FPGA Engineer

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the difference between blocking and non-blocking assignments in Verilog and when each should be used.

Blocking vs Non-Blocking Assignments

Blocking assignments (=) execute sequentially within a procedural block. The next statement waits until the current assignment completes. They model combinational logic and are used in always @(*) blocks.

Non-blocking assignments (<=) schedule assignments that occur simultaneously at the end of the time step. They model sequential logic and are used in always @(posedge clk) blocks.

Best Practices:

- Use **blocking (=)** for combinational logic in always @(*)
- Use **non-blocking (<=)** for sequential logic in clocked always blocks
- Never mix both types in the same always block
- Violating these rules causes simulation/synthesis mismatches

```
// Sequential - use non-blocking
always @(posedge clk)
  q <= d;

// Combinational - use blocking
always @(*)
  sum = a + b;
```

2. What is metastability in digital design and how do you prevent it in FPGA implementations?

Metastability Overview

Metastability occurs when a flip-flop's setup or hold time is violated, causing the output to enter an undefined state between logic 0 and 1. This happens when asynchronous signals cross clock domains.

Consequences:

- Unpredictable circuit behavior
- Propagation of invalid logic levels
- System failures in downstream logic

Prevention Techniques:

- **Synchronizer chains:** Use 2-3 flip-flop stages to allow metastability to resolve (MTBF increases exponentially)
- **Handshake protocols:** Implement request/acknowledge signaling
- **Gray coding:** For multi-bit signals (counters, pointers)
- **Async FIFOs:** For data transfer between clock domains
- **Timing constraints:** Set proper false paths and max delay constraints

```
// Two-stage synchronizer
reg sync_ff1, sync_ff2;
always @(posedge clk) begin
  sync_ff1 <= async_in;
  sync_ff2 <= sync_ff1;
end
assign sync_out = sync_ff2;
```

3. Describe the concept of timing closure in FPGA design. What strategies do you use to meet timing requirements?

Timing Closure Definition

Timing closure is the process of meeting all setup and hold time requirements across all paths in your design at the target clock frequency. It's often the most challenging phase of FPGA development.

Key Timing Metrics:

- **Setup slack:** Time margin before data must be stable (positive = passing)
- **Hold slack:** Time data must remain stable after clock edge
- **WNS (Worst Negative Slack):** Most critical failing path
- **TNS (Total Negative Slack):** Sum of all negative slacks

Timing Closure Strategies:

- **Pipelining:** Insert registers to break long combinational paths
- **Retiming:** Move registers across combinational logic
- **Floorplanning:** Place related logic physically close
- **Clock constraints:** Properly constrain all clocks, I/O, and cross-domain paths
- **Resource optimization:** Reduce LUT depth, use dedicated resources (DSP, BRAM)
- **Incremental compilation:** Lock down timing-met regions
- **Physical synthesis:** Enable timing-driven placement/routing

```
// Pipelining example
always @(posedge clk) begin
  stage1 <= a * b;
  stage2 <= stage1 + c;
  result <= stage2;
end
```

4. What are the key differences between SRAM-based FPGAs (Xilinx/Intel) and Flash-based FPGAs (Microsemi/Lattice)?

SRAM-Based FPGAs (Xilinx, Intel)

- **Configuration:** Volatile - requires external memory (Flash, EEPROM) to load bitstream on power-up
- **Advantages:** Higher density, more logic resources, faster reconfiguration, partial reconfiguration support
- **Use cases:** High-performance computing, data centers, complex signal processing
- **Security:** Bitstream can be encrypted but vulnerable during configuration
- **Power-up time:** Slower (10-100ms) due to external loading

Flash-Based FPGAs (Microsemi, Lattice)

- **Configuration:** Non-volatile - bitstream stored in on-chip Flash
- **Advantages:** Instant-on, single-chip solution, better security, lower system cost
- **Use cases:** Aerospace, defense, automotive, industrial control
- **Radiation tolerance:** Better SEU immunity
- **Limitations:** Lower density, limited write cycles (Flash endurance)

Selection Criteria:

- Choose **SRAM-based** for maximum performance and density
- Choose **Flash-based** for security, instant-on, harsh environments

5. Explain clock domain crossing (CDC) and describe at least three methods to safely transfer data between clock domains.

Clock Domain Crossing (CDC)

CDC occurs when signals pass between logic operating on different clocks. Improper handling causes metastability, data corruption, and functional failures.

Method 1: Two-Stage Synchronizer

For single-bit control signals. Reduces MTBF of metastability.

```
reg [1:0] sync;
always @(posedge clk_dst)
  sync <= {sync[0], sig_src};
```

Method 2: Handshake Protocol

For multi-bit data requiring guaranteed delivery. Uses request/acknowledge signals synchronized independently.

Method 3: Asynchronous FIFO

For high-throughput data streams. Uses Gray-coded pointers synchronized across domains. Read/write operate on separate clocks.

Method 4: Gray Code Counters

For multi-bit values where only one bit changes per transition, minimizing metastability risk.

Method 5: MUX Recirculation

For slow-to-fast domain crossing with data hold guarantee.

Critical Considerations:

- Never synchronize multi-bit buses directly
- Apply proper timing constraints (set_false_path, set_max_delay)
- Use CDC verification tools (Spyglass, Questa CDC)
- Document all CDC crossings in design

6. What is the difference between a latch and a flip-flop? Why are latches generally avoided in FPGA designs?

Flip-Flop vs Latch

Flip-flop: Edge-triggered storage element. Captures data only on clock edge (rising or falling). Output changes only at clock transitions.

Latch: Level-sensitive storage element. Transparent when enable is active (passes input to output), holds value when enable is inactive.

Why Latches Are Avoided in FPGAs:

- **Timing analysis complexity:** Level-sensitive behavior makes static timing analysis difficult
- **Glitch sensitivity:** Can capture spurious transitions during transparent phase
- **Race conditions:** More susceptible to timing hazards
- **Synthesis issues:** Often inferred unintentionally due to incomplete sensitivity lists or missing else clauses
- **Resource inefficiency:** FPGAs optimized for flip-flops; latches use more resources
- **Simulation/synthesis mismatch:** Behavioral simulation may differ from synthesized hardware

When Latches Appear (Unintentionally):

```
// Creates a latch - missing else
always @(*)
  if (enable)
    q = d;
// Correct version
always @(*)
  q = enable ? d : q;
```

Best practice: Use flip-flops exclusively; ensure all combinational paths are fully specified.

7. Describe the architecture of a modern FPGA. What are LUTs, CLBs, DSP blocks, and block RAM?

FPGA Architecture Components

1. Look-Up Tables (LUTs)

The fundamental logic building block. Typically 4-6 input LUTs implement any combinational function. Internally, they're small SRAM arrays storing truth tables.

- 6-input LUT can implement any 6-input Boolean function
- Can be fractured into smaller LUTs or used as distributed RAM/shift registers

2. Configurable Logic Blocks (CLBs/LABs)

Groups of LUTs with associated flip-flops, multiplexers, and carry chains. The basic repeating unit.

- **Xilinx CLB:** Contains 2 slices, each with 4 LUTs and 8 flip-flops
- **Intel ALM:** Adaptive logic module with fracturable LUTs

3. DSP Blocks (DSP48/DSP58)

Hard-wired multiply-accumulate units for arithmetic operations.

- 25x18 or 27x27 multipliers
- 48-bit accumulators, pre-adders
- Used for filters, FFTs, matrix operations
- Much faster and more efficient than LUT-based arithmetic

4. Block RAM (BRAM)

Dedicated memory blocks (18Kb-36Kb typical).

- True dual-port capability
- Configurable width/depth
- Optional output registers
- Used for FIFOs, buffers, lookup tables

Other Components:

- **Clock management:** PLLs, MMCMs, clock buffers
- **I/O blocks:** Serializers, DDR registers, differential pairs
- **Interconnect:** Programmable routing matrix

8. What is the purpose of timing constraints in FPGA design? Explain the difference between setup and hold constraints.

Purpose of Timing Constraints

Timing constraints guide the synthesis and place-and-route tools to meet design performance requirements. They define:

- Clock frequencies and relationships
- Input/output delays relative to clocks
- Exceptions (false paths, multicycle paths)

- Clock domain crossing requirements

Without proper constraints, tools optimize for area/resources rather than timing, leading to failures in hardware.

Setup Time Constraint

Setup time (Tsu): Minimum time data must be stable **before** the clock edge for reliable capture.

The data path delay must satisfy:

Tclk > Tco + Tlogic + Troute + Tsu - Tskew

- Violated when combinational paths are too long
- Fixed by: pipelining, reducing logic depth, improving placement

Hold Time Constraint

Hold time (Th): Minimum time data must remain stable **after** the clock edge.

The data path delay must satisfy:

Tco + Tlogic + Troute > Th + Tskew

- Violated when paths are too short (common with clock skew)
- Fixed by: adding delay buffers, adjusting placement, fixing clock distribution

Key Difference:

Setup violations prevent meeting target frequency; hold violations cause functional failures regardless of frequency.

9. Explain the concept of resource sharing in FPGA design and when it should be applied.

Resource Sharing Overview

Resource sharing is the technique of reusing the same hardware block (multiplier, adder, ALU) for multiple operations that don't occur simultaneously. This reduces area at the cost of potentially increased latency or reduced throughput.

When to Apply Resource Sharing:

- **Time-multiplexed operations:** Multiple calculations using the same resources at different times
- **Resource-constrained designs:** Limited DSP blocks or logic resources
- **Low-throughput applications:** Where latency is acceptable
- **Power optimization:** Fewer active resources reduce dynamic power

Common Shared Resources:

- Multipliers and DSP blocks
- Large adders/subtractors
- Dividers (expensive in FPGAs)
- Memory interfaces
- Complex arithmetic units (CORDIC, square root)

Implementation Example:

```
// Shared multiplier
always @(posedge clk)
  case (sel)
    2'b00: result <= a * b;
    2'b01: result <= c * d;
    2'b10: result <= e * f;
  endcase
```

Trade-offs:

- **Advantages:** Reduced area, lower cost, less power
- **Disadvantages:** Increased latency, control complexity, potential throughput reduction

Avoid when: Operations must occur in parallel or timing is critical.

10. What are the differences between simulation, synthesis, and implementation in the FPGA design flow?

1. Simulation (Verification)

Testing the functional correctness of RTL code using testbenches and simulators (ModelSim, VCS, Xcelium).

- **Types:** Behavioral simulation (pre-synthesis), functional simulation (post-synthesis), timing simulation (post-implementation)
- **Purpose:** Verify logic correctness, protocol compliance, edge cases
- **Does not consider:** Actual hardware resources or timing (in behavioral sim)
- **Output:** Waveforms, coverage reports, functional verification

2. Synthesis

Converting RTL code (Verilog/VHDL) into a gate-level netlist using FPGA primitives (LUTs, FFs, DSPs, BRAMs).

- **Tools:** Vivado Synthesis, Quartus Synthesis, Synplify
- **Purpose:** Translate high-level descriptions to hardware primitives
- **Optimizations:** Logic minimization, resource mapping, FSM encoding
- **Output:** Netlist, resource utilization report, initial timing estimates
- **Constraints used:** Clock definitions, synthesis directives

3. Implementation (Place and Route)

Mapping synthesized netlist to physical FPGA resources and routing connections.

- **Stages:** Placement → Routing → Bitstream generation
- **Purpose:** Create physical design meeting timing and resource constraints
- **Output:** Bitstream file, timing reports, power analysis
- **Considers:** Actual delays, routing congestion, physical constraints

Critical Flow:

RTL → Simulate → Synthesize → Simulate → Implement → Timing Analysis → Bitstream → Hardware Testing

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How would you implement a hardware-efficient FIFO buffer in an FPGA, and what are the key considerations for depth and width?

Hardware FIFO Implementation

A FIFO (First-In-First-Out) buffer in FPGA uses **dual-port RAM** with separate read and write pointers. Key considerations include:

- **Depth:** Must be a power of 2 for efficient address wrapping using pointer rollover
- **Width:** Should match data path width to avoid serialization overhead
- **Empty/Full flags:** Generated by comparing read and write pointers
- **Clock domains:** Async FIFOs require Gray code counters for CDC

```
always @(posedge clk) begin
  if (wr_en && !full)
    mem[wr_ptr] <= wr_data;
  if (rd_en && !empty)
    rd_data <= mem[rd_ptr];
end
assign full = (wr_ptr + 1 == rd_ptr);
assign empty = (wr_ptr == rd_ptr);
```

Time Complexity: O(1) for read/write operations. **Space Complexity:** O(n) where n is FIFO depth.

2. Explain how you would implement an LRU cache in hardware for a memory controller. What data structures would you use?

Hardware LRU Cache Design

An LRU (Least Recently Used) cache in FPGA requires efficient tracking of access order:

- **CAM (Content Addressable Memory):** For parallel tag matching
- **Age counters:** Each cache line has a counter incremented on access
- **Priority encoder:** Finds the line with minimum age for eviction
- **Valid bits:** Track occupied cache lines

```
always @(posedge clk) begin
  if (cache_hit)
    age[hit_idx] <= max_age;
  else begin
    age[lru_idx] <= max_age;
    tag[lru_idx] <= new_tag;
  end
  age <= age - 1; // Decrement all
end
```

Time Complexity: O(1) for hit detection using CAM, O(n) for LRU victim selection. **Alternative:** Use pseudo-LRU tree for O(log n) complexity with less hardware.

3. How would you design a priority queue in hardware for packet scheduling? Compare different implementation approaches.

Hardware Priority Queue Implementations

Several approaches exist for FPGA-based priority queues:

- **Systolic array:** Parallel comparison tree, O(1) insertion/extraction but high resource usage
- **Shift register with comparators:** Sorted insertion, O(n) time but simple
- **Hierarchical bitmap:** Multiple priority levels with bitmaps, O(log n) complexity
- **Calendar queue:** Time-wheel approach for time-based priorities

```
// Hierarchical bitmap approach
always @(posedge clk) begin
  bitmap_l1[priority[7:4]] <= 1;
  bitmap_l2[priority] <= 1;
end
assign highest = {find_first(bitmap_l1),
                  find_first(bitmap_l2)};
```

Best choice: Hierarchical bitmap for packet schedulers - balances resource usage with O(log n) performance.

4. Describe an efficient algorithm to detect cycles in a linked-list structure used for buffer management in

hardware. How would you implement it?

Floyd's Cycle Detection in Hardware

The **Floyd's tortoise and hare algorithm** is hardware-friendly for cycle detection:

- **Two pointers:** Slow pointer advances 1 step, fast pointer advances 2 steps
- **Cycle detection:** If pointers meet, a cycle exists
- **Hardware advantage:** Only requires 2 pointer registers and comparison logic
- **Deterministic latency:** Completes in at most 2n cycles for list of length n

```
always @(posedge clk) begin
  if (detect_en) begin
    slow_ptr <= mem[slow_ptr].next;
    fast_ptr <= mem[mem[fast_ptr].next].next;
    cycle_detected <= (slow_ptr == fast_ptr);
  end
end
```

Time Complexity: O(n). **Space Complexity:** O(1) - only 2 pointers needed, making it ideal for resource-constrained FPGAs.

5. How would you implement a hash table for fast packet classification in an FPGA? What collision resolution strategy is most hardware-efficient?

Hardware Hash Table Design

FPGA hash tables for packet classification require careful design:

- **Hash function:** CRC or XOR-folding for low latency (1-2 cycles)
- **Collision resolution:** Cuckoo hashing preferred - guarantees O(1) lookup
- **Multiple tables:** 2-4 parallel tables with different hash functions
- **Block RAM utilization:** Map hash buckets to BRAM for efficient storage

```
// Cuckoo hash lookup
assign addr1 = hash1(key);
assign addr2 = hash2(key);
assign match = (table1[addr1] == key) ||
               (table2[addr2] == key);
assign hit = match & valid;
```

Why Cuckoo hashing: Worst-case O(1) lookup vs. O(n) for chaining. Insertion is O(1) amortized. **Alternative:** Linear probing for simpler logic but variable latency.

6. Explain how to implement a sliding window algorithm for network traffic monitoring in hardware. What optimizations can you apply?

Hardware Sliding Window Implementation

Sliding window algorithms for traffic monitoring require efficient state management:

- **Circular buffer:** Store recent packet timestamps/counts
- **Running sum:** Maintain cumulative statistics, subtract expired entries
- **Time-based indexing:** Use timestamp modulo window size for addressing
- **Lazy expiration:** Remove old entries only when needed to reduce logic

```
always @(posedge clk) begin
  window_sum <= window_sum + new_val
                - buffer[wr_ptr];
  buffer[wr_ptr] <= new_val;
  wr_ptr <= (timestamp % WINDOW_SIZE);
  avg <= window_sum >> LOG2_SIZE;
end
```

Optimization: Use power-of-2 window sizes for modulo operations to become simple bit masking. **Time Complexity:** O(1) per update.

7. How would you find pairs in a data stream that sum to a target value using hardware parallelism? Optimize for throughput.

Parallel Pair Sum Detection

Hardware can exploit parallelism for pair sum problems:

- **Dual-port RAM lookup:** Store seen values, check if (target - current) exists
- **Parallel CAM:** Search all stored values simultaneously
- **Bloom filter pre-check:** Fast negative confirmation before RAM lookup
- **Pipeline stages:** Hash computation, lookup, and result generation

```
// Single-cycle pair detection
always @(posedge clk) begin
  complement = target - stream_data;
  cam_match = |cam_hit[complement];
```

```
  pair_found <= cam_match & valid;
  cam[stream_data] <= 1'b1;
end
```

Throughput: One pair check per cycle using CAM. **Space Complexity:** O(n) for storing seen values. **Alternative:** Use hash table for larger value ranges where CAM is impractical.

8. Design a hardware-efficient sorting network for sorting 8 packet priorities. What is the optimal comparator count?

Sorting Network for Packet Priorities

For sorting 8 elements, use a **Batcher's odd-even mergesort network**:

- **Comparator count:** 19 comparators (optimal for 8 elements)
- **Depth:** 6 stages for fully pipelined operation
- **Throughput:** One sorted result per cycle after pipeline fill
- **Latency:** 6 cycles fixed, deterministic timing

```
// Stage 1: 4 parallel comparators
compare_swap(d[0],d[1], s1[0],s1[1]);
compare_swap(d[2],d[3], s1[2],s1[3]);
// ... stages 2-6
// Each compare_swap is:
assign {max,min} = (a>b) ? {a,b}:{b,a};
```

Advantages: Fixed latency, fully parallel, no control logic. **Resource usage:** 19 comparators × data width. Better than bubble sort $O(n^2)$ comparisons.

9. How would you implement a Trie data structure in FPGA for IP address lookup? What are the memory access patterns?

Hardware Trie for IP Lookup

A Trie (prefix tree) for IP lookup requires careful memory organization:

- **Multi-bit Trie:** Use 4-8 bit strides to reduce tree depth
- **Memory layout:** Store nodes in block RAM with next-level pointers
- **Pipeline stages:** One lookup per stride level (4-8 stages for IPv4)
- **Leaf compression:** Store results at any node for longest prefix match

```
always @(posedge clk) begin
  stride = ip_addr[bit_pos +: STRIDE];
  node_addr <= node[stride].next_ptr;
  if (node[stride].valid)
    result <= node[stride].data;
  bit_pos <= bit_pos + STRIDE;
end
```

Memory accesses: Sequential, one per pipeline stage. **Optimization:** Use 8-bit stride for 4 lookups vs 32 for 1-bit. **Throughput:** One lookup per cycle pipelined.

10. Explain how to implement a rate limiter using the token bucket algorithm in hardware. How do you handle token accumulation efficiently?

Hardware Token Bucket Rate Limiter

Token bucket algorithm controls bandwidth with burst allowance:

- **Token counter:** Accumulates at fixed rate up to bucket capacity
- **Time-based increment:** Use timer to add tokens periodically
- **Packet admission:** Deduct tokens equal to packet size
- **Overflow handling:** Saturating counter prevents token overflow

```
always @(posedge clk) begin
  if (timer_tick)
    tokens <= min(tokens+RATE, BURST);
  if (pkt_valid && tokens >= pkt_size) begin
    tokens <= tokens - pkt_size;
    admit <= 1'b1;
  end else admit <= 1'b0;
end
```

Efficiency: Incremental update avoids multiplication. **Time Complexity:** O(1) per packet. **Precision:** Timer resolution determines rate granularity. Supports burst up to BURST tokens.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a high-performance packet processing pipeline for a network switch FPGA. What are the key architectural considerations?

Packet Processing Pipeline Architecture

A high-performance network switch FPGA design requires careful consideration of throughput, latency, and resource utilization:

- **Multi-stage pipeline architecture:** Parse, classify, forward, modify, and transmit stages with balanced latency
- **Memory hierarchy:** Use on-chip BRAM for lookup tables (MAC/IP tables), external DDR for packet buffering, and TCAM for complex rule matching
- **Parallel processing:** Multiple packet processing engines to handle line-rate traffic (e.g., 100Gbps requires ~148Mpps for minimum-sized packets)
- **Flow control and backpressure:** Credit-based or ready/valid handshaking between pipeline stages
- **Clock domain crossing:** Proper CDC techniques for interfaces running at different rates

Key metrics: Target wire-speed processing with deterministic latency (<1µs for cut-through), implement QoS with priority queuing, and ensure graceful degradation under congestion.

Resource optimization: Use block RAM efficiently, implement hash-based lookup tables, and consider HLS for complex control logic while keeping datapath in RTL.

2. How would you design a video processing system on FPGA that performs real-time 4K video scaling, rotation, and color space conversion?

Real-Time Video Processing Architecture

For 4K@60fps video processing (8.3 million pixels/frame, 498 Mpixels/sec), the design must meet strict timing requirements:

- **Streaming architecture:** Use AXI-Stream interfaces with line buffers for row-based processing to minimize memory bandwidth
- **Scaling engine:** Implement polyphase FIR filters with configurable tap counts; use separable 2D filtering (horizontal then vertical) to reduce multiplier usage
- **Rotation:** For 90°/270° rotation, use external DDR with burst transactions; for arbitrary angles, implement bilinear interpolation with coordinate transformation
- **Color space conversion:** Matrix multiplication pipeline (RGB to YCbCr): Y = 0.299R + 0.587G + 0.114B using fixed-point arithmetic
- **Memory management:** Dual-port frame buffers, DMA engines with scatter-gather for efficient transfers

Timing closure: Pipeline critical paths, target 300MHz+ for pixel clock domain, use multi-cycle paths where appropriate.

Resource estimation: ~500 DSP blocks for filters, ~10Mb BRAM for line buffers, external DDR bandwidth ~15GB/s for 4K60.

3. Design a fault-tolerant FPGA system for aerospace applications. What redundancy and error detection mechanisms would you implement?

Fault-Tolerant FPGA Design

Aerospace applications require robust error detection and mitigation strategies due to radiation-induced SEUs (Single Event Upsets):

- **Triple Modular Redundancy (TMR):** Replicate critical logic three times with majority voting; apply to state machines, configuration registers, and control paths
- **Scrubbing:** Continuous configuration memory scrubbing to detect and correct bitflips; implement external scrubber or use internal SEM IP
- **ECC protection:** Enable ECC on all block RAMs and external memory interfaces; implement custom ECC for critical data paths
- **Watchdog timers:** Multiple independent watchdogs monitoring different subsystems with automatic recovery
- **Safe state design:** Ensure system enters known safe state on error detection; implement graceful degradation

Implementation example:

```
// TMR voting logic
always @(posedge clk) begin
  voted_output <= (moduleA & moduleB) |
                  (moduleB & moduleC) |
                  (moduleA & moduleC);
end
```

Verification: Fault injection testing, radiation testing, and formal verification of safety-critical paths.

4. How would you architect a software-defined radio (SDR) on FPGA with multiple simultaneous channel processing?

Multi-Channel SDR Architecture

A flexible SDR platform requires reconfigurable signal processing chains and efficient resource sharing:

- **Front-end processing:** ADC interface with DDC (Digital Down Converter) using NCO and CIC filters for decimation; support multiple carrier frequencies simultaneously
- **Channelization:** Polyphase filter bank or FFT-based channelizer to separate frequency bands; each channel gets independent processing chain
- **Modulation/demodulation:** Configurable CORDIC engines for phase/frequency processing; support multiple standards (QPSK, QAM, OFDM)
- **Time-multiplexed processing:** Share expensive resources (multipliers, FFT cores) across channels using TDM with sufficient throughput margin
- **Control plane:** Soft processor (MicroBlaze/NIOS) or hard ARM core for configuration, protocol handling, and MAC layer

Resource allocation example: For 8 channels at 20MHz each, use shared 2048-point FFT (300MHz operation rate), dedicated CIC filters per channel, and time-multiplexed FIR filters.

Interface: High-speed links (PCIe, 10GbE) for baseband I/Q data transfer to host processor for higher-layer processing.

5. Design a hardware accelerator for convolutional neural network inference on FPGA. How would you optimize for throughput and energy efficiency?

CNN Hardware Accelerator Design

Optimizing CNN inference requires balancing compute, memory bandwidth, and energy:

- **Systolic array architecture:** 2D array of PEs (Processing Elements) for matrix multiplication; dimension based on available DSP blocks (e.g., 16x16 for mid-range FPGAs)
- **Dataflow optimization:** Choose between weight-stationary, output-stationary, or row-stationary dataflow based on layer characteristics
- **Quantization:** Use INT8 or INT4 fixed-point arithmetic instead of FP32; reduces DSP usage by 4-8x and memory bandwidth proportionally
- **On-chip buffering:** Tiling strategy to fit activations and weights in BRAM; minimize external memory accesses (highest energy cost)
- **Layer fusion:** Combine Conv-BN-ReLU into single pass to avoid intermediate memory writes

PE structure example:

```
// Processing Element
always @(posedge clk) begin
  partial_sum <= partial_sum + (weight * activation);
  if (accumulate_done)
    output_data <= relu(partial_sum + bias);
end
```

Performance: Target 1000+ GOP/s for INT8 operations; memory bandwidth reduction through compression and sparsity exploitation.

6. How would you design a high-speed data acquisition system with 1 GSPS sampling rate and real-time signal processing capabilities?

High-Speed Data Acquisition Architecture

A 1 GSPS system requires careful attention to timing, signal integrity, and data throughput:

- **ADC interface:** Use LVDS or JESD204B for multi-Gbps serial data; implement 8b10b decoding, lane alignment, and frame synchronization
- **Clock architecture:** Ultra-low jitter clock distribution (<100fs RMS); use PLLs with external VCXO for phase noise optimization
- **Data processing pipeline:** Immediate decimation/filtering to reduce data rate; implement FIR filters in polyphase decomposition for efficiency
- **Triggering system:** Hardware-based pattern matching and threshold detection with <10ns latency; circular buffer for pre-trigger capture
- **Data streaming:** PCIe Gen3 x8 (64 Gbps) or multiple 10GbE links for sustained throughput; implement scatter-gather DMA

Memory subsystem: Use DDR4 with interleaved banks for sustained write bandwidth; implement write combining and burst optimization.

Signal processing: FFT engines for spectral analysis (use streaming FFT for continuous operation), digital filtering, and real-time statistics calculation.

Latency budget: ADC to processing: <20ns, trigger decision: <50ns, data to host: <1µs.

7. Design a cryptographic accelerator for AES-256 encryption/decryption with support for multiple modes of operation. What are the performance and security considerations?

Cryptographic Accelerator Design

Implementing secure and high-performance AES requires both algorithmic optimization and protection against side-channel

attacks:

- **Core architecture:** Pipelined or iterative implementation; pipelined achieves 1 block/cycle after initial latency but uses more resources
- **Mode support:** ECB, CBC, CTR, GCM; CTR and GCM allow parallelization for higher throughput
- **Key expansion:** On-the-fly key schedule generation vs. pre-computed round keys; trade-off between latency and memory
- **SubBytes optimization:** Use BRAM-based S-box lookup tables or composite field arithmetic for area efficiency
- **Security hardening:** Constant-time implementation to prevent timing attacks; balanced routing to mitigate power analysis (DPA/SPA)

AES round structure:

```
// Single AES round
assign after_subbytes = sbox[state];
assign after_shiftrows = shift_rows(after_subbytes);
assign after_mixcols = mix_columns(after_shiftrows);
assign round_output = after_mixcols ^ round_key;
```

Performance: Pipelined design achieves 10+ Gbps throughput at 250MHz; GCM mode adds authentication with minimal overhead.

8. How would you design a PCIe-based FPGA accelerator card with DMA capabilities and host-device communication?

PCIe Accelerator Card Architecture

Designing an efficient PCIe accelerator requires understanding the PCIe protocol stack and optimizing data movement:

- **PCIe endpoint:** Use hard IP core (Gen3 x8 or Gen4 x16); implement AXI-MM bridge for internal fabric connection
- **DMA engine:** Scatter-gather DMA with descriptor queues; support multiple concurrent transfers with independent channels
- **Memory mapping:** BAR (Base Address Register) configuration for register access and small data transfers; separate control and data planes
- **Interrupt handling:** MSI-X for efficient interrupt delivery; minimize interrupt rate by using completion queues
- **Data buffering:** Internal FIFOs to handle PCIe burst nature; implement flow control to prevent overflow

DMA descriptor structure:

```
typedef struct {
  uint64_t src_addr;
  uint64_t dst_addr;
  uint32_t length;
  uint32_t control; // flags, channel ID
} dma_descriptor_t;
```

Optimization: Maximize payload size (256B for Gen3), minimize TLP overhead, use relaxed ordering when possible, implement read completion coalescing.

Software interface: Kernel driver with mmap() for register access, ioctl() for control, and async I/O for data transfers.

9. Design a time-sensitive networking (TSN) switch for industrial automation on FPGA. What features and guarantees must be implemented?

TSN Switch Design for Industrial Applications

Time-Sensitive Networking requires deterministic latency and precise time synchronization:

- **IEEE 802.1AS (gPTP):** Implement hardware timestamping at PHY layer (<8ns accuracy); transparent clock or boundary clock functionality
- **IEEE 802.1Qbv (TAS):** Time-Aware Shaper with gate control lists; schedule high-priority traffic in protected windows
- **IEEE 802.1Qbu (Frame Preemption):** Allow express frames to interrupt best-effort traffic; implement fragment reassembly
- **Traffic shaping:** Credit-Based Shaper (CBS) for AVB streams; strict priority queuing with 8 priority levels
- **Latency guarantees:** Bounded worst-case latency through admission control and bandwidth reservation

Timestamping implementation:

```
// Capture timestamp on SFD detection
always @(posedge clk) begin
  if (sfd_detected)
    timestamp <= ptp_counter;
  tx_frame <= {preamble, timestamp, payload};
end
```

Synchronization: PTP slave clock with <1µs accuracy; hardware-assisted correction field update for transparent clock operation.

Determinism: Guaranteed <100µs end-to-end latency for critical control messages in industrial networks.

10. How would you design a radar signal processing system on FPGA for automotive FMCW radar with object detection and tracking?

Automotive FMCW Radar Processing

FMCW radar processing requires real-time FFT processing, CFAR detection, and multi-object tracking:

- **Range FFT:** First FFT stage (1024-4096 points) on each chirp to determine object range; use streaming FFT IP with windowing (Hamming/Blackman)
- **Doppler FFT:** Second FFT across chirps to extract velocity information; creates range-Doppler map
- **CFAR detection:** Cell-Averaging CFAR or OS-CFAR to detect targets above noise floor; implement in sliding window fashion
- **Angle estimation:** Digital beamforming or MUSIC algorithm for azimuth/elevation using multiple RX channels
- **Tracking:** Kalman filter or particle filter for multi-object tracking; predict position/velocity for next frame

CFAR threshold calculation:

```
// CA-CFAR detector
avg_noise = (sum_leading_cells + sum_lagging_cells) /
          (num_cells - num_guard_cells);
threshold = avg_noise * scale_factor;
detection = (test_cell > threshold);
```

Performance requirements: Process 77GHz chirps at 10-20ms frame rate; detect objects 0-200m range, ±60° FOV, velocity ±100 km/h.

Resource usage: Multiple FFT cores (4-8 parallel), CORDIC for angle processing, ~50K LUTs for complete pipeline.

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Verilog module to implement a parameterized FIFO buffer with full and empty flags.

Parameterized FIFO Implementation

Here's a synchronous FIFO with configurable depth and width:

```
module fifo #(parameter DEPTH=8, WIDTH=8)(
  input clk, rst, wr_en, rd_en,
  input [WIDTH-1:0] data_in,
  output reg [WIDTH-1:0] data_out,
  output full, empty
);
  reg [WIDTH-1:0] mem [0:DEPTH-1];
  reg [$clog2(DEPTH):0] wr_ptr, rd_ptr;
  assign full = (wr_ptr[($clog2(DEPTH)-1):0]==rd_ptr[($clog2(DEPTH)-1):0]) && (wr_ptr[$clog2(DEPTH)]!=rd_ptr[$clog2(DEPTH)]);
  assign empty = (wr_ptr == rd_ptr);
endmodule
```

Key features:

- Uses extra bit in pointers to distinguish full from empty
- Parameterized for flexibility
- $clog2 function calculates address width automatically

2. How do you debug timing violations in FPGA designs? What tools and techniques do you use?

Timing Violation Debug Strategy

Primary techniques:

- **Static Timing Analysis (STA):** Use vendor tools (Vivado Timing Summary, Quartus TimeQuest) to identify critical paths
- **Timing reports:** Analyze setup/hold violations, examine slack values, identify failing paths
- **Cross-probing:** Trace critical paths back to RTL source code
- **Constraint review:** Verify clock definitions, I/O delays, false paths, and multicycle paths

Common solutions:

- Pipeline insertion to break long combinational paths
- Register replication for high-fanout nets
- Clock domain crossing (CDC) verification using formal tools
- Floorplanning and placement constraints for critical logic
- Retiming optimization to balance logic stages

Advanced tools: ChipScope/SignalTap for runtime verification, Synopsys PrimeTime for advanced STA

3. Write SystemVerilog code to create a clock divider that generates a clock with 1/N frequency and 50% duty cycle.

Parameterized Clock Divider

Implementation with guaranteed 50% duty cycle:

```
module clk_divider #(parameter DIV=4)(
  input clk_in, rst_n,
  output reg clk_out
);
  reg [$clog2(DIV)-1:0] counter;
  always @(posedge clk_in or negedge rst_n) begin
    if (!rst_n) begin
      counter <= 0; clk_out <= 0;
    end else if (counter == DIV/2-1) begin
      clk_out <= ~clk_out; counter <= 0;
    end else counter <= counter + 1;
  end
endmodule
```

Important notes:

- Works correctly only for even divisors
- For odd divisors, duty cycle won't be exactly 50%
- Generated clock should ideally be used as clock enable, not actual clock

4. Explain metastability in FPGA designs. How do you prevent it and what is a synchronizer chain?

Metastability and CDC Handling

Metastability occurs when:

- A signal changes too close to the clock edge (violates setup/hold time)
- The flip-flop enters an undefined state between 0 and 1
- Most common in Clock Domain Crossing (CDC) scenarios

Synchronizer chain solution:

```
module synchronizer #(parameter STAGES=2)(
  input clk, async_in,
  output sync_out
);
  reg [STAGES-1:0] sync_reg;
  always @(posedge clk)
    sync_reg <= {sync_reg[STAGES-2:0], async_in};
  assign sync_out = sync_reg[STAGES-1];
endmodule
```

Best practices:

- Use minimum 2-stage synchronizers (3 stages for high-speed designs)
- Apply timing constraints: set_false_path or set_max_delay
- Never synchronize multi-bit buses directly (use Gray coding or handshaking)
- Mark synchronizer registers with ASYNC_REG attribute

5. What debugging tools do you use for FPGA designs? Explain integrated logic analyzers like ChipScope and SignalTap.

FPGA Debugging Tools

Integrated Logic Analyzers (ILA):

- **Xilinx ChipScope/ILA:** Captures internal signals at runtime, triggered by conditions
- **Intel SignalTap II:** Similar functionality for Intel/Altera FPGAs
- Minimal resource overhead, captures live data from actual hardware
- Supports complex triggering, cross-probing with design hierarchy

Simulation tools:

- **ModelSim/QuestaSim:** Industry-standard HDL simulators
- **Vivado Simulator:** Integrated with Xilinx toolchain
- **VCS:** High-performance Synopsys simulator
- Waveform viewers for signal analysis

Other techniques:

- **Assertion-based verification:** SystemVerilog Assertions (SVA) for property checking
- **Formal verification:** Tools like JasperGold for exhaustive proof
- **Protocol analyzers:** For debugging communication interfaces
- **LED/GPIO debugging:** Simple but effective for basic issues

6. Write a Verilog module to detect a specific sequence (e.g., 1011) in a serial bit stream using a state machine.

Sequence Detector FSM

Moore machine implementation for detecting '1011':

```
module seq_detect_1011(
  input clk, rst, din,
  output reg detected
);
  typedef enum reg [2:0] {S0,S1,S10,S101,S1011} state_t;
  state_t state, next_state;
  always @(posedge clk or posedge rst)
    if (rst) state <= S0; else state <= next_state;
  always @(*) begin
    case(state)
      S0: next_state = din ? S1 : S0;
      S1: next_state = din ? S1 : S10;
      S10: next_state = din ? S101 : S0;
      S101: next_state = din ? S1011 : S10;
      S1011: next_state = din ? S1 : S10;
    endcase
  end
  assign detected = (state == S1011);
endmodule
```

Features: Overlapping detection, handles continuous streams correctly

7. How do you handle memory profiling and resource utilization analysis in FPGA designs?

FPGA Resource Analysis

Resource utilization metrics:

- **Logic resources:** LUTs (Look-Up Tables), FFs (Flip-Flops), slice count
- **Memory resources:** Block RAM (BRAM), distributed RAM, LUTRAM
- **DSP slices:** Hardware multipliers and MAC units
- **I/O resources:** Pin usage, I/O standards, differential pairs
- **Routing resources:** Interconnect congestion analysis

Analysis tools:

- **Synthesis reports:** Show estimated resource usage before place-and-route
- **Implementation reports:** Actual resource consumption post-PAR
- **Power analysis:** XPE (Xilinx Power Estimator) or PowerPlay
- **Floorplanner:** Visual representation of resource placement

Optimization strategies:

- Use BRAM instead of registers for large memories
- Share DSP blocks across multiple operations
- Resource sharing through multiplexing
- Balance between speed and area using synthesis directives

8. Write a SystemVerilog assertion to verify that a valid signal stays high for exactly 3 clock cycles after a request.

SystemVerilog Assertion (SVA)

Property-based verification for timing requirements:

```
property valid_duration;
  @(posedge clk) disable iff (!rst_n)
  $rose(request) |-> valid[->3] ##1 !valid;
endproperty

assert property (valid_duration)
  else $error("Valid signal duration violation");

// Alternative: exact 3 consecutive cycles
property valid_3cycles;
  @(posedge clk) disable iff (!rst_n)
  $rose(request) |-> valid ##1 valid ##1 valid ##1 !valid;
endproperty
```

Key operators:

- **|->:** Overlapping implication
- **##N:** Delay by N clock cycles
- **[->N]:** Non-consecutive repetition (Nth occurrence)
- **$rose():** Detects 0-to-1 transition

9. Explain the concept of false paths and multicycle paths in FPGA timing constraints. How do you identify and constrain them?

Advanced Timing Constraints

False Paths:

- Paths that never propagate data in actual operation
- Examples: asynchronous resets, test/debug logic, static configuration signals
- **Constraint:** set_false_path -from [get_pins config_reg*] -to [get_pins data_path*]
- Prevents timing analysis on these paths, improves runtime

Multicycle Paths:

- Paths where data takes multiple clock cycles to propagate
- Examples: slow control logic, pipelined datapaths with known latency
- **Constraint:** set_multicycle_path 2 -setup -from [get_clocks clk_a] -to [get_clocks clk_b]
- Must specify both setup and hold multicycle values

Identification techniques:

- Review timing reports for paths with excessive slack or violations
- Analyze control flow and data dependencies in RTL
- Use formal verification to prove functional correctness
- Cross-check with architectural documentation

Best practice: Document all timing exceptions thoroughly and review during design changes

10. Write a Verilog function to calculate the parity bit for error detection and implement a simple error correction code checker.

Parity and Error Detection

Implementation with even parity calculation:

```
module parity_checker #(parameter WIDTH=8)(
  input [WIDTH-1:0] data_in,
  input parity_in,
  output parity_error
);
  function automatic calc_parity;
    input [WIDTH-1:0] data;
    integer i;
    begin
      calc_parity = 0;
      for (i=0; i
```

Advanced ECC: For single-error correction, use Hamming codes or SECDED (Single Error Correction, Double Error Detection) implemented in BRAM controllers

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to debug a complex timing violation in an FPGA design. How did you approach it?

Situation: During a high-speed networking project, our design failed timing closure at 250MHz with setup violations on critical data paths.

Task: I was responsible for identifying the root cause and implementing fixes to meet the timing requirements without compromising functionality.

Action: I analyzed the timing reports to identify the critical paths, used timing constraints analysis to verify SDC files, applied pipeline registers on the longest combinational paths, and utilized physical synthesis optimizations. I also cross-probed the design in Vivado to understand placement issues.

Result: Successfully achieved timing closure with 5% positive slack, reducing the critical path delay by 30% through strategic pipelining and constraint refinement. The design met all performance targets and passed validation.

2. Describe a situation where you had to make a trade-off between resource utilization and performance in an FPGA design.

Situation: I was designing a video processing pipeline for a resource-constrained FPGA (Xilinx Artix-7) where the initial implementation consumed 95% of available DSP blocks and LUTs.

Task: I needed to optimize the design to fit within 70% resource utilization while maintaining real-time processing at 60fps for 1080p video.

Action: I performed resource-performance analysis, replaced some parallel processing blocks with time-multiplexed architectures, optimized filter implementations using distributed arithmetic instead of full multipliers, and implemented clock gating for unused modules. I benchmarked each optimization to ensure performance requirements were maintained.

Result: Reduced resource utilization to 68% (DSP blocks down to 62%, LUTs to 71%) while maintaining the required 60fps throughput. This allowed headroom for future feature additions and reduced power consumption by 18%.

3. Tell me about a time when you had to collaborate with software engineers to integrate FPGA functionality into a larger system.

Situation: Our team was developing a hardware-accelerated machine learning inference system where FPGA acceleration needed to integrate with a Linux-based software stack.

Task: I was responsible for designing the FPGA accelerator and establishing the hardware-software interface, ensuring seamless data transfer and synchronization.

Action: I organized weekly sync meetings with software engineers, designed a memory-mapped AXI interface with clear register maps and documentation, created a detailed interface specification document, implemented DMA transfers for high-throughput data movement, and developed a C API library for the software team. I also created test benches that simulated the software interaction patterns.

Result: The integration was completed two weeks ahead of schedule with zero interface-related bugs in production. The system achieved 15x speedup over CPU-only implementation, and the software team praised the clear documentation and intuitive API design.

4. Describe a challenging project where you had to meet aggressive timing deadlines. How did you manage your time and priorities?

Situation: I was assigned to lead the FPGA development for a new product with a fixed launch date six months away, requiring implementation of a complex signal processing algorithm with no existing reference design.

Task: I needed to complete architecture design, RTL implementation, verification, and hardware validation within the tight timeline while ensuring design quality.

Action: I created a detailed project plan with weekly milestones, prioritized core functionality for early implementation, adopted an agile approach with bi-weekly reviews, parallelized verification by creating testbenches alongside RTL development, and communicated risks early to stakeholders. I also automated regression testing and used version control effectively to manage incremental progress.

Result: Delivered the complete FPGA design one week before the deadline, passing all functional and performance requirements on first silicon. The structured approach and risk management prevented any major surprises, and the design required only minor bug fixes post-launch.

5. Tell me about a time when you identified and prevented a potential design flaw that others missed.

Situation: During a design review for a high-speed data acquisition system, I noticed that the clock domain crossing (CDC) strategy proposed by a colleague had potential metastability issues that weren't caught in initial simulations.

Task: I needed to diplomatically raise the concern, validate my hypothesis, and propose a robust solution without derailing the project timeline.

Action: I created a detailed simulation showing corner cases where the existing CDC implementation could fail, presented the findings with supporting timing analysis and literature references, and proposed using a proper two-stage synchronizer with MTBF calculations. I worked with the original designer to refine the solution and updated the design guidelines to prevent similar issues.

Result: The team adopted the improved CDC architecture, preventing potential field failures. The design passed rigorous stress testing and has been running in production for two years with zero CDC-related issues. The updated guidelines became part of our standard design practices.

6. Describe a situation where you had to learn a new FPGA tool or technology quickly to complete a project.

Situation: Our company decided to switch from Xilinx to Intel FPGAs for a new project due to supply chain constraints, and I had exclusively worked with Xilinx tools for five years.

Task: I needed to become proficient with Intel Quartus Prime and the Intel FPGA architecture within three weeks to maintain the project schedule.

Action: I created a structured learning plan covering Quartus synthesis, timing analysis, and IP integration. I completed Intel's official training modules, ported a small reference design from Vivado to Quartus to understand tool differences, joined Intel FPGA forums and communities for quick problem-solving, and maintained a knowledge document of key differences and gotchas. I also scheduled knowledge-sharing sessions with a colleague who had Intel FPGA experience.

Result: Successfully became productive with Intel tools within two weeks, delivered the first design milestone on time, and created a comprehensive migration guide that helped three other team members transition. The project was completed successfully, and I'm now proficient in both FPGA ecosystems.

7. Tell me about a time when you had to advocate for a technical decision that was initially met with resistance.

Situation: During architecture discussions for a radar signal processing system, I proposed using High-Level Synthesis (HLS) for complex algorithm blocks, but senior engineers preferred traditional RTL due to concerns about performance and resource efficiency.

Task: I needed to demonstrate that HLS could meet our requirements while significantly reducing development time, and convince the team to adopt this approach.

Action: I created a proof-of-concept implementing a critical FFT-based processing block in both HLS (Vitis HLS) and hand-coded Verilog, comparing performance, resource utilization, and development time. I presented quantitative data showing HLS achieved 95% of hand-coded performance with 60% less development time, and demonstrated how HLS pragmas could optimize critical sections. I also addressed concerns about maintainability and debugging.

Result: The team approved using HLS for algorithm-intensive blocks while keeping control logic in RTL. This hybrid approach reduced overall development time by 40%, and the design met all performance targets. The success led to broader HLS adoption across other projects in the organization.

8. Describe a situation where you had to mentor or help a junior engineer overcome a technical challenge.

Situation: A junior engineer on my team was struggling with understanding and implementing proper reset strategies in a multi-clock domain design, causing synthesis warnings and potential initialization issues.

Task: I needed to help them understand reset best practices and guide them to implement a robust solution while building their confidence and skills.

Action: I scheduled one-on-one sessions to explain synchronous vs. asynchronous resets, demonstrated reset synchronizer implementations with timing diagrams, reviewed their code together and asked guiding questions rather than giving direct answers, provided reference designs and documentation, and encouraged them to present their final solution to the team. I also shared resources on clock domain crossing and reset strategies.

Result: The junior engineer successfully implemented a proper reset architecture that eliminated all warnings and passed verification. They gained confidence in handling clock domain issues and later became the go-to person for CDC reviews. This mentoring approach strengthened team capability and improved overall design quality.

9. Tell me about a time when you had to optimize an FPGA design for power consumption.

Situation: I was working on a battery-powered IoT edge device where the FPGA-based sensor processing was consuming excessive power, reducing battery life below acceptable levels (target was 72 hours, achieving only 36 hours).

Task: I needed to reduce FPGA power consumption by at least 40% without compromising processing capabilities or increasing latency.

Action: I analyzed power reports to identify high-consumption modules, implemented clock gating for idle processing blocks, reduced clock frequencies for non-critical paths, optimized state machines to minimize switching activity, utilized block RAM power-down modes, and migrated from distributed logic to more power-efficient hard IP blocks where possible. I used Xilinx Power Estimator to validate improvements iteratively.

Result: Achieved 47% power reduction, extending battery life to 78 hours, exceeding the target. The optimizations included: 30% reduction from clock gating, 10% from frequency optimization, and 7% from RAM power management. The product successfully launched and received positive customer feedback on battery performance.

10. Describe a time when you had to handle a critical bug found late in the development cycle or in production.

Situation: Two weeks before product launch, our customer discovered a rare data corruption issue in the FPGA-based network packet processor that occurred only under specific high-load conditions with particular packet sequences.

Task: I needed to quickly identify the root cause, implement a fix, and ensure comprehensive verification without delaying the launch.

Action: I immediately reproduced the issue using customer-provided traffic patterns, used ChipScope/ILA to capture internal signals during failure conditions, identified a race condition in the buffer management logic when simultaneous read/write occurred at boundary conditions, implemented a fix using proper handshaking and added pipeline stages, created targeted regression tests for the specific failure scenario, and performed 72-hour stress testing with randomized traffic patterns.

Result: Identified and fixed the bug within 4 days, completed verification in 3 days, and delivered the updated bitstream on schedule. No further issues were reported in production. I also updated our verification methodology to include more comprehensive corner-case testing, preventing similar issues in future projects.