1. Bits and Bytes

1.1 64-Byte Record

Æthernet operates exclusively on fixed-sized records so every transmission has bounded entropy, and pipelines can be made in the hardware with latency guarantees. Most SerDes on the market are designed to operate on 8-byte (64-bit) atomic slices, which align naturally with fixed size encoding schemes like 64b/66b and enable efficient, low-entropy, and latency-predictable data movement through hardware pipelines.

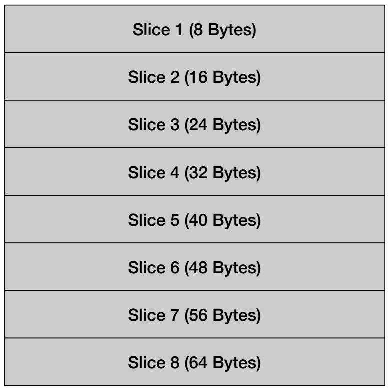

Figure 1.1: 64-Byte Record. 8 × 8 byte slices, pre-emptible by responders

[+40mm]

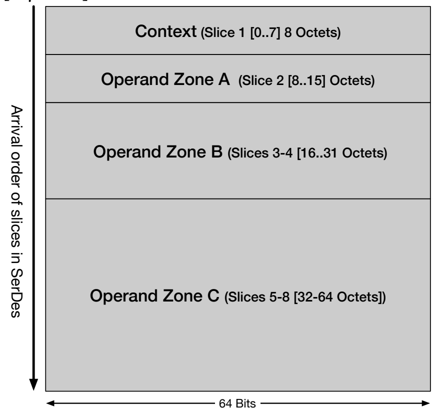
Context (Slice 1 [0..7] 8 Octets)
Operand Zone A (Slice 2 [8..15] Octets)
Operand Zone B (Slices 3-4 [16..31 Octets)
Operand Zone C (Slices 5-8 [32-64 Octets])
Arrival order of slices in SerDes
64 Bits

Figure 1.2: Slice Arrival order (Temporal Intimacy Depth)

8-bytes is the atomic unit of transmission in Æthernet, but the fundamental record size is 64 bytes, representing the maximum uninterrupted knowledge transfer permitted by a `LINK`. Each frame is structured into *slots* that correspond to exponentially increasing levels of entropy, at the expense of temporal intimacy.

1.1.1 Slot Boundaries

In Æthernet, each slot boundary contributes to the progressive construction of meaning. Rather than dividing slots into fixed roles (e.g., header vs payload), each slice refines the shared semantic context between sender and receiver. This unfolding process is tracked through a series of Sub-ACKs (SACKs), signaling progressively deeper certainty at four boundaries (1, 2, 4, and 8 slices). These boundaries correspond to conceptual layers of comprehension:

Slice 1: **Arrival of Context** — Establishes the physical link is live. The receiver confirms deserialization and framing; the message has landed.

Slice 2: **Recognition of Form** — Basic headers or structure emerge. Receiver begins to interpret role and framing, setting state machines into motion.

Slices 3–4: **Activation of Semantics** — The receiver has seen enough to begin logical interpretation: which class of message is it? What resources must it allocate?

Slices 5–8: **Consolidation of Understanding** — With full delivery, the entire 64-byte message is interpreted as a coherent unit. At this point, delivery to the host or downstream actor becomes safe and lossless.

Every slice carries data, but also a layer of **epistemic weight**. The meaning of the message doesn't come from a single part, but from the

cumulative structure of all slices, layered like a wavefunction collapsing toward certainty.

1.1.2 Pre-Emption

In Æthernet, it is the responsibility of the receiver to jam the sender and borrow the sender's token to transfer a frame the receiver wants to send. Due to physical limitations, the first few slices will arrive at the receiver before the jam signal contained in the first slice acknowledgement will override the frame's ownership to the receiver until the other side jams for ownership.

This allows one side of the link to jam the other side and utilize the full interaction capacity of the link for its frames. Pre-emption is decided on the first slice acknowledgement, until the other side has something it needs to send, and jams for ownership of one or both snakes.

There is no jam hierarchy or recursive jamming, frame ownership is a state owned entirely by the `LINK` and the `LINK` state machines determine when a frame is jammed for ownership. The jammed frame is immediately removed from the sender's queue, and ownership of the jammed frame is returned to the controller for possible re-routing or to jam the frame in at some backoff.

To ensure fair use of frames for maximum throughput, each link communicates status frames with the other side for leaning throughput in one direction or the other.

1.2 Flow Transactions

ULL protocol designers play around with 32 bits as the minimum unit of transactional transfer, but experiments demonstrate the difficulty of making this consistently reliable i; the general consensus is that modern SerDes' work best with $\geq$ 64 bit (8 Byte) slices/flits. Ethernet has a minimum frame size of 64 bytes (although only 42 bytes were available for the payload).

We therefore choose a *fixed* 64 Byte frame for the Shannon Slots, but make them *pre-emptable* so that even the minimum size frame does not need to occupy space on the wire, increase latency, or FPGA processing steps, when the receiver has something more important it wishes to send (e.g. local status messages sent in the background can be pre-empted, giving way to a two phase commit (2PC) transaction).

Some transactional systems are sensitive to making transactions reliable, but don't mind missing events, such as highly perishable market data. We might call these one-phase commit (1PC) transactions. These

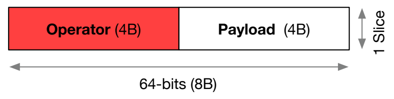

Figure 1.3: 1 Slice Flow Subtransaction

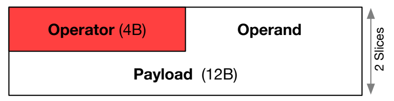

Figure 1.4: 2 slice Flow SubTransaction

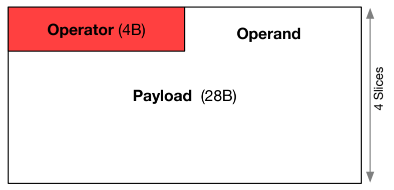

Figure 1.5: 4 4 slice Flow SubTransaction with 28B payload (operand)

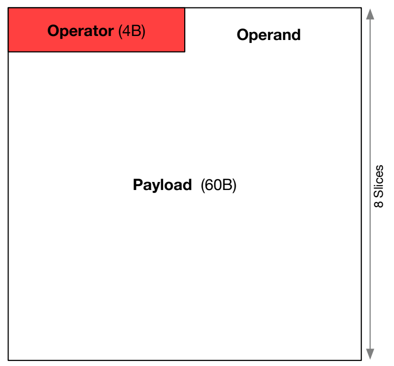

Figure 1.6: 1 $\times$ 8 slice Flow Transaction with 60B payload

can be made to flow at maximum line rate, even though each individual slice is being acknowledged. This is particularly important in HFT for example.

We therefore provide the following “flow" transactions in the encoding scheme:

1.2.1 Flow Unit Encodings

To enable ultra-low-latency transaction processing, the receiver must begin interpreting and dispatching semantic units (operator + operand) before the full 64-byte frame has arrived. This is made possible through lightweight inline encodings that declare, in the first slice of a transaction, the total number of slices that comprise that flow unit.

These encodings allow the receiver to pipeline semantic processing based on declared intent rather than full-frame arrival, dramatically reducing end-to-end transaction latency while preserving reliability.

1. **One 1-slice Flow Unit (4B payload)**
   `00` Indicates this flow unit consists of 1 slice.
2. **One 2-slice Flow Unit (12B payload)**
   `01` Indicates 2 slices are part of this flow unit. The receiver counts down remaining slices before handoff.
3. **One 4-slice Flow Unit (28B payload)**
   `10` Indicates 4 slices make up this flow unit. The receiver pipelines semantic interpretation during arrival.
4. **One 8-slice Flow Unit (60B total payload)**
   `11` Indicates 8 slices make up this flow unit. The receiver waits for the full frame before semantic interpretation.

1.2.2 Mixing and Matching Flow Transactions

You can also mix them in the same frame, but remember, they can only be used for One-Phase-Commit (1PC) in a single stream of transactions. This is because 1PC requires only one "round trip", whereas 2PC requires two round trips (although this scheme can be made to work for 2PC, and perhaps 4PC, but they have not yet been tested).

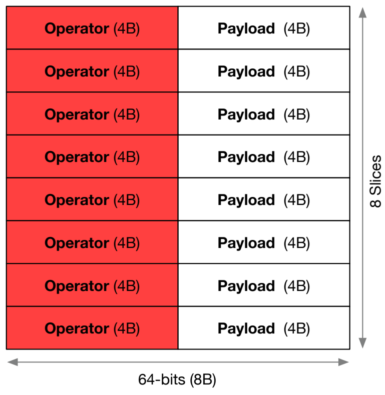

Figure 1.7: 8 independent Flow Transactions in a one frame

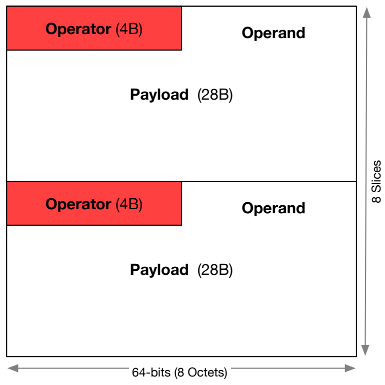

Figure 1.8: 2 × 4 slice Flow Transactions

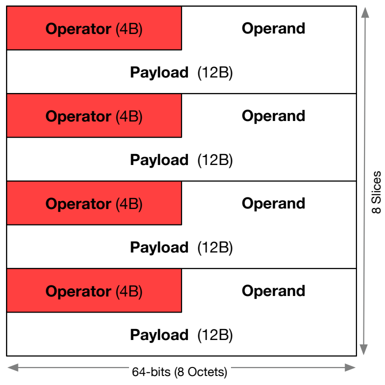

Figure 1.9: 4 × 2 slice Flow Transactions

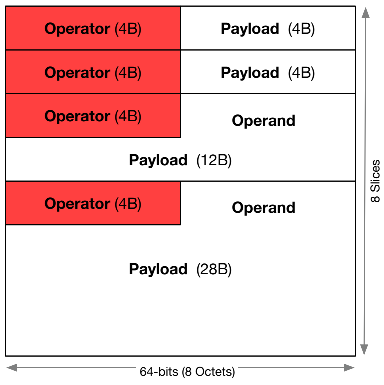

Figure 1.10: 1 64 Byte frame with differently sized flow units

Flows	Operator	Operand	Efficiency
1	4	4	50%
1	4	12	75%
1	4	28	87.5%
1	4	60	93.75%
2	4	4	100%
2	4	12	150%
2	4	28	175%
2	4	60	187.5%
4	4	4	200%
4	4	12	300%
4	4	28	350%
4	4	60	375%
8	4	4	400%
8	4	12	600%
8	4	28	700%
8	4	60	750%

Table 1.1: Transaction efficiency by operator and operand size.

1.3 RISC Protocol Design: OPCODE (Information)

1.3.1 CONTEXT Frame format: First Slice, First Byte: OPCODE

(SLICE, BEATS, PROTOCOL, JAM) provides state encodings for an ultra-low-latency, hardware-friendly, and atomic transaction-friendly Æthernet protocol.

Supports transactional operations, structured acknowledgments, and reversible flow control (causal backpropagation). Instead of positive-only credits, the first hop receiver provides the equivalent of negative credits, to indicate it is returning previously sent frames.

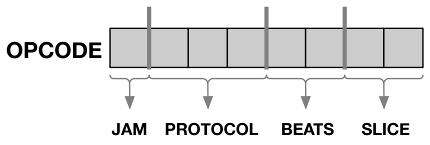

Figure 1.11: One Byte Provides the entry point for an Entire family of Protocols

1.3.2 nSLICE

Set by the Sender to`[00]` – indicating a new context.

Modified by the receiver Closing the loop: `[11]`$\rightarrow$`[10]`$\rightarrow$`[01]`$\rightarrow$`[00]`

Encodes how many slices of the sender's 64-byte Frame has been received so far. A 2-bit field with reversed temporal direction to encode the acknowledgment depth in a power of 2 *number of slices*. This might represent the trailing edge of a window in a reversible or partially committed state machine. The naming "SACK" suggests slot or slice acknowledgments, as fine-grained positions in the interaction.

8SLICE	`11 -- TX Sender Init`
	`11 -- RX SACK 1 (8B)`
	`10 -- RX SACK 2 (16B)`
	`01 -- RX SACK 3 (32B)`
	`00 -- RX SACK 4 (64B)`

4SLICE	`10 -- TX Sender Init`
	`10 -- RX SACK 3 (32B)`
	`01 -- RX SACK 1 (8B)`
	`00 -- RX SACK 2 (16B)`

2SLICE	`01 -- TX Sender Init`
	`01 -- RX SACK 1 (8B)`
	`00 -- RX SACK 2 (16B)`

1SLICE	`00 -- TX Sender Init`
	`00 -- RX SACK 1 (8B)`

1.3.3 BEATS

1

Defines a beat-structured flow control mechanism. Sender declares the number of frames it plans to send advance. The receiver responds with a corresponding "slot acknowledgment". Aimed at reliable, ordered delivery without the need for heavyweight TCP.

BEATS	`00 -- TX 1 FRAME (64B)`
	`01 -- TX 4 FRAMES (256B)`
	`10 -- TX 16 FRAMES (1024B)`
	`11 -- TX 64 FRAMES (4096B)`
	`00 -- RACK 1 FRAME (64B)`
	`01 -- RACK 4 FRAMES (256B)`
	`10 -- RACK 16 FRAMES (1024B)`
	`11 -- RACK 64 FRAMES (4096B)`

PROTOCOL	`000 -- Initialization`
	`001 -- Liveness`
	`010 -- State Machines`
	`011 -- RESERVED`
	`100 -- RESERVED`
	`101 -- RESERVED`
	`110 -- RESERVED`
	`111 -- ESCAPE`

JAM	`ABORT/CANCEL`

1.3.4 PROTOCOL

This field defines the high-level intent of the frame or transaction, by the sender (causal initiator). The 3-bit code is always in the first (context) slice of the Frame. Three of the eight possibilities are defined in this specification. The remaining ones are reserved for higher level protocols in this standard. Escape will always be available to escape to legacy protocols. This compact opcode space (3 bits) is similar to what RISC architectures do. This simplifies logic at the NIC or SmartNIC level and allows for deterministic dispatch.

1.3.5 PRE-EMPT/JAM

[2]

[2] The use of "JAM" evokes classic Ethernet collision handling, but here it's modernized for transactional cancellation or rollback.

Set by TX to `[0]`. Set by RX to `[0]` to accept, and `[1]` to pre-empt, for error, or to (cancel/rollback the transaction).

1.4 RISC Protocol Design: LIVENESS (Knowledge)

Protocol								Liveness								State Machine								Transition							

[3]

[3] *First Slice:* `CONTEXT` *(Packet Mission). All bits are green (owned and written by Alice)*

1.4.1 Bipartite Link

There are exactly two parties on the DAE Link. We could call them `alice` and `bob`. We prefer to call them self and not self. From Alice's perspective, she knows her own identify, but she does not know the identity of the party she is communicating with (yet). We aim to achieve mathematical precision in our specifications. This will be important when we wish to formally verify the scouting, routing, and cluster membership protocols. It will be critical also in formally verifying confinement properties of the trees above.

The encoding supports Intanglement (hidden circulating events internal to the link) and Extanglement (Atomic Token Passing through the link (Newtons cradle). These protocols obey the mathematics of mutual information, and provides some of the properties of quantum entanglement, such as superposition, conservation of information, and no-cloning. We use these properties to provide our protocols with a clear notion of simultaneity (through the synchronization of mutual information), and guarantee atomicity for transaction protocols through conserved quantities which in-turn guarantees exactly once semantics (EOS).

Conventional L2 & L3 networks rely on redundancy, repetition and rerouting, in multipartite (1:N) relationships. Which was necessary when information is disseminated (transmitted blindly hoping the receiver catches it). When information can also be synchronized, by a Tx/Rx—T/Rx loop on a bipartite Ethernet link, we can employ Pseudo Entanglement: A form of temporal intimacy, where bits shared in a circulating frame can exploit the same mathematics, (but not the full quantum properties) of Entanglement. This insight allows us to engineer a clear notion of simultaneity, and exploit a classical version of the no-cloning theorem to achieve the holy grail in distributed systems and database isolation: exactly once semantics.

1.4.2 Link Engine

Alternating Causality (AC) is the name we give to the initialization, maintenance and tear down of Common Knowledge (CK) in the Link. Experience with modern SerDes designs leads us to an 8 byte slice architecture for a "minimum irreducible" CK protocol. Symmetry demands that we use half (4-bytes) for `alice` (what I know about me) and the other 4-bytes for `bob` (what I know about you). Three packet exchanges get us from initialization (both sides know nothing about each other) to the "I know that you know that I know" (IKT YKT IKT) equilibrium state for basic liveness.

It takes a while to gain an intuition for this issue of causality, based on the physics. For now, please accept that this is way of doing things is essential and enables a rich set of transaction types to be built on top, all with immunity to link hazards.

We don't use classical (increment only) clocks, counters, or timers in the link. Instead, we use balanced ternary arithmetic [1] The digits of a balanced ternary numeral are coefficients of powers of 3, but instead of coming from the set 0, 1, 2, the digits are -1, 0 and +1. They are balanced because they are arranged symmetrically about zero. We use this symmetry to manage the direction of causality (is alice the initiator of causal flow sending tokens to bob, or the receiver in causal flow receiving tokens from bob?). This becomes important as we go up the protocol stack and construct reversible subtransactions.

We extend the simple ternary arithmetic with plus and minus zero. -1,-0,+0,+1. This enables the protocol to differentiate between the posibits and negabits [2], with an ancilla control over the intended direction of the next operation (positive or negative). This is used to control the direction of the state machine when recovering from errors.

Intanglement is enabled by reserving 4 bits in the frame for CK (2 bits for Alice, 2 Bits for Bob). One message will let Bob know about Alice. A second message lets Alice know that Bob knows, the third message lets Bob know that Alice knows that Bob Knows, consistent with both Moses and Halpern version of CK, and the Spekkens Knowledge Balance Principle (KBP). Time, inside the link moves forward when packets arrive. Time moves backwards when packets depart. It doesn't matter how many times a packet bounces around, time goes forward only when it is received by one end of the link and it stays (is absorbed). Information is then turned into Knowledge.

In a similar way to two phase locking, Link CK can be extended from 2 Ternary bits (Trits) to any number. Since we are using 2 binary bits to encode one Trit, we posit that the set of 2-message exchanges to synchronize them is 1, 2, 4 and any multiple of 4. This observation drives the encoding for the State Machine Engine, Described below. Our Protocol is based on Reversible Computing.

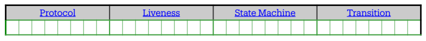

Figure 1.12: First Slice: CONTEXT. Least significant 32 bits of transmitted packet.

1.4.3 Slice Engine

The core of the Æ protocol is the Slice Engine. The first slice (or pre-frame slice) determines the packet mission, and carries the alternating causality for the Link State Machine (LSM).

Each 64-bit slice represents an atomic delivery of bits on the wire from the SerDes. Typically 2 slices will be sent back to back and the Slice Engine must be prepared to receive both, although the receiver may decide to pre-empt the frame in its immediate response to the first slice if it wishes to immediately begin a real data or transaction operation. The second slice will be on its way, and its Error Detection Byte must be evaluated before forwarding on other ports (with the exception of the port it was received on, which is the entanglement mechanism).

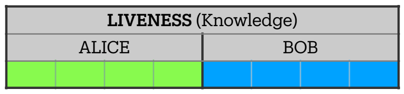

Figure 1.13: One Byte Provides Knowledge

The first slice completely defines the rest of the frame. There are 4 fields: PROTOCOL, LIVENESS, STATE, and TRANSITION. This is "reflected" from the upper half to the lower half by the receiver, so that only the lower 32 bits are modified, and the upper 32 bits remain unmodified.

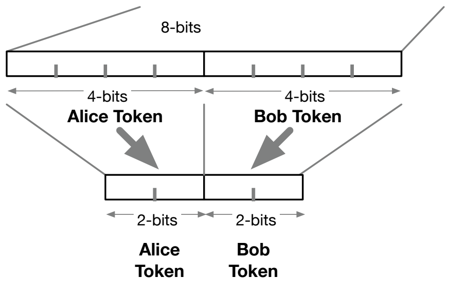

Figure 1.14: First Rewriting Rule. Alice Owns and possesses Context Slice

The `PROTOCOL` Byte defines the "mission" of the packet. What each side of the link needs the other side to know about the current frame. `LIVENESS` defines the Temporal Intimacy of the link — whether events on both sides of the link are directly connected or not.

`STATE` Defines which state machine is currently in use. Can be used as a sanity check in conjunction with Protocol. Transition Defines which state in the state machine we are in, and which direction we are going (forward or reverse).

1.4.4 General Principles

Links are constantly interacting, at the slice level, instead broadcasting entire frames (or sets of frames) imposing on the other side and hoping they catch the bits. This provides opportunities for error detection and correction that would otherwise require ECC and FEC. The theory behind this is described in detail in the document "Shannon-Interaction-Machine".

The first 4 slices are dedicated to Theseus (scouting protocols). The payload (slices 4-7) contain the Theseus Opcode and parameters — instructions to the scout, including what to do if it encounters an exception (a software or hardware hazard).

When the protocol type is Ariadne (groundplane/trees) the last 4 slices (payload) contains tree-building instructions, such as the CellID of the originator, and the CellID of the Deputy (one hop away from root). This becomes a complete specification for dissemination of the tree without unnecessarily revealing secrets which need to be kept local (confined).

Another protocol type is Icarus (legacy connections to the outside world). This represents a more heavyweight protocol which provides a formally verified TPI (Transaction Processing Interface), which provides significant guarantees, but with costs.

1.4.5 General Frame Format

	B1	B2	B3	B4	B5	B6	B7	B8
S1	Protocol	Liveness	State Machine	Transition	Protocol	Liveness	State Machine	Transition
S2	Operand 1 (2nd Slice)							
S3	Operand 2 (3rd/4th Slice)							
S4								
S5	Operand 3 (5th through 8th Slice)							
S6								
S7								
S8								

This protocol is *symmetric*. We describe all operations from the perspective of `ALICE`, with responses from `BOB`.

1.4.6 Error Detection and Correction

The transmitted first (context) slice is reflected by the receiver back to the transmitter – this Perfect Information Feedback [Ref] means that the context byte does not need additional error detection codes such as Checksums, CRC or FEC. This is especially true with flow transactions.

However, the rest of the payload is under the complete control of the application, and the Application can append (within the available blocks) any coding scheme it wishes to ensure that the data arrives intact and untampered with. This will often mean that the senders and receivers will have pre-arranged cryptographic keys which allow them to manage the entropy and cryptographic strength of the authentication.

1.4.7 No EDC or FEC

Each side of the link maintains two EPI (epistricted) registers : the last slice sent out, and the last slice received. The sender "owns" the lower 32 bits, and preserves the upper 32 bits. When slice 1 is received, the upper 32 bits are swapped with the lower 32 bits. This preserves the symmetry of the protocol, and clearly delineates the causal initiator register field ownership in addition to causal ownership.

See Quantum Ethernet

This provides the first level of error detection: the Initiator has Perfect Information Feedback (PIF) and sees. exactly what the receiver sees, and compare it to what was sent, And if they don't agree, declare an error and proceed with mitigations to get the link back in sync again.

1.4.8 Epistricted registers

Imagine two vectors [abcd] one for Alice and one for Bob. A 4 x 4 matrix has 16 slots, which has $2^{16} = 65,535$ possible states. However, according t o the Spekkens Toy model applied to FPGA Registers, there are only 12 'disjoint' (6 for Alice and a complimentary 6 for Bob). Instead of trying to build a EDD/EDC code, we check only the disjoint states by combining them into one register and sending them back and forth in the context frame.

1.4.9 OVERVIEW

1.4.10 Protocol Overview

TRANSACTION FABRIC: A separate compute realm, sandwiched between the CXL bus and Ethernet, to support database semantics. We eliminate CAP Theorem tradeoffs, by providing the illusion of an unbreakable network: detecting, isolating and healing failures far faster than protocol or application stacks using traditional timeouts and retries.

THESEUS: Ethernet-based scouting protocols explore local environments to discover and bring back knowledge of resources, constraints, and topologies in local (Chiplet) environments. THESEUS silently monitors local connectivity, raising alerts when links become flakey or server software hiccups.

ARIADNE: Ethernet based routing protocols dynamically construct and tear down communication graphs for consensus, load balancing and failover in global (rack-scale) environments. Enables: observability on demand, fault isolation and distributed debugging.

ICARUS: Connects the secure internal world of the Transaction Fabrix with the hostile external world of legacy systems and networks; using compositional (zero knowledge) techniques: formally verified APIs, comprehensively tested implementations.

LABYRINTH: A simulator driven toolset for Chiplet based micro-datadatacenters. Based on algorithms whose assumptions about causality go beyond simplistic notions of time. We empower distributed system developers with formally verified rules and FPGAs to execute Reversible Subtransactions 'invisibly' and 'indivisibly' in the Transaction Fabrix.

1.5 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.

1.5.1 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 1.2: 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 pre-frame negotiation).

1.5.2 Balance Indicator Encoding

1.5.3 Operation Code 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 1.3: Balance Indicator 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 1.4: Operation Code Encoding

1.5.4 State Machine Definition

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

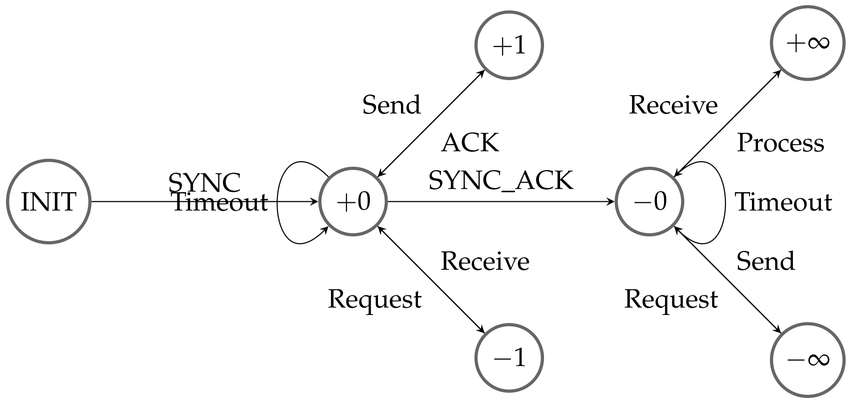

Figure 1.15: CQ Protocol State Machine

1.5.5 FPGA Implementation Architecture

1.5.6 Registers and Memory Structure

1.5.7 Memory Organization

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

1.5.8 Pseudo-Verilog for Core State Machine

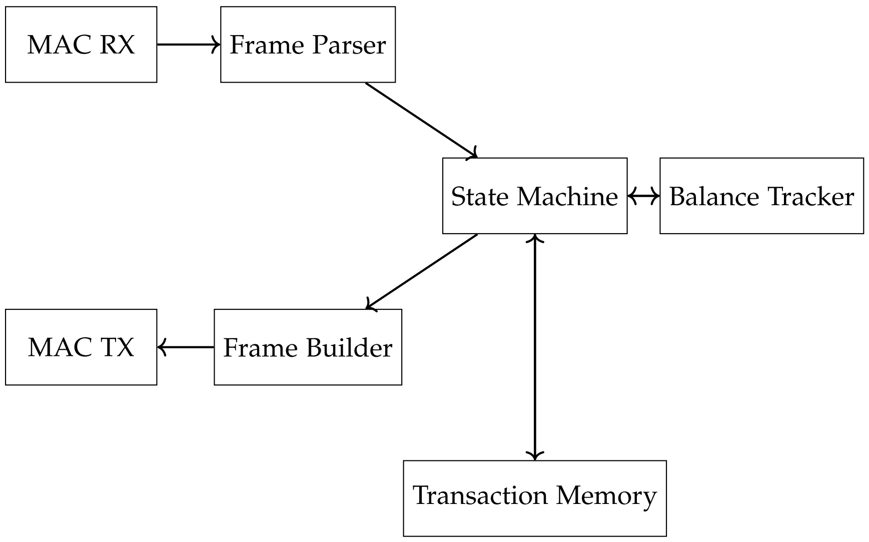

Figure 1.16: FPGA Implementation Architecture

Table 1.5: 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

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

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

Table 1.6: Transaction Memory Structure

```
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
```

1.5.9 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

1.5.10 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

1.5.11 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

1.6 Atomic Ethernet Frame Format: Æ-Link CQ Interactions

Field	Size (bits)	Context
Slice 1	8	OPCODE (Protocol Specifier)
Slice 1	8	Liveness (TIKTIKTIK)
Slice 1	8	State (State Machine Specifier)
Slice 1	8	Transition (State Machine Transition)
Slice 1	32	Operand (One-shot CQ Interactions)
Slice 2–8	512	Operand (One-Shot CQ Interactions)

Table 1.7: Æ Minimalist CQ Protocol. See: Slice Engine Frame Format Spec.

1.7 Slice 1 – Byte 1 Protocol

Field	Size (bits)	OPCODE
Slice 1	8	Context (Protocol Specifier)

Table 1.8: Protocol Specifier.

1.7.1 Slice 1 – Byte 2 Liveness

Field	Size (bits)	LIVENESS
Slice 1	8	TIKTIKTIK (Liveness Specifier)

Table 1.9: Liveness Specifier.