_state <= 1;
                            end
                            4: begin
                                if (j < INPUT_DIM) begin
                                    B_rdaddress <= i * INPUT_DIM + j;
                                    u_rdaddress <= k * INPUT_DIM + j;
                                    fp_state <= 5;
                                end else begin
                                    fp_state <= 7;
                                end
                            end
                            5: fp_state <= 6;
                            6: begin
                                temp_sum <= temp_sum + B_data_out *
u_data_out;
                                j <= j + 1;
                                fp_state <= 4;
                            end
                            7: begin
                                x_wraddress <= (k+1) * STATE_DIM + i;
                                x_data_in <= temp_sum;
                                x_wren <= 1;
                                fp_state <= 8;
```

```
                                end
                                8: begin
                                    x_wren <= 0;
                                    if (i < STATE_DIM - 1) begin
                                        i <= i + 1;
                                        fp_state <= 0;
                                    end else begin
                                        i <= 0;
                                        substate <= FP_STORE;
                                        fp_state <= 0;
                                    end
                                end
                            endcase
                        end

                        FP_STORE: begin
                            // Placeholder for storing u[k] - similar to x
above
                            if (k < active_horizon - 1) begin
                                k <= k + 1;
                                substate <= FP_COMPUTE_X;
                            end else begin
                                state <= DONE_STATE;
                            end
                        end
                    endcase
                end

                DONE_STATE: begin
                    done <= 1;
                    if (!start) begin
                        state <= IDLE;
                        done <= 0;
                    end
                end
            endcase
        end
    end

endmodule
```

9.9) slack_update.sv

```
None
// Implements constraint projection for both u and x variables

`timescale 1ps/1ps
module slack_update #(
    parameter int STATE_DIM   = 12,     // Dimension of state vector (nx)
    parameter int INPUT_DIM   = 4,      // Dimension of input vector (nu)
    parameter int HORIZON     = 30,     // Maximum MPC horizon length (N)
    parameter int DATA_WIDTH  = 16,     // Fixed-point width
    parameter int ADDR_WIDTH  = 9       // Address width (enough for
HORIZON*dim)
) (
    input  logic                     clk,             // Clock
    input  logic                     rst,             // Reset (synchronous)
    input  logic                     start,           // Start signal

    // ? u and y memory interface (for z-update) ?
    output logic [ADDR_WIDTH-1:0]    u_rdaddress,
    input  logic signed [DATA_WIDTH-1:0] u_data_out,
    output logic [ADDR_WIDTH-1:0]    y_rdaddress,
    input  logic signed [DATA_WIDTH-1:0] y_data_out,

    // ? z write port ?
    output logic [ADDR_WIDTH-1:0]    z_wraddress,
    output logic signed [DATA_WIDTH-1:0] z_data_in,
    output logic                     z_wren,

    // ? x and g memory interface (for v-update) ?
    output logic [ADDR_WIDTH-1:0]    x_rdaddress,
    input  logic signed [DATA_WIDTH-1:0] x_data_out,
    output logic [ADDR_WIDTH-1:0]    g_rdaddress,
    input  logic signed [DATA_WIDTH-1:0] g_data_out,

    // ? v write port ?
    output logic [ADDR_WIDTH-1:0]    v_wraddress,
    output logic signed [DATA_WIDTH-1:0] v_data_in,
    output logic                     v_wren,

    // ? box bounds ?
    input  wire signed [DATA_WIDTH-1:0] u_min [INPUT_DIM],
    input  wire signed [DATA_WIDTH-1:0] u_max [INPUT_DIM],
    input  wire signed [DATA_WIDTH-1:0] x_min [STATE_DIM],
    input  wire signed [DATA_WIDTH-1:0] x_max [STATE_DIM],
```

```
    // ? active horizon (? HORIZON) ?
    input  logic [31:0]              active_horizon,

    output logic                     done             // High when finished
);

    // States
    typedef enum logic [1:0] {
        S_IDLE,
        S_PROJ_Z,
        S_PROJ_V,
        S_DONE
    } state_t;

    state_t state;
    logic [31:0] k, i;
    logic [2:0]  phase;
    logic signed [DATA_WIDTH-1:0] temp_u, temp_y, temp_x, temp_g, temp_val;


//------------------------------------------------------------------------------
-
    // State machine

//------------------------------------------------------------------------------
-
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state       <= S_IDLE;
            done        <= 1'b0;
            k           <= 0;
            i           <= 0;
            phase       <= 0;
            // clear ports
            u_rdaddress <= 0; y_rdaddress <= 0; z_wraddress <= 0; z_wren <= 0;
            x_rdaddress <= 0; g_rdaddress <= 0; v_wraddress <= 0; v_wren <= 0;
        end else begin
            case (state)

                S_IDLE: begin
                    done <= 1'b0;
                    if (start) begin
                        state <= S_PROJ_Z;
```

```
                    k     <= 0;
                    i     <= 0;
                    phase <= 0;
                end
            end

            //--- PROJECT Z: z = clip(u + y, u_min, u_max) ---
            S_PROJ_Z: begin
                case (phase)
                    3'd0: begin
                        // drive read addresses
                        u_rdaddress <= k*INPUT_DIM + i;
                        y_rdaddress <= k*INPUT_DIM + i;
                        phase       <= 3'd1;
                    end
                    3'd1: begin
                        // capture reads
                        temp_u <= u_data_out;
                        temp_y <= y_data_out;
                        phase  <= 3'd2;
                    end
                    3'd2: begin
                        // compute and clip
                        temp_val = temp_u + temp_y;
                        if      (temp_val < u_min[i]) z_data_in <=
u_min[i];
                        else if (temp_val > u_max[i]) z_data_in <=
u_max[i];
                        else                          z_data_in <=
temp_val;
                        z_wraddress <= k*INPUT_DIM + i;
                        z_wren      <= 1'b1;
                        phase       <= 3'd3;
                    end
                    3'd3: begin
                        // finish this element
                        z_wren <= 1'b0;
                        // advance indices
                        if (i == INPUT_DIM-1) begin
                            i <= 0;
                            if (k == active_horizon-2) begin
                                // done with all z
                                state <= S_PROJ_V;
                                k     <= 0;
```

```
                                i     <= 0;
                                phase <= 0;
                            end else begin
                                k <= k + 1;
                            end
                        end else i <= i + 1;
                        phase <= 3'd0;
                    end
                endcase
            end

            //--- PROJECT V: v = clip(x + g, x_min, x_max) ---
            S_PROJ_V: begin
                case (phase)
                    3'd0: begin
                        x_rdaddress <= k*STATE_DIM + i;
                        g_rdaddress <= k*STATE_DIM + i;
                        phase       <= 3'd1;
                    end
                    3'd1: begin
                        temp_x <= x_data_out;
                        temp_g <= g_data_out;
                        phase  <= 3'd2;
                    end
                    3'd2: begin
                        temp_val = temp_x + temp_g;
                        if      (temp_val < x_min[i]) v_data_in <=
x_min[i];
                        else if (temp_val > x_max[i]) v_data_in <=
x_max[i];
                        else                          v_data_in <=
temp_val;
                        v_wraddress <= k*STATE_DIM + i;
                        v_wren      <= 1'b1;
                        phase       <= 3'd3;
                    end
                    3'd3: begin
                        v_wren <= 1'b0;
                        if (i == STATE_DIM-1) begin
                            i <= 0;
                            if (k == active_horizon-1) begin
                                state <= S_DONE;
                            end else k <= k + 1;
                        end else i <= i + 1;
```

```
                        phase <= 3'd0;
                    end
                endcase
            end

            S_DONE: begin
                done  <= 1'b1;
                if (!start) begin
                    state <= S_IDLE;
                end
            end

            default: state <= S_IDLE;
        endcase
    end
end

endmodule
```

9.10) dual_update.sv

```
None
// y-update and g-update (dual variable update) steps

`timescale 1 ps / 1 ps
module dual_update #(
    parameter STATE_DIM = 12,             // Dimension of state vector (nx)
    parameter INPUT_DIM = 4,             // Dimension of input vector (nu)
    parameter HORIZON = 30,              // Maximum MPC horizon length (N)

    parameter DATA_WIDTH = 16,           // 16-bit fixed point
    parameter FRAC_BITS = 8,             // Number of fractional bits for fixed
point
    parameter ADDR_WIDTH = 9
)(
    input logic clk,                     // Clock
    input logic rst,                     // Reset
    input logic start,                   // Start signal

    // ADMM variables - memory interfaces for trajectories and auxiliaries
    // State trajectory (x)
    output logic [ADDR_WIDTH-1:0] x_rdaddress,
    input logic [DATA_WIDTH-1:0] x_data_out,

    // Input trajectory (u)
    output logic [ADDR_WIDTH-1:0] u_rdaddress,
    input logic [DATA_WIDTH-1:0] u_data_out,

    // Input auxiliary variables (z)
    output logic [ADDR_WIDTH-1:0] z_rdaddress,
    input logic [DATA_WIDTH-1:0] z_data_out,

    // State auxiliary variables (v)
    output logic [ADDR_WIDTH-1:0] v_rdaddress,
    input logic [DATA_WIDTH-1:0] v_data_out,

    // Dual variables - memory interfaces
    // Input dual variables (y)
    output logic [ADDR_WIDTH-1:0] y_rdaddress,
    input logic [DATA_WIDTH-1:0] y_data_out,
    output logic [ADDR_WIDTH-1:0] y_wraddress,
    output logic [DATA_WIDTH-1:0] y_data_in,
    output logic y_wren,
```

```
// State dual variables (g)
output logic [ADDR_WIDTH-1:0] g_rdaddress,
input logic [DATA_WIDTH-1:0] g_data_out,
output logic [ADDR_WIDTH-1:0] g_wraddress,
output logic [DATA_WIDTH-1:0] g_data_in,
output logic g_wren,

// Linear cost terms - memory interfaces
// Input cost terms (r)
output logic [ADDR_WIDTH-1:0] r_rdaddress,
input logic [DATA_WIDTH-1:0] r_data_out,
output logic [ADDR_WIDTH-1:0] r_wraddress,
output logic [DATA_WIDTH-1:0] r_data_in,
output logic r_wren,

// State cost terms (q)
output logic [ADDR_WIDTH-1:0] q_rdaddress,
input logic [DATA_WIDTH-1:0] q_data_out,
output logic [ADDR_WIDTH-1:0] q_wraddress,
output logic [DATA_WIDTH-1:0] q_data_in,
output logic q_wren,

// Terminal cost terms (p)
output logic [ADDR_WIDTH-1:0] p_rdaddress,
input logic [DATA_WIDTH-1:0] p_data_out,
output logic [ADDR_WIDTH-1:0] p_wraddress,
output logic [DATA_WIDTH-1:0] p_data_in,
output logic p_wren,

// Reference trajectories
input logic [STATE_DIM*HORIZON*DATA_WIDTH-1:0] x_ref,
input logic [INPUT_DIM*(HORIZON-1)*DATA_WIDTH-1:0] u_ref,


// Cost matrices
output logic [ADDR_WIDTH-1:0] R_rdaddress,
input logic [DATA_WIDTH-1:0] R_data_out,
output logic [ADDR_WIDTH-1:0] Q_rdaddress,
input logic [DATA_WIDTH-1:0] Q_data_out,
output logic [ADDR_WIDTH-1:0] P_rdaddress,
input logic [DATA_WIDTH-1:0] P_data_out,

// ADMM parameter
input logic [DATA_WIDTH-1:0] rho,
```

```
    // Residual calculation outputs
    output logic [DATA_WIDTH-1:0] pri_res_u,                            //
Primal residual for inputs
    output logic [DATA_WIDTH-1:0] pri_res_x,                            //
Primal residual for states

    // Configuration
    input logic [31:0] active_horizon,  // Current horizon length to use

    output logic done                    // Done signal
);

    // State machine states
    localparam IDLE = 3'd0;
    localparam UPDATE_Y = 3'd1;         // Update y dual variables (for inputs)
    localparam UPDATE_G = 3'd2;         // Update g dual variables (for states)
    localparam CALC_RESIDUALS = 3'd3;   // Calculate residuals
    localparam UPDATE_LINEAR_COST = 3'd4; // Update linear cost terms
    localparam DONE_STATE = 3'd5;

    // State variables
    logic [2:0] state;
    logic [31:0] i;                     // Generic counter
    logic [31:0] k;                     // Step counter for horizon
    logic [31:0] state_timer;           // State timer

    // Memory access state variables
    logic [31:0] read_stage;            // Tracks memory read sequencing
    logic [31:0] write_stage;           // Tracks memory write sequencing

    // Temporary storage for values read from memory

    logic [DATA_WIDTH-1:0] temp_u;
    logic [DATA_WIDTH-1:0] temp_z;
    logic [DATA_WIDTH-1:0] temp_x;
    logic [DATA_WIDTH-1:0] temp_v;
    logic [DATA_WIDTH-1:0] temp_y;
    logic [DATA_WIDTH-1:0] temp_g;
    logic [DATA_WIDTH-1:0] temp_r;
    logic [DATA_WIDTH-1:0] temp_q;
    logic [DATA_WIDTH-1:0] temp_p;
    logic [DATA_WIDTH-1:0] temp_R;
    logic [DATA_WIDTH-1:0] temp_Q;
```

```
    logic [DATA_WIDTH-1:0] temp_P;

    // Temporary computation variables
    logic [DATA_WIDTH-1:0] temp_val;
    logic [DATA_WIDTH-1:0] max_pri_res_u; // Maximum primal residual for inputs
    logic [DATA_WIDTH-1:0] max_pri_res_x; // Maximum primal residual for states

    logic [DATA_WIDTH-1:0] x_ref_array [STATE_DIM][HORIZON];
    logic [DATA_WIDTH-1:0] u_ref_array [INPUT_DIM][HORIZON-1];

    always_comb begin
       for(int i = 0; i<STATE_DIM; i++) begin
          for(int j = 0; j < HORIZON; j++) begin

                  x_ref_array[i][j] = x_ref[(i*HORIZON+j)*DATA_WIDTH +:
DATA_WIDTH] ;

           end
       end
       for(int i = 0; i<INPUT_DIM; i++) begin
          for(int j = 0; j < HORIZON-1; j++) begin

                   u_ref_array[i][j] = u_ref[(i*(HORIZON-1)+j)*DATA_WIDTH +:
DATA_WIDTH];

           end
       end
    end

    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            done <= 0;
            i <= 0;
            k <= 0;
            state_timer <= 0;
            read_stage <= 0;
            write_stage <= 0;
            pri_res_u <= 0;
            pri_res_x <= 0;
            max_pri_res_u <= 0;
            max_pri_res_x <= 0;

            // Initialize memory control signals
```

```
        x_rdaddress <= 0;
        u_rdaddress <= 0;
        z_rdaddress <= 0;
        v_rdaddress <= 0;
        y_rdaddress <= 0;
        g_rdaddress <= 0;
        r_rdaddress <= 0;
        q_rdaddress <= 0;
        p_rdaddress <= 0;
        R_rdaddress <= 0;
        Q_rdaddress <= 0;
        P_rdaddress <= 0;

        y_wraddress <= 0;
        y_data_in <= 0;
        y_wren <= 0;

        g_wraddress <= 0;
        g_data_in <= 0;
        g_wren <= 0;

        r_wraddress <= 0;
        r_data_in <= 0;
        r_wren <= 0;

        q_wraddress <= 0;
        q_data_in <= 0;
        q_wren <= 0;

        p_wraddress <= 0;
        p_data_in <= 0;
        p_wren <= 0;
    end else begin
       r_rdaddress <= 0;
        q_rdaddress <= 0;
        p_rdaddress <= 0;
        case (state)
            IDLE: begin
                if (start) begin
                    state <= UPDATE_Y;
                    done <= 0;
                    i <= 0;
                    k <= 0;
                    state_timer <= 0;
```

```
            read_stage <= 0;
            write_stage <= 0;
            max_pri_res_u <= 0;
            max_pri_res_x <= 0;

            // Reset all memory write enables
            y_wren <= 0;
            g_wren <= 0;
            r_wren <= 0;
            q_wren <= 0;
            p_wren <= 0;
        end
    end

    UPDATE_Y: begin
        // Update y dual variables: y += u - z
        state_timer <= state_timer + 1;

        if (state_timer == 1) begin
            // Initialize the y-update process
            read_stage <= 1;
            k <= 0;
            i <= 0;
        end

        if (read_stage == 1) begin
            // Read u, z, and y values for current element
            if (k < active_horizon-1 && i < INPUT_DIM) begin
                // Calculate current index
                int index;
                index = k*INPUT_DIM + i;

                // Memory read/write sequence
                if (state_timer % 6 == 1) begin
                    // Stage 1: Set read addresses
                    u_rdaddress <= index;
                    z_rdaddress <= index;
                    y_rdaddress <= index;
                end else if (state_timer % 6 == 2) begin
                    // Stage 2: Wait for read to complete
                end else if (state_timer % 6 == 3) begin
                    // Stage 3: Capture values
                    temp_u <= u_data_out;
                    temp_z <= z_data_out;
```

```
                temp_y <= y_data_out;
            end else if (state_timer % 6 == 4) begin
                // Stage 4: Compute y update and residual
                // y += u - z
                temp_val = temp_u - temp_z;
                y_data_in <= temp_y + temp_val;

                // Set write address
                y_wraddress <= index;
                y_wren <= 1;

                // Track maximum residual (in absolute value)
                if (temp_val < 0) begin
                    temp_val = -temp_val; // Absolute value
                end
                if (temp_val > max_pri_res_u) begin
                    max_pri_res_u <= temp_val;
                end

            end else if (state_timer % 6 == 0) begin
                // Stage 6: Advance to next element
                y_wren <= 0;

                i <= i + 1;
                if (i == INPUT_DIM-1) begin
                    i <= 0;
                    k <= k + 1;
                end
            end




        end else begin
            // Done updating all y values
            read_stage <= 0;
            y_wren <= 0;

            // Move to next state
            state <= UPDATE_G;
            k <= 0;
            i <= 0;
            state_timer <= 0;
        end
```

```
    end
end

UPDATE_G: begin
    // Update g dual variables: g += x - v
    state_timer <= state_timer + 1;

    if (state_timer == 1) begin
        // Initialize the g-update process
        read_stage <= 1;
        k <= 0;
        i <= 0;
    end

    if (read_stage == 1) begin
        // Read x, v, and g values for current element
        if (k < active_horizon && i < STATE_DIM) begin
            // Calculate current index
            int index;
            index = k*STATE_DIM + i;

            // Memory read/write sequence
            if (state_timer % 6 == 1) begin
                // Stage 1: Set read addresses
                x_rdaddress <= index;
                v_rdaddress <= index;
                g_rdaddress <= index;
            end else if (state_timer % 6 == 2) begin
                // Stage 2: Wait for read to complete
            end else if (state_timer % 6 == 3) begin
                // Stage 3: Capture values
                temp_x <= x_data_out;
                temp_v <= v_data_out;
                temp_g <= g_data_out;
            end else if (state_timer % 6 == 4) begin
                // Stage 4: Compute g update and residual
                // g += x - v
                temp_val = temp_x - temp_v;
                g_data_in <= temp_g + temp_val;

                // Set write address
                g_wraddress <= index;
                g_wren <= 1;
```

```
                    // Track maximum residual (in absolute value)
                    if (temp_val < 0) begin
                        temp_val = -temp_val; // Absolute value
                    end
                    if (temp_val > max_pri_res_x) begin
                        max_pri_res_x <= temp_val;
                    end
                end

            else if (state_timer % 6 == 0) begin
                    // Stage 6: Advance to next element
                    g_wren <= 0;

                    i <= i + 1;
                    if (i == STATE_DIM-1) begin
                        i <= 0;
                        k <= k + 1;
                    end
                end

            end else begin
                // Done updating all g values
                read_stage <= 0;
                g_wren <= 0;

                // Move to next state
                state <= CALC_RESIDUALS;
                state_timer <= 0;
            end
        end
    end

    CALC_RESIDUALS: begin
        // Store final residual values
        pri_res_u <= max_pri_res_u;
        pri_res_x <= max_pri_res_x;
        state <= UPDATE_LINEAR_COST;
        k <= 0;
        i <= 0;
        state_timer <= 0;
    end

    UPDATE_LINEAR_COST: begin
        // Update linear cost terms: r, q, and p
```

```
                    state_timer <= state_timer + 1;

                    if (state_timer == 1) begin
                        // Initialize the linear cost update process
                        read_stage <= 1;
                        k <= 0;
                        i <= 0;
                    end

                    if (read_stage == 1) begin
                        // First update r and q for k=0 to N-2
                        if (k < active_horizon-1) begin
                            // Update r vector (for inputs)
                            if (i < INPUT_DIM) begin
                                // Calculate current index
                                int index;
                               index = k*INPUT_DIM + i;

                                // Memory read/write sequence
                                if (state_timer % 8 == 1) begin
                                    // Stage 1: Set read addresses for r update
                                    R_rdaddress <= i*INPUT_DIM + i; // Diagonal
element of R

                                    z_rdaddress <= index;
                                    y_rdaddress <= index;
                                end else if (state_timer % 8 == 2) begin
                                    // Stage 2: Wait for read to complete
                                end else if (state_timer % 8 == 3) begin
                                    // Stage 3: Capture values
                                    temp_R <= R_data_out;
                                    temp_z <= z_data_out;
                                    temp_y <= y_data_out;
                                end else if (state_timer % 8 == 4) begin
                                    // Stage 4: Compute r update
                                    // r[:, k] = -R @ u_ref[:, k]
                                    // r[:, k] -= rho * (z[:, k] - y[:, k])
                                    temp_val = -temp_R * u_ref_array[i][k]; //
Simplified matrix-vector product
                                    r_data_in <= temp_val - rho * (temp_z -
temp_y);

                                    // Set write address
                                    r_wraddress <= index;
                                    r_wren <= 1;
```

```
                                end else if (state_timer % 8 == 5) begin
                                    // Stage 5: Set read addresses for q update
                                    Q_rdaddress <= i*STATE_DIM + i; // Diagonal
element of Q
                                    v_rdaddress <= k*STATE_DIM + i;
                                    g_rdaddress <= k*STATE_DIM + i;
                                end else if (state_timer % 8 == 6) begin
                                    // Stage 6: Wait for read to complete
                                    r_wren <= 0;
                                end else if (state_timer % 8 == 7) begin
                                    // Stage 7: Capture values
                                    temp_Q <= Q_data_out;
                                    temp_v <= v_data_out;
                                    temp_g <= g_data_out;
                                end else begin
                                    // Stage 8: Compute q update
                                    // q[:, k] = -Q @ x_ref[:, k]
                                    // q[:, k] -= rho * (v[:, k] - g[:, k])
                                    temp_val = -temp_Q * x_ref_array[i][k]; //
Simplified matrix-vector product
                                    q_data_in <= temp_val - rho * (temp_v -
temp_g);

                                    // Set write address
                                    q_wraddress <= k*STATE_DIM + i;
                                    q_wren <= 1;

                                    // Advance to next element
                                    i <= i + 1;
                                    if (i == STATE_DIM-1) begin
                                        i <= 0;
                                        k <= k + 1;
                                    end
                                    q_wren <= 0;
                                end
                            end
                        end else if (k == active_horizon-1) begin
                            // Update p vector for the terminal state (k=N-1)
                            if (i < STATE_DIM) begin
                                // Calculate current index
                                int index;
                               index = k*STATE_DIM + i;

                                // Memory read/write sequence
```

```
                                if (state_timer % 6 == 1) begin
                                    // Stage 1: Set read addresses
                                    P_rdaddress <= i*STATE_DIM + i; // Diagonal
element of P
                                    v_rdaddress <= index;
                                    g_rdaddress <= index;
                                end else if (state_timer % 6 == 2) begin
                                    // Stage 2: Wait for read to complete
                                end else if (state_timer % 6 == 3) begin
                                    // Stage 3: Capture values
                                    temp_P <= P_data_out;
                                    temp_v <= v_data_out;
                                    temp_g <= g_data_out;
                                end else if (state_timer % 6 == 4) begin
                                    // Stage 4: Compute p update
                                    // p[:,N-1] = -P @ x_ref[:, N-1]
                                    // p[:,N-1] -= rho * (v[:, N-1] - g[:,
N-1])
                                    temp_val = -temp_P * x_ref_array[i][k]; //
Simplified matrix-vector product
                                    p_data_in <= temp_val - rho * (temp_v -
temp_g);

                                    // Set write address
                                    p_wraddress <= index;
                                    p_wren <= 1;
                                end else if (state_timer % 6 == 0) begin
                                    // Stage 6: Advance to next element
                                    p_wren <= 0;

                                    i <= i + 1;
                                    if (i == STATE_DIM-1) begin
                                        // Done with all updates
                                        read_stage <= 0;
                                        state <= DONE_STATE;
                                    end
                                end
                            end
                        end else begin
                            // Done with all updates
                            read_stage <= 0;
                            r_wren <= 0;
                            q_wren <= 0;
                            p_wren <= 0;
```

```
                        state <= DONE_STATE;
                    end
                end
            end

            DONE_STATE: begin
                done <= 1;
                // Ensure all write enables are off
                y_wren <= 0;
                g_wren <= 0;
                r_wren <= 0;
                q_wren <= 0;
                p_wren <= 0;

                if (!start) begin
                    state <= IDLE;
                    done <= 0;
                end
            end

            default: state <= IDLE;
        endcase
    end
end

endmodule
```

9.11) cost_update.sv

```
None
`timescale 1ns/1ps

module cost_update #(
    parameter STATE_DIM   = 12,             // Dimension of state vector (nx)
    parameter INPUT_DIM   = 4,              // Dimension of input vector (nu)
    parameter HORIZON     = 30,             // Maximum MPC horizon length (N)
    parameter DATA_WIDTH  = 16,             // 16-bit fixed point
    parameter FRAC_BITS   = 8,              // Number of fractional bits for
fixed point
    parameter ADDR_WIDTH  = 9               // Address width for memory access
)(
    input  logic                          clk,
    input  logic                          rst,
    input  logic                          start,

    // State and input trajectories
    output logic [ADDR_WIDTH-1:0]         x_rdaddress,
    input  logic [DATA_WIDTH-1:0]         x_data_out,
    output logic [ADDR_WIDTH-1:0]         u_rdaddress,
    input  logic [DATA_WIDTH-1:0]         u_data_out,

    // Reference trajectories
    input  wire [DATA_WIDTH-1:0]        x_ref [STATE_DIM][HORIZON],
    input  wire [DATA_WIDTH-1:0]        u_ref [INPUT_DIM][HORIZON-1],

    // Cost matrices
    output logic [ADDR_WIDTH-1:0]         R_rdaddress,
    input  logic [DATA_WIDTH-1:0]         R_data_out,
    output logic [ADDR_WIDTH-1:0]         Q_rdaddress,
    input  logic [DATA_WIDTH-1:0]         Q_data_out,
    output logic [ADDR_WIDTH-1:0]         P_rdaddress,
    input  logic [DATA_WIDTH-1:0]         P_data_out,

    // Regularization parameter
    input  logic [DATA_WIDTH-1:0]         rho,

    // Active horizon length
    input  logic [31:0]                   active_horizon,

    // Residual memory write interfaces
    output logic [ADDR_WIDTH-1:0]         r_wraddress,
    output logic [DATA_WIDTH-1:0]         r_data_in,
```

```
    output logic                          r_wren,
    output logic [ADDR_WIDTH-1:0]         q_wraddress,
    output logic [DATA_WIDTH-1:0]         q_data_in,
    output logic                          q_wren,
    output logic [ADDR_WIDTH-1:0]         p_wraddress,
    output logic [DATA_WIDTH-1:0]         p_data_in,
    output logic                          p_wren,

    // Residual calculation outputs
    output logic [DATA_WIDTH-1:0]         pri_res_u,
    output logic [DATA_WIDTH-1:0]         pri_res_x,

    // Done signal
    output logic                          done
);

    // State machine states
    localparam IDLE           = 3'd0;
    localparam UPDATE_R       = 3'd1;
    localparam UPDATE_Q       = 3'd2;
    localparam UPDATE_P       = 3'd3;
    localparam CALC_RESIDUALS = 3'd4;
    localparam DONE_STATE     = 3'd5;

    // State variables
    logic [2:0]   state;
    logic [31:0]  k, i, state_timer;
    logic [31:0]  read_stage, write_stage;

    // Temp variables for computation
    logic [DATA_WIDTH-1:0] temp_r, temp_q, temp_p;
    logic [DATA_WIDTH-1:0] temp_u, temp_x;
    logic [DATA_WIDTH-1:0] max_pri_res_u, max_pri_res_x;

    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            // reset state
            state         <= IDLE;
            done          <= 1'b0;
            i             <= 0;
            k             <= 0;
            state_timer   <= 0;
            read_stage    <= 0;
            write_stage   <= 0;
```

```
        pri_res_u     <= 0;
        pri_res_x     <= 0;
        max_pri_res_u <= 0;
        max_pri_res_x <= 0;

        // initialize read addresses
        x_rdaddress <= 0;
        u_rdaddress <= 0;
        R_rdaddress <= 0;
        Q_rdaddress <= 0;
        P_rdaddress <= 0;

        // initialize residual write interfaces
        r_wraddress <= 0;
        q_wraddress <= 0;
        p_wraddress <= 0;
        r_data_in   <= 0;
        q_data_in   <= 0;
        p_data_in   <= 0;
        r_wren      <= 1'b0;
        q_wren      <= 1'b0;
        p_wren      <= 1'b0;
    end else begin
        case (state)
            IDLE: begin
                if (start) begin
                    state         <= UPDATE_R;
                    done          <= 1'b0;
                    k             <= 0;
                    i             <= 0;
                    state_timer   <= 0;
                    read_stage    <= 0;
                    write_stage   <= 0;
                    max_pri_res_u <= 0;
                    max_pri_res_x <= 0;
                end
            end

            UPDATE_R: begin
                // r = ?R*u_ref ? rho*(z?y)
                state_timer <= state_timer + 1;
                if (state_timer == 1) begin
                    read_stage <= 1;
                    k <= 0; i <= 0;
```

```
                    end
                    if (read_stage == 1) begin
                        if (k < active_horizon && i < INPUT_DIM) begin
                            int index;
                           index = k*INPUT_DIM + i;
                            case (state_timer % 6)
                                1: begin
                                    R_rdaddress <= i;
                                    u_rdaddress <= index;
                                end
                                3: temp_u <= u_data_out;
                                4: begin
                                    temp_r     = -R_data_out * temp_u;
                                    r_data_in  <= temp_r - rho * (temp_u -
temp_u);
                                    r_wraddress<= index;
                                    r_wren     <= 1'b1;
                                end
                                default: begin
                                    r_wren <= 1'b0;
                                    i <= i + 1;
                                    if (i == INPUT_DIM-1) begin
                                        i <= 0;
                                        k <= k + 1;
                                    end
                                end
                            endcase
                        end else begin
                            read_stage <= 0;
                            r_wren     <= 1'b0;
                            state      <= UPDATE_Q;
                        end
                    end
                end

                UPDATE_Q: begin
                    // q = ?Q*x_ref ? rho*(v?g)
                    state_timer <= state_timer + 1;
                    if (state_timer == 1) begin
                        read_stage <= 1;
                        k <= 0; i <= 0;
                    end
                    if (read_stage == 1) begin
                        if (k < active_horizon && i < STATE_DIM) begin
```

```
                        int index;
                       index = k*STATE_DIM + i;
                        case (state_timer % 6)
                            1: begin
                                Q_rdaddress <= i;
                                x_rdaddress <= index;
                            end
                            3: temp_x <= x_data_out;
                            4: begin
                                temp_q     = -Q_data_out * temp_x;
                                q_data_in  <= temp_q - rho * (temp_x -
temp_x);
                                q_wraddress<= index;
                                q_wren     <= 1'b1;
                            end
                            default: begin
                                q_wren <= 1'b0;
                                i <= i + 1;
                                if (i == STATE_DIM-1) begin
                                    i <= 0;
                                    k <= k + 1;
                                end
                            end
                        endcase
                    end else begin
                        read_stage <= 0;
                        q_wren     <= 1'b0;
                        state      <= UPDATE_P;
                    end
                end
            end

            UPDATE_P: begin
                // p = ?P*x_ref (terminal cost)
                state_timer <= state_timer + 1;
                if (state_timer == 1) begin
                    read_stage <= 1;
                    k <= 0; i <= 0;
                end
                if (read_stage == 1) begin
                    if (k < active_horizon && i < STATE_DIM) begin
                        int index;
                       index = k*STATE_DIM + i;
                        case (state_timer % 6)
```

```
                    1: begin
                        P_rdaddress <= i;
                        x_rdaddress <= index;
                    end
                    3: temp_x <= x_data_out;
                    4: begin
                        temp_p     = -P_data_out * temp_x;
                        p_data_in  <= temp_p;
                        p_wraddress<= index;
                        p_wren     <= 1'b1;
                    end
                    default: begin
                        p_wren <= 1'b0;
                        i <= i + 1;
                        if (i == STATE_DIM-1) begin
                            i <= 0;
                            k <= k + 1;
                        end
                    end
                endcase
            end else begin
                read_stage <= 0;
                p_wren     <= 1'b0;
                state      <= CALC_RESIDUALS;
            end
        end
    end

    CALC_RESIDUALS: begin
        pri_res_u <= max_pri_res_u;
        pri_res_x <= max_pri_res_x;
        state     <= DONE_STATE;
    end

    DONE_STATE: begin
        done   <= 1'b1;
        r_wren <= 1'b0;
        q_wren <= 1'b0;
        p_wren <= 1'b0;
        if (!start) begin
            state <= IDLE;
            done  <= 1'b0;
        end
    end
```

```
                default: state <= IDLE;
            endcase
        end
    end

endmodule
```

9.12) residual_calculator.sv

```
None
// Computes primal and dual residuals to check for ADMM convergence


module residual_calculator #(
    parameter STATE_DIM = 12,             // Dimension of state vector (nx)
    parameter INPUT_DIM = 4,             // Dimension of input vector (nu)
    parameter HORIZON = 30,              // Maximum MPC horizon length (N)
    parameter DATA_WIDTH = 16,           // 16-bit fixed point
    parameter FRAC_BITS = 8,             // Number of fractional bits for fixed
point
    parameter ADDR_WIDTH = 9
)(
    input logic clk,                     // Clock
    input logic rst,                     // Reset
    input logic start,                   // Start signal

    // Inputs for residual calculation
    input logic signed [DATA_WIDTH-1:0] pri_res_u,    // Primal residual for
inputs from dual_update
    input logic signed [DATA_WIDTH-1:0] pri_res_x,    // Primal residual for
states from dual_update

    // Memory interfaces for z and z_prev
    output logic [ADDR_WIDTH-1:0] z_rdaddress,

    input logic [DATA_WIDTH-1:0] z_data_out,

    output logic [ADDR_WIDTH-1:0] z_prev_rdaddress,
    input logic [DATA_WIDTH-1:0] z_prev_data_out,

    input logic signed [DATA_WIDTH-1:0] pri_tol,      // Primal residual
tolerance
    input logic signed [DATA_WIDTH-1:0] dual_tol,     // Dual residual
tolerance

    input logic [31:0] active_horizon,  // Current horizon length to use

    // Outputs
    output logic signed [DATA_WIDTH-1:0] dual_res,    // Dual residual
    output logic converged,                    // Convergence flag
    output logic done                          // Done signal
);
```

```
    // State machine states
    localparam IDLE = 3'd0;
    localparam CALC_DUAL_RESIDUAL = 3'd1;
    localparam CHECK_CONVERGENCE = 3'd2;
    localparam DONE_STATE = 3'd3;

    // State variables
    logic [2:0] state;
    logic [31:0] i;                     // Generic counter
    logic [31:0] k;                     // Step counter for horizon
    logic [31:0] state_timer;           // State timer

    logic [31:0] read_stage;            // Tracks memory read sequencing

    // Temporary storage for values read from memory
    logic signed [DATA_WIDTH-1:0] temp_z;
    logic signed [DATA_WIDTH-1:0] temp_z_prev;

    // Temporary computation variables
    logic signed [DATA_WIDTH-1:0] temp_diff;    // Temporary diff value for
calculations
    logic signed [DATA_WIDTH-1:0] max_dual_res; // Maximum dual residual

    // Convergence flags
    logic pri_converged_u;
    logic pri_converged_x;
    logic dual_converged;

    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            done <= 0;
            converged <= 0;
            i <= 0;
            k <= 0;
            state_timer <= 0;
            read_stage <= 0;
            max_dual_res <= 0;
            pri_converged_u <= 0;
            pri_converged_x <= 0;
            dual_converged <= 0;
            dual_res <= '0;
```

```
            // Initialize memory control signals
            z_rdaddress <= 0;
            z_prev_rdaddress <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (start) begin
                        state <= CALC_DUAL_RESIDUAL;
                        done <= 0;
                        converged <= 0;
                        i <= 0;
                        k <= 0;
                        read_stage <= 0;
                        state_timer <= 0;
                        max_dual_res <= 0;
                    end
                end

                CALC_DUAL_RESIDUAL: begin
                    // Calculate dual residual based on z change: max_i |z_k -
z_{k-1}|
                    state_timer <= state_timer + 1;

                    if (state_timer == 1) begin
                        // Initialize the dual residual calculation process
                        read_stage <= 1;
                        k <= 0;
                        i <= 0;
                    end

                    if (read_stage == 1) begin
                        // Read z and z_prev values for current element
                        if (k < active_horizon-1 && i < INPUT_DIM) begin
                            // Calculate current index
                            int index;
                           index = k*INPUT_DIM + i;

                            // Memory read sequence
                            if (state_timer % 5 == 1) begin
                                // Stage 1: Set read addresses
                                z_rdaddress <= index;
                                z_prev_rdaddress <= index;
                            end else if (state_timer % 5 == 2) begin
                                // Stage 2: Wait for read to complete
```

```
                        end else if (state_timer % 5 == 3) begin
                            // Stage 3: Capture values
                            temp_z <= z_data_out;
                            temp_z_prev <= z_prev_data_out;
                        end else if (state_timer % 5 == 4) begin
                            // Stage 4: Calculate difference and update max
residual
                            temp_diff = temp_z - temp_z_prev;
                            if (temp_diff < 0) begin
                                temp_diff = -temp_diff; // Absolute value
                            end

                            // Update maximum dual residual
                            if (temp_diff > max_dual_res) begin
                                max_dual_res <= temp_diff;
                            end
                        end else begin
                            // Stage 5: Advance to next element
                            i <= i + 1;
                            if (i == INPUT_DIM-1) begin
                                i <= 0;
                                k <= k + 1;
                            end
                        end
                    end else begin
                        // Done calculating maximum dual residual
                        read_stage <= 0;
                        dual_res <= max_dual_res; // Store the final dual
residual
                        state <= CHECK_CONVERGENCE;
                    end
                end
            end

            CHECK_CONVERGENCE: begin
                // Check primal residuals against tolerance
                pri_converged_u <= (pri_res_u <= pri_tol);
                pri_converged_x <= (pri_res_x <= pri_tol);

                // Check dual residual against tolerance
                dual_converged <= (dual_res <= dual_tol);

                // Set overall convergence flag
```

```
                    converged <= pri_converged_u && pri_converged_x &&
dual_converged;

                    state <= DONE_STATE;
                end

                DONE_STATE: begin
                    done <= 1;
                    if (!start) begin
                        state <= IDLE;
                        //done <= 0;
                    end
                end

                default: state <= IDLE;
            endcase
        end
    end

endmodule
```

9.13) tb_slack_update.sv

None

```
`timescale 1ps/1ps

module tb_slack_update_alex3;
  // override parameters for easy testing
  localparam int STATE_DIM   = 2;
  localparam int INPUT_DIM   = 2;
  localparam int HORIZON     = 3;
  localparam int DATA_WIDTH  = 16;
  localparam int ADDR_WIDTH  = 3;  // enough to index up to 6 entries

  // DUT signals
  logic                          clk;
  logic                          rst;
  logic                          start;
  logic [ADDR_WIDTH-1:0]         u_rdaddress, y_rdaddress;
  logic signed [DATA_WIDTH-1:0] u_data_out, y_data_out;
  logic [ADDR_WIDTH-1:0]         z_wraddress;
  logic signed [DATA_WIDTH-1:0] z_data_in;
  logic                          z_wren;
  logic [ADDR_WIDTH-1:0]         x_rdaddress, g_rdaddress;
  logic signed [DATA_WIDTH-1:0] x_data_out, g_data_out;
  logic [ADDR_WIDTH-1:0]         v_wraddress;
  logic signed [DATA_WIDTH-1:0] v_data_in;
  logic                          v_wren;
  logic signed [DATA_WIDTH-1:0] u_min   [INPUT_DIM];
  logic signed [DATA_WIDTH-1:0] u_max   [INPUT_DIM];
  logic signed [DATA_WIDTH-1:0] x_min   [STATE_DIM];
  logic signed [DATA_WIDTH-1:0] x_max   [STATE_DIM];
  logic [31:0]                   active_horizon;
  logic                          done;

  // memories to drive DUT and capture outputs
  logic signed [DATA_WIDTH-1:0] u_mem [0:INPUT_DIM*HORIZON-1];
  logic signed [DATA_WIDTH-1:0] y_mem [0:INPUT_DIM*HORIZON-1];
  logic signed [DATA_WIDTH-1:0] x_mem [0:STATE_DIM*HORIZON-1];
  logic signed [DATA_WIDTH-1:0] g_mem [0:STATE_DIM*HORIZON-1];
  logic signed [DATA_WIDTH-1:0] z_mem [0:INPUT_DIM*HORIZON-1];
  logic signed [DATA_WIDTH-1:0] v_mem [0:STATE_DIM*HORIZON-1];

  // expected values
  logic signed [DATA_WIDTH-1:0] expected_z [0:INPUT_DIM*(HORIZON-1)-1];
  logic signed [DATA_WIDTH-1:0] expected_v [0:STATE_DIM*HORIZON-1];
```

```
// hook up DUT
slack_update_alex3 #(
  .STATE_DIM(STATE_DIM),
  .INPUT_DIM(INPUT_DIM),
  .HORIZON(HORIZON),
  .DATA_WIDTH(DATA_WIDTH),
  .ADDR_WIDTH(ADDR_WIDTH)
) dut (
  .clk(clk),
  .rst(rst),
  .start(start),
  .u_rdaddress(u_rdaddress),
  .u_data_out(u_data_out),
  .y_rdaddress(y_rdaddress),
  .y_data_out(y_data_out),
  .z_wraddress(z_wraddress),
  .z_data_in(z_data_in),
  .z_wren(z_wren),
  .x_rdaddress(x_rdaddress),
  .x_data_out(x_data_out),
  .g_rdaddress(g_rdaddress),
  .g_data_out(g_data_out),
  .v_wraddress(v_wraddress),
  .v_data_in(v_data_in),
  .v_wren(v_wren),
  .u_min(u_min),
  .u_max(u_max),
  .x_min(x_min),
  .x_max(x_max),
  .active_horizon(active_horizon),
  .done(done)
);

// drive memories
assign u_data_out = u_mem[u_rdaddress];
assign y_data_out = y_mem[y_rdaddress];
assign x_data_out = x_mem[x_rdaddress];
assign g_data_out = g_mem[g_rdaddress];

// capture DUT writes
always_ff @(posedge clk) begin
  if (z_wren)
    z_mem[z_wraddress] <= z_data_in;
```

```
  if (v_wren)
    v_mem[v_wraddress] <= v_data_in;
end

// clock
always #5 clk = ~clk;

initial begin
  // initialize clock and reset
  clk = 0;
  rst = 1;
  start = 0;
  active_horizon = HORIZON;

  // fill in test data for u and y
  // u_mem: [2, -3, 5, 0]
  u_mem[0] =  16'sd2;  u_mem[1] = -16'sd3;
  u_mem[2] =  16'sd5;  u_mem[3] =  16'sd0;
  // y_mem: [1, 1, -2, 4]
  y_mem[0] =  16'sd1;  y_mem[1] =  16'sd1;
  y_mem[2] = -16'sd2;  y_mem[3] =  16'sd4;

  // u_min=[0, -1], u_max=[3, 2]
  u_min[0] = 16'sd0;  u_min[1] = -16'sd1;
  u_max[0] = 16'sd3;  u_max[1] =  16'sd2;

  // expected z = clip(u+y) =
  // k=0,i=0: 2+1=3?3 ; i=1: -3+1=-2?-1
  // k=1,i=0: 5+(-2)=3?3; i=1:0+4=4?2
  expected_z[0] = 16'sd3;
  expected_z[1] = -16'sd1;
  expected_z[2] = 16'sd3;
  expected_z[3] = 16'sd2;

  // fill x and g
  // x_mem: [0,5, -1,8, -3,2]
  x_mem[0] =  16'sd0;   x_mem[1] =  16'sd5;
  x_mem[2] = -16'sd1;   x_mem[3] =  16'sd8;
  x_mem[4] = -16'sd3;   x_mem[5] =  16'sd2;
  // g_mem: [1,-1, 0,3, -2,4]
  g_mem[0] =  16'sd1;   g_mem[1] = -16'sd1;
  g_mem[2] =  16'sd0;   g_mem[3] =  16'sd3;
  g_mem[4] = -16'sd2;   g_mem[5] =  16'sd4;
```

```
    // x_min=[-2,-3], x_max=[4,6]
    x_min[0] = -16'sd2;   x_min[1] = -16'sd3;
    x_max[0] =  16'sd4;   x_max[1] =  16'sd6;

    // expected v = clip(x+g):
    // k=0:0+1=1?1; 5+(-1)=4?4
    // k=1:-1+0=-1?-1; 8+3=11?6
    // k=2:-3+(-2)=-5?-2; 2+4=6?6
    expected_v[0] =  16'sd1;
    expected_v[1] =  16'sd4;
    expected_v[2] = -16'sd1;
    expected_v[3] =  16'sd6;
    expected_v[4] = -16'sd2;
    expected_v[5] =  16'sd6;

    // pulse reset, then start
    #10 rst = 0;
    #10 start = 1;
    #10 start = 0;

    // wait for completion
    wait(done);
    #10;

    // check results
    $display("\n--- Checking z outputs ---");
    foreach (z_mem[i]) begin
      $display("z[%0d] = %0d (expected %0d)%s",
               i, z_mem[i], expected_z[i],
               (z_mem[i] === expected_z[i]) ? "" : "  <-- MISMATCH");
    end

    $display("\n--- Checking v outputs ---");
    foreach (v_mem[i]) begin
      $display("v[%0d] = %0d (expected %0d)%s",
               i, v_mem[i], expected_v[i],
               (v_mem[i] === expected_v[i]) ? "" : "  <-- MISMATCH");
    end

    $finish;
  end
endmodule
```

9.14) tb_dual_update.sv

```
None
`timescale 1ps/1ps

module tb_dual_update_alex2;
  //--- Parameters
  localparam STATE_DIM   = 2;
  localparam INPUT_DIM   = 2;
  localparam HORIZON     = 3;
  localparam DATA_WIDTH  = 8;
  localparam FRAC_BITS   = 0;
  localparam ADDR_WIDTH  = 4;

  //--- Clock, reset, start
  logic clk = 0;
  always #5 clk = ~clk;
  logic rst, start;
  logic [31:0] active_horizon;
  logic [DATA_WIDTH-1:0] rho;

  //--- DUT address & data ports
  logic [ADDR_WIDTH-1:0] x_rdaddress,  u_rdaddress,  z_rdaddress,  v_rdaddress;
  logic [ADDR_WIDTH-1:0] y_rdaddress,  g_rdaddress,  r_rdaddress,  q_rdaddress,
p_rdaddress;
  logic [ADDR_WIDTH-1:0] y_wraddress,  g_wraddress,  r_wraddress,  q_wraddress,
p_wraddress;
  logic [DATA_WIDTH-1:0] x_data_out,   u_data_out,   z_data_out,   v_data_out;
  logic [DATA_WIDTH-1:0] y_data_out,   g_data_out,   r_data_out,   q_data_out,
p_data_out;
  logic [DATA_WIDTH-1:0] y_data_in,    g_data_in,    r_data_in,    q_data_in,
p_data_in;
  logic [ADDR_WIDTH-1:0] R_rdaddress,  Q_rdaddress,  P_rdaddress;
  logic [DATA_WIDTH-1:0] R_data_out,   Q_data_out,   P_data_out;
  logic y_wren, g_wren, r_wren, q_wren, p_wren;
  logic done;

  //--- Reference trajectories = 0
  logic [DATA_WIDTH-1:0] x_ref[0:STATE_DIM-1][0:HORIZON-1];
  logic [DATA_WIDTH-1:0] u_ref[0:INPUT_DIM-1][0:HORIZON-2];

  //--- DUT instantiation
  dual_update #(
    .STATE_DIM(STATE_DIM),
    .INPUT_DIM(INPUT_DIM),
```

```
  .HORIZON(HORIZON),
  .DATA_WIDTH(DATA_WIDTH),
  .FRAC_BITS(FRAC_BITS),
  .ADDR_WIDTH(ADDR_WIDTH)
) dut (
  .clk(clk), .rst(rst), .start(start),
  // primal reads
  .x_rdaddress(x_rdaddress), .x_data_out(x_data_out),
  .u_rdaddress(u_rdaddress), .u_data_out(u_data_out),
  .z_rdaddress(z_rdaddress), .z_data_out(z_data_out),
  .v_rdaddress(v_rdaddress), .v_data_out(v_data_out),
  // dual reads/writes
  .y_rdaddress(y_rdaddress), .y_data_out(y_data_out),
  .y_wraddress(y_wraddress), .y_data_in(y_data_in), .y_wren(y_wren),
  .g_rdaddress(g_rdaddress), .g_data_out(g_data_out),
  .g_wraddress(g_wraddress), .g_data_in(g_data_in), .g_wren(g_wren),
  // linear-cost reads/writes
  .r_rdaddress(r_rdaddress), .r_data_out(r_data_out),
  .r_wraddress(r_wraddress), .r_data_in(r_data_in), .r_wren(r_wren),
  .q_rdaddress(q_rdaddress), .q_data_out(q_data_out),
  .q_wraddress(q_wraddress), .q_data_in(q_data_in), .q_wren(q_wren),
  .p_rdaddress(p_rdaddress), .p_data_out(p_data_out),
  .p_wraddress(p_wraddress), .p_data_in(p_data_in), .p_wren(p_wren),
  // cost-matrix reads
  .R_rdaddress(R_rdaddress), .R_data_out(R_data_out),
  .Q_rdaddress(Q_rdaddress), .Q_data_out(Q_data_out),
  .P_rdaddress(P_rdaddress), .P_data_out(P_data_out),
  // refs & params
  .x_ref(x_ref), .u_ref(u_ref),
  .rho(rho),
  .pri_res_u(), .pri_res_x(),
  .active_horizon(active_horizon),
  .done(done)
);

//--- Behavioral ?memories?
reg [DATA_WIDTH-1:0] u_mem[(HORIZON*INPUT_DIM)-1:0];
reg [DATA_WIDTH-1:0] z_mem[(HORIZON*INPUT_DIM)-1:0];
reg [DATA_WIDTH-1:0] y_mem[((HORIZON-1)*INPUT_DIM)-1:0];
reg [DATA_WIDTH-1:0] x_mem[(HORIZON*STATE_DIM)-1:0];
reg [DATA_WIDTH-1:0] v_mem[(HORIZON*STATE_DIM)-1:0];
reg [DATA_WIDTH-1:0] g_mem[(HORIZON*STATE_DIM)-1:0];
reg [DATA_WIDTH-1:0] r_mem[((HORIZON-1)*INPUT_DIM)-1:0];
reg [DATA_WIDTH-1:0] q_mem[((HORIZON-1)*STATE_DIM)-1:0];
```

```
  reg [DATA_WIDTH-1:0] p_mem[(HORIZON*STATE_DIM)-1:0];
  reg [DATA_WIDTH-1:0] R_mem[0:INPUT_DIM-1];
  reg [DATA_WIDTH-1:0] Q_mem[0:STATE_DIM-1];
  reg [DATA_WIDTH-1:0] P_mem[0:STATE_DIM-1];

  //--- Drive DUT reads
  always_comb begin
    u_data_out = u_mem[u_rdaddress];
    z_data_out = z_mem[z_rdaddress];
    y_data_out = y_mem[y_rdaddress];
    x_data_out = x_mem[x_rdaddress];
    v_data_out = v_mem[v_rdaddress];
    g_data_out = g_mem[g_rdaddress];
    r_data_out = r_mem[r_rdaddress];
    q_data_out = q_mem[q_rdaddress];
    p_data_out = p_mem[p_rdaddress];
    R_data_out = R_mem[R_rdaddress];
    Q_data_out = Q_mem[Q_rdaddress];
    P_data_out = P_mem[P_rdaddress];
  end

  //--- Capture DUT writes
  always_ff @(posedge clk) begin
    if (y_wren) y_mem[y_wraddress] <= y_data_in;
    if (g_wren) g_mem[g_wraddress] <= g_data_in;
    if (r_wren) r_mem[r_wraddress] <= r_data_in;
    if (q_wren) q_mem[q_wraddress] <= q_data_in;
    if (p_wren) p_mem[p_wraddress] <= p_data_in;
  end

initial begin
  integer idx;
  // Zero u_mem and z_mem
  for (idx = 0; idx < HORIZON*INPUT_DIM; idx++) begin
    u_mem[idx] = 0;
    z_mem[idx] = 0;
  end

  // Zero y_mem and r_mem
  for (idx = 0; idx < (HORIZON-1)*INPUT_DIM; idx++) begin
    y_mem[idx] = 0;
    r_mem[idx] = 0;
  end
  // Zero x_mem, v_mem, g_mem
```

```
  for (idx = 0; idx < HORIZON*STATE_DIM; idx++) begin
    x_mem[idx] = 0;
    v_mem[idx] = 0;
    g_mem[idx] = 0;
  end

  // Zero q_mem
  for (idx = 0; idx < (HORIZON-1)*STATE_DIM; idx++)
    q_mem[idx] = 0;

  // Zero p_mem
  for (idx = 0; idx < HORIZON*STATE_DIM; idx++)
    p_mem[idx] = 0;
    // Sample primal data
    // u = [1,2,3,4,...], z = [0,1,1,1,...]
    u_mem[0]=1; u_mem[1]=2; u_mem[2]=3; u_mem[3]=4;
    z_mem[0]=0; z_mem[1]=1; z_mem[2]=1; z_mem[3]=1;
    // x = [10,20,30,40,50,60], v = [5,15,25,35,45,55]
    x_mem[0]=10; x_mem[1]=20; x_mem[2]=30; x_mem[3]=40; x_mem[4]=50;
x_mem[5]=60;
    v_mem[0]= 5; v_mem[1]=15; v_mem[2]=25; v_mem[3]=35; v_mem[4]=45;
v_mem[5]=55;

    // Cost matrices (diagonals only matter)
    R_mem[0]=2; R_mem[1]=4;
    Q_mem[0]=3; Q_mem[1]=5;
    P_mem[0]=7; P_mem[1]=9;

    // Zero refs
    for (idx=0; idx<STATE_DIM; idx++)
      for (int k=0; k<HORIZON; k++) x_ref[idx][k] = 0;
    for (idx=0; idx<INPUT_DIM; idx++)
      for (int k=0; k<HORIZON-1; k++) u_ref[idx][k] = 0;

    // Run
    rst = 1; start = 0; active_horizon = HORIZON; rho = 1;
    #20 rst = 0;
    #20 start = 1;
    #10 start = 0;
    wait(done);
    #20;

    //--- Check results
    $display("\n=== y_mem (expected [1,1,2,3]) ===");
```

```
    for (idx=0; idx<(HORIZON-1)*INPUT_DIM; idx++)
      $display(" y_mem[%0d] = %0d", idx, $signed(y_mem[idx]));

    $display("\n=== g_mem (expected all 5) ===");
    for (idx=0; idx<HORIZON*STATE_DIM; idx++)
      $display(" g_mem[%0d] = %0d", idx, $signed(g_mem[idx]));

    $display("\n=== r_mem = y-z (expected [1,0,1,2]) ===");
    for (idx=0; idx<(HORIZON-1)*INPUT_DIM; idx++)
      $display(" r_mem[%0d] = %0d", idx, $signed(r_mem[idx]));

    $display("\n=== q_mem = g-v (expected [0,-10,-20,-30]) ===");
    for (idx=0; idx<(HORIZON-1)*STATE_DIM; idx++)
      $display(" q_mem[%0d] = %0d", idx, $signed(q_mem[idx]));

    $display("\n=== p_mem = g-v @ k=N-1 (expected [-40,-50]) ===");
    for (idx=0; idx<HORIZON*STATE_DIM; idx++)
      $display(" p_mem[%0d] = %0d", idx, $signed(p_mem[idx]));

    $finish;
  end
endmodule
```

9.15) tb_residual_calculator.sv

```
None
`timescale 1ns/1ps

module tb_residual_calculator_alex;

  //------------------------------------------------------------------------
  // Override parameters for a small test
  //------------------------------------------------------------------------
  localparam integer STATE_DIM   = 2;
  localparam integer INPUT_DIM   = 2;
  localparam integer HORIZON     = 3;
  localparam integer DATA_WIDTH  = 16;
  localparam integer FRAC_BITS   = 8;
  localparam integer ADDR_WIDTH  = 2;

  //------------------------------------------------------------------------
  // Testbench signals
  //------------------------------------------------------------------------
  logic                       clk;
  logic                       rst;
  logic                       start;
  logic [DATA_WIDTH-1:0]      pri_res_u;
  logic [DATA_WIDTH-1:0]      pri_res_x;
  logic [ADDR_WIDTH-1:0]      z_rdaddress;
  logic [ADDR_WIDTH-1:0]      z_prev_rdaddress;
  logic [DATA_WIDTH-1:0]      z_data_out;
  logic [DATA_WIDTH-1:0]      z_prev_data_out;
  logic [DATA_WIDTH-1:0]      pri_tol;
  logic [DATA_WIDTH-1:0]      dual_tol;
  logic [31:0]                active_horizon;
  logic [DATA_WIDTH-1:0]      dual_res;
  logic                       converged;
  logic                       done;

  //------------------------------------------------------------------------
  // Simple memories for z and z_prev
  //------------------------------------------------------------------------
  reg [DATA_WIDTH-1:0] z_mem     [0:INPUT_DIM*HORIZON-2];
  reg [DATA_WIDTH-1:0] z_prev_mem[0:INPUT_DIM*HORIZON-2];

  // Drive data_out combinationally
  assign z_data_out      = z_mem[z_rdaddress];
  assign z_prev_data_out = z_prev_mem[z_prev_rdaddress];
```

```
//------------------------------------------------------------------------
// Instantiate DUT
//------------------------------------------------------------------------
residual_calculator_alex #(
  .STATE_DIM   (STATE_DIM),
  .INPUT_DIM   (INPUT_DIM),
  .HORIZON     (HORIZON),
  .DATA_WIDTH  (DATA_WIDTH),
  .FRAC_BITS   (FRAC_BITS),
  .ADDR_WIDTH  (ADDR_WIDTH)
) dut (
  .clk             (clk),
  .rst             (rst),
  .start           (start),
  .pri_res_u       (pri_res_u),
  .pri_res_x       (pri_res_x),
  .z_rdaddress     (z_rdaddress),
  .z_data_out      (z_data_out),
  .z_prev_rdaddress(z_prev_rdaddress),
  .z_prev_data_out (z_prev_data_out),
  .pri_tol         (pri_tol),
  .dual_tol        (dual_tol),
  .active_horizon  (active_horizon),
  .dual_res        (dual_res),
  .converged       (converged),
  .done            (done)
);

//------------------------------------------------------------------------
// Clock generator
//------------------------------------------------------------------------
initial clk = 0;
always #5 clk = ~clk;

//------------------------------------------------------------------------
// Test sequence
//------------------------------------------------------------------------
initial begin
  // Reset
  rst = 1;  start = 0;
  #20 rst = 0;

  // Initialize memories:
```

```
    // index = k*INPUT_DIM + i
    // { z_prev_mem[0], z_prev_mem[1], z_prev_mem[2], z_prev_mem[3] } =
{1,4,2,7}
    // { z_mem[0],      z_mem[1],      z_mem[2],      z_mem[3] }      =
{3,1,8,5}
    z_prev_mem[0] = 16'd1;  z_mem[0] = 16'd3;
    z_prev_mem[1] = 16'd4;  z_mem[1] = 16'd1;
    z_prev_mem[2] = 16'd2;  z_mem[2] = 16'd8;
    z_prev_mem[3] = 16'd7;  z_mem[3] = 16'd5;

    // Set residual inputs and tolerances
    pri_res_u      = 16'd5;  // ≤ pri_tol → pri_converged_u = 1
    pri_res_x      = 16'd10; // >  pri_tol → pri_converged_x = 0
    pri_tol        = 16'd8;
    dual_tol       = 16'd6;
    active_horizon = 3;

    // Pulse start
    #10 start = 1;
    #10 start = 0;

    // Wait until done
    wait(done);

    // Display results
    $display("=== Simulation Results ===");
    $display("dual_res   = %0d  (expected 6)", dual_res);
    $display("converged  = %0b  (expected 0)", converged);
    $display("done       = %0b  (expected 1)", done);

    #10 $finish;
  end

endmodule
```

References

[1] F.R.Hogan,E.R.Grau,andA.Rodriguez,"Reactiveplanarmanipula- tion with convex hybrid mpc," in 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018, pp. 247–253.

[2] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. Del Prete, "Optimization-based control for dynamic legged robots," IEEE Transactions on Robotics, 2023.

[3] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Per- menter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," Autonomous robots, vol. 40, pp. 429–455, 2016.

[4] T. Antony and M. J. Grant, "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures," vol. 54, no. 5, pp. 1081– 1091.

[5] B. Plancher, S. M. Neuman, R. Ghosal, S. Kuindersma, and V. J. Reddi, "GRiD: GPU-Accelerated Rigid Body Dynamics with Analytical Gradients," in 2022 International Conference on Robotics and Automation (ICRA). IEEE, pp. 6253–6260.

[6] Y. Lee, M. Cho, and K.-S. Kim, "Gpu-parallelized iterative lqr with input constraints for fast collision avoidance of autonomous vehicles," in 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2022, pp. 4797–4804.

[7] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in 2024 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2024, pp. 9787–9794.

[8] K. Nguyen, S. Schoedel, A. Alavilli, B. Plancher, and Z. Manchester, "Tinympc: Model-predictive control on resource-constrained micro- controllers," in 2024 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2024, pp. 1–7.