0.1 FPGA Implementation Specification (Conventional Ethernet)

This appendix provides detailed specifications for implementing the CQ protocol in a Conventional Ethernet packet format in an FPGA, suitable for testing and evaluation.

0.1.1 Frame Format (Conventional Ethernet)

Field	Size (bits)	Description
Preamble*	64	Standard Ethernet preamble with SFD
Destination MAC*	48	Destination MAC address
Source MAC*	48	Source MAC address
EtherType*	16	Custom EtherType (0xCQ01)
Balance Indicator	3	Encoded balance state
Operation Code	5	Operation type
Transaction ID	16	Unique transaction identifier
Payload Length	16	Length of payload in bytes
Payload	Variable	Data payload (if applicable)
CRC	32	Frame check sequence

Table 1: CQ Protocol Frame Format.
*Not Needed in Æ-Link Interconnects.

Source & destination identifiers are redundant between adjacent Æ Cells.
i.e. Software Endpoints Directly Connected over a single link where (*private* identities and identifiers are in preframe negotiation).

0.1.2 Balance Indicator Encoding

Value	Meaning
000	$-\infty$ (Complete deficit)
001	-1 (Specific deficit)
010	-0 (Balance with negative tendency)
011	$+0$ (Balance with positive tendency)
100	$+1$ (Specific surplus)
101	$+\infty$ (Complete surplus)
110-111	Reserved

Table 2: Balance Indicator Encoding

0.1.3 Operation Code Encoding

Value	Operation
00000	NOP (No Operation)
00001	DATA (Data Transfer)
00010	ACK (Acknowledgment)
00011	REQ (Request for Data)
00100	RSP (Response to Request)
00101	SYNC (Synchronization)
00110	SYNC_ACK (Synchronization Acknowledgment)
00111	RESET (Connection Reset)
01000-11111	Reserved

Table 3: Operation Code Encoding

0.1.4 State Machine Definition

The core state machine for the CQ protocol implementation is defined as follows:

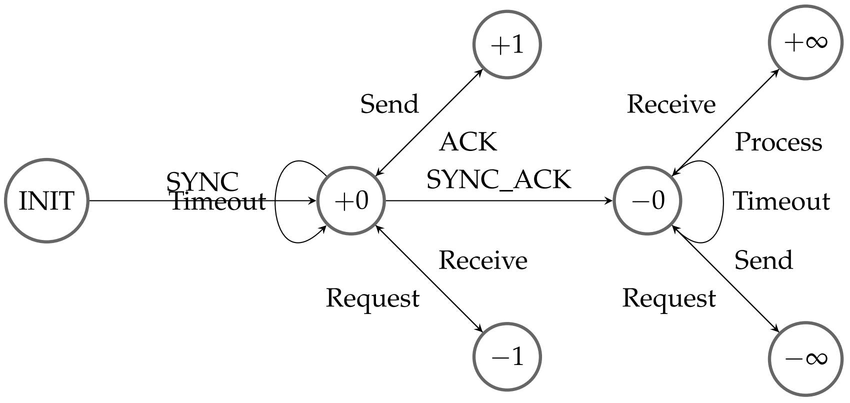

Figure 1: CQ Protocol State Machine

0.1.5 FPGA Implementation Architecture

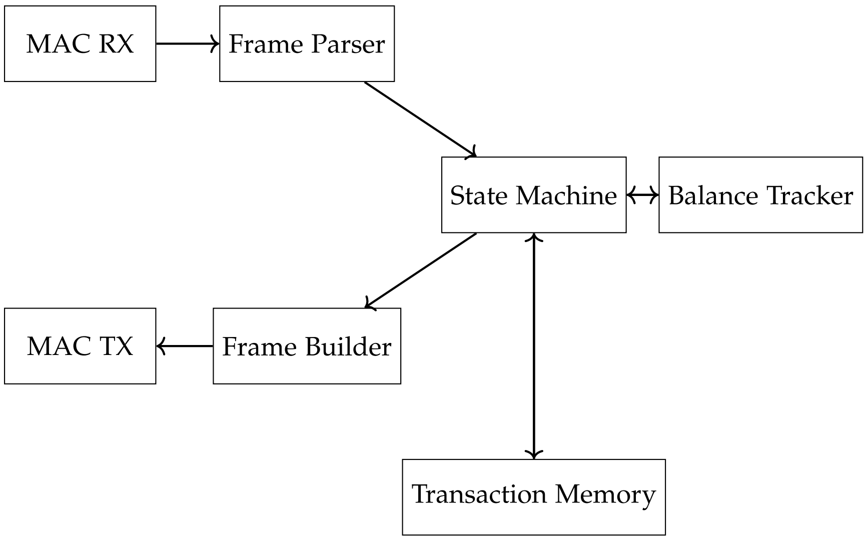

Figure 2: FPGA Implementation Architecture

0.1.6 Registers and Memory Structure

0.1.7 Memory Organization

The Transaction Memory should be implemented as dual-port RAM with the following structure:

0.1.8 Pseudo-Verilog for Core State Machine

```
module cq_state_machine (
    input wire clk,
```

Table 4: Register Definitions

Register	Width (bits)	Description
STATE_REG	3	Current protocol state
BALANCE_REG	3	Current balance indicator
TRANS_ID_REG	16	Current transaction ID
TIMEOUT_COUNTER	32	Timeout counter
CONTROL_REG	8	Control register
STATUS_REG	8	Status register

Table 5: Transaction Memory Structure

Field	Width (bits)	Description
Transaction ID	16	Key for the entry
Balance State	3	Associated balance state
Operation	5	Associated operation
Timestamp	32	Timestamp of last activity
Data Pointer	16	Pointer to data in payload memory
Data Length	16	Length of associated data

```
    input wire reset,
    input wire [2:0] rx_balance,
    input wire [4:0] rx_operation,
    input wire [15:0] rx_transaction_id,
    input wire frame_valid,
    output reg [2:0] tx_balance,
    output reg [4:0] tx_operation,
    output reg [15:0] tx_transaction_id,
    output reg tx_request,
    output reg [2:0] current_state
);

// State definitions
localparam STATE_INIT = 3'b000;
localparam STATE_PLUS_ZERO = 3'b001;
localparam STATE_PLUS_ONE = 3'b010;
localparam STATE_MINUS_ONE = 3'b011;
localparam STATE_MINUS_ZERO = 3'b100;
localparam STATE_PLUS_INF = 3'b101;
localparam STATE_MINUS_INF = 3'b110;

// Operation codes
localparam OP_NOP = 5'b00000;
localparam OP_DATA = 5'b00001;
localparam OP_ACK = 5'b00010;
localparam OP_REQ = 5'b00011;
localparam OP_RSP = 5'b00100;
localparam OP_SYNC = 5'b00101;
localparam OP_SYNC_ACK = 5'b00110;
localparam OP_RESET = 5'b00111;

// Internal registers
reg [31:0] timeout_counter;
reg timeout_occurred;

// State machine logic
always @(posedge clk or posedge reset) begin
    if (reset) begin
        current_state <= STATE_INIT;
        tx_balance <= 3'b000;
        tx_operation <= OP_NOP;
        tx_transaction_id <= 16'h0000;
        tx_request <= 1'b0;
        timeout_counter <= 32'h00000000;
        timeout_occurred <= 1'b0;
    end else begin
        // Default values
        tx_request <= 1'b0;

        // Timeout detection
        if (timeout_counter > 0) begin
            timeout_counter <= timeout_counter - 1;
            if (timeout_counter == 1) begin
                timeout_occurred <= 1'b1;
            end
        end

        // State transitions based on received frames and timeouts
        case (current_state)
            STATE_INIT: begin
                if (frame_valid && rx_operation == OP_SYNC) begin
                    current_state <= STATE_PLUS_ZERO;
                    tx_balance <= 3'b011; // +0
                    tx_operation <= OP_SYNC_ACK;
                    tx_transaction_id <= rx_transaction_id;
                    tx_request <= 1'b1;
                    timeout_counter <= 32'd100000; // Set appropriate timeout value
                end
            end

            STATE_PLUS_ZERO: begin
                if (frame_valid) begin
```

```
                    case (rx_operation)
                        OP_DATA: begin
                            current_state <= STATE_PLUS_ONE;
                            tx_balance <= 3'b100; // +1
                            tx_operation <= OP_ACK;
                            tx_transaction_id <= rx_transaction_id;
                            tx_request <= 1'b1;
                        end
                        OP_REQ: begin
                            current_state <= STATE_MINUS_ONE;
                            tx_balance <= 3'b001; // -1
                            tx_operation <= OP_RSP;
                            tx_transaction_id <= rx_transaction_id;
                            tx_request <= 1'b1;
                        end
                        OP_SYNC_ACK: begin
                            current_state <= STATE_MINUS_ZERO;
                            tx_balance <= 3'b010; // -0
                        end
                        // Handle other operations...
                    endcase
                end else if (timeout_occurred) begin
                    // Handle timeout in +0 state
                    timeout_occurred <= 1'b0;
                    tx_operation <= OP_SYNC;
                    tx_transaction_id <= tx_transaction_id + 1;
                    tx_request <= 1'b1;
                    timeout_counter <= 32'd100000;
                end
            end

            // Additional states and transitions...
            // STATE_PLUS_ONE, STATE_MINUS_ONE, etc.

        endcase
    end
end

endmodule
```

0.1.9 Test Vectors (Conventional Ethernet)

The following test vectors can be used to verify the implementation:

1. **Connection Establishment**:

- Node A sends SYNC with balance $+0$
- Node B responds with SYNC_ACK with balance -0
- Expected outcome: Both nodes establish connection

2. **Basic Data Transfer**:
   - Node A sends DATA with balance $+1$
   - Node B responds with ACK with balance $+0$
   - Expected outcome: Data successfully transferred
3. **Data Request**:
   - Node A sends REQ with balance -1
   - Node B responds with RSP with balance $+0$
   - Expected outcome: Requested data successfully received
4. **Error Recovery**:
   - Node A sends DATA with balance $+1$
   - Frame is lost (not injected in test)
   - Timeout occurs at Node A
   - Node A sends SYNC with balance $+0$
   - Node B responds with state information
   - Node A resends missing data
   - Expected outcome: Error recovered with minimal retransmission

0.1.10 Implementation Guidelines (Conventional Ethernet)

When implementing the CQ protocol in an FPGA, consider the following:

1. Use a pipelined architecture to achieve high throughput
2. Implement the transaction memory as dual-port RAM for simultaneous access
3. Use a parameterized design to allow configuration of buffer sizes, timeout values, etc.
4. Include comprehensive error detection and reporting mechanisms
5. Add debug ports to monitor internal state transitions
6. Implement the CRC calculation using parallel techniques for high performance
7. Consider using a dedicated timeout counter for each active transaction

0.1.11 Verification Plan (Conventional Ethernet)

To verify the implementation:

1. Use simulation with the provided test vectors to verify basic functionality
2. Test edge cases such as simultaneous transmissions and maximum-size frames
3. Measure performance metrics including latency, throughput, and resource utilization

4. Conduct stress testing with high packet rates and induced errors
5. Verify interoperability between multiple implementations