

A field guide to production trading
systems

BARE-METAL FPGA ACCELERATION FOR C++ TRADING SYSTEMS

**HDL Design, PCIe/DMA Transfer, and Clock-
Domain Synchronization for Sub-Microsecond
Execution.**

Weston Ashgrove

Copyright © 2026 by Weston Ashgrove

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This book is intended for informational and educational purposes only. While every effort has been made to ensure the accuracy and completeness of the information contained herein, the author and publisher make no representations or warranties with respect to the accuracy or completeness of the contents and specifically disclaim any implied warranties. The hardware designs, HDL code, and design patterns described in this book are provided for educational purposes and should be independently verified through simulation and testing before deployment on physical hardware or in any production trading environment.

Working with FPGA hardware and PCIe interfaces carries inherent risk of hardware damage or system instability if design constraints, timing requirements, or manufacturer specifications are not followed correctly. Readers are responsible for consulting their specific hardware documentation and following all manufacturer safety guidelines before implementing any design described in this book.

All trademarks, service marks, and trade names referenced in this book are the property of their respective owners and are used for identification purposes only. Use of these names does not imply endorsement.

First Edition: 2026

What You'll Build

By the end of this book, you will have built an FPGA-accelerated order gateway that moves the hottest part of the execution path, from wire to order decision, into hardware for deterministic sub-microsecond latency. Starting from HDL fundamentals taught from first principles, you will design a wire-to-logic ingestion path, define a fixed-latency interface boundary with a software matching engine, build a PCIe and DMA transfer path that avoids CPU involvement on the hot path, and resolve the clock-domain synchronization issues that arise when hardware and software must communicate without adding jitter. By the final chapters, the design is verified through timing closure analysis, hardware-in-the-loop testing, and a direct latency comparison against a pure software baseline. You will finish with a working reference HDL design and a clear understanding of exactly where the boundary between software and hardware should sit in a real trading system.

Prerequisites

You should be comfortable with C++ at a working level and understand basic digital logic concepts, including gates, flip-flops, and clocking. No prior HDL experience is required; Verilog and SystemVerilog are taught from first principles in this book, mapped to familiar C++ analogies where useful. Book 1, *Deterministic Matching Engines in C++*, is a useful reference for the matching logic being accelerated, but its internals are treated as a black box here and are not required reading.

How to Use This Book: The Latency Budget Ledger

Every chapter in this book closes with a Latency Budget Ledger, a short running record of the nanoseconds moved from software into hardware at that stage of the design.

This exists because hardware acceleration is not a single leap from slow to fast. It is a sequence of deliberate decisions about exactly what moves into hardware, what stays in software, and what that split costs or saves at each boundary. By the end of the book, the ledger becomes a complete record of the full wire-to-decision path, showing exactly how much latency was removed at each stage of the design and why.

Each entry follows the same short format:

- **Where:** the stage of the design being added or refined
- **Moved to hardware:** what moved from software into the FPGA at this stage
- **Latency saved:** the measured reduction from the previous stage

By Chapter 11, the ledger doubles as a complete record of the full software-to-hardware latency migration, ready to reference when deciding what else might be worth accelerating.

PART I

Hardware Foundations for Software Engineers

Chapter 1

Why Software Alone Hits a Latency Floor

1.1 Where the nanoseconds go in a pure software execution path

A team spends four months rewriting their order gateway from Python to C++, benchmarks it, and finds they've gone from 40 microseconds tick-to-trade down to 8. They ship it, feel good about it, and then watch a competitor with a similar strategy consistently get filled first on the same signal. The C++ rewrite was real work and it was not wasted, but it ran into a floor that no amount of further software optimization was going to move. The floor wasn't in their code. It was in everything underneath their code that they didn't write and couldn't touch.

Walk the actual path a packet takes and the picture gets uncomfortable fast. A market data update leaves the exchange, crosses the wire, and hits your network interface card. From there it has to cross the PCIe bus into host memory, get picked up by the NIC driver, traverse the kernel's network stack, get copied into a user-space buffer, and only then does your application code start doing anything a profiler would show you. Even with a tuned setup, kernel bypass frameworks, and a NIC that supports it, you're looking at something in the range of 1 to 3 microseconds before your parser touches a single byte. On a stock Linux network stack without bypass, that number is closer to 10 to 30 microseconds, and most of it is spent in places your flame graph never visits, because the time is lost to interrupt handling, context switches, and buffer copies that happen below the abstraction your tools measure.

Then there's the CPU itself, which behaves nothing like the mental model most C++ engineers carry around. A cache miss that goes all the way to main memory costs on the order of 100 to 300 nanoseconds, and a tight matching

engine loop touching an order book that doesn't fit in L2 will eat several of these per message. Branch mispredictions cost 10 to 20 cycles each, cheap in isolation, expensive when your order book logic has conditional paths that depend on runtime data the branch predictor can't learn. None of this shows up as a function call in your profiler. It shows up as your function taking longer than the instruction count says it should, and most engineers chase that mystery by rewriting the function instead of looking at what the function is waiting on. Operating system scheduling adds its own tax, and this is the one that variance-sensitive strategies get hurt by most. Even a thread pinned to an isolated core and running at real-time priority will occasionally lose the core to a hardware interrupt or a kernel housekeeping task. These events are individually rare, in the range of once every few hundred milliseconds to a few seconds depending on how aggressively the box has been tuned, but when one lands during your hot path, it doesn't cost you 100 nanoseconds. It costs you tens of microseconds, sometimes more, and that single event will dominate your tail latency distribution regardless of how fast your median case runs. A trading system that looks excellent at p50 and falls apart at p99.9 is almost always describing this exact failure mode, not a logic problem.

Put the whole path together, wire to decision, on a well-tuned kernel-bypass C++ system, and a realistic budget looks something like this: 1 to 3 microseconds getting data off the wire into user space, 200 to 500 nanoseconds parsing and validating the message, 300 to 800 nanoseconds updating book state and running strategy logic, and then the same 1 to 3 microseconds again going back out to the wire. Call it 3 to 7 microseconds end to end for a system that has already had every obvious software optimization applied. That's not a bad number. It's also not sub-microsecond, and no further C++ tuning is going to make the PCIe round trip or the kernel's interrupt handling disappear, because those costs aren't in your code. They're in the boundary between your code and the physical wire, and that boundary is exactly what the rest of this book exists to move.

Latency budget ledger — chapter 1.1

Baseline established: 3 to 7 microseconds wire-to-decision for a fully optimized pure-software C++ path, with the largest fixed costs sitting outside application code, in NIC-to-host transfer (1 to 3 μ s round trip) and OS-induced tail events (tens of microseconds, low frequency but dominant at p99.9). This is the number every FPGA design decision in the rest of the book gets measured against.

1.2 What FPGAs can and cannot improve, set against realistic expectations

A firm spends eight months and a meaningful fraction of their engineering budget building an FPGA-accelerated gateway, expecting it to cut their strategy's overall latency by an order of magnitude. It ships, it works, and their tick-to-trade number improves by roughly 40 percent. The FPGA team did excellent work. The gap between what they delivered and what leadership expected came from a planning failure, not an engineering one: nobody had mapped out which parts of the latency budget an FPGA can actually touch before the project was funded.

An FPGA wins by removing the parts of the software path that are fixed overhead regardless of how well your code is written. Market data ingestion is the clearest case. A packet arriving on a NIC has to cross PCIe, get picked up by a driver, traverse the kernel stack or a bypass library, and land in a buffer before your C++ parser sees the first byte, and that whole sequence costs 1 to 3 microseconds even in a tuned kernel-bypass setup. An FPGA sitting directly on the wire parses the packet in fabric as the bits arrive, with no bus crossing, no driver, and no context switch involved, and that step drops to tens of nanoseconds. Fixed-format message parsing, checksum validation, and simple filtering all move from a software cost measured in hundreds of nanoseconds to a hardware cost measured in single-digit nanoseconds, because the logic runs in parallel across dedicated gates instead of sequentially through an instruction pipeline. Deterministic, low-jitter execution is the other real win. Software running on a general-purpose OS is at the mercy of the scheduler, and even a well-isolated core will occasionally lose to an interrupt or a housekeeping task, producing a tail latency spike that can be an order of magnitude worse than the median. Hardware logic has no scheduler. The same operation takes the same number of clock cycles every time, which is why FPGA designs are judged on tail latency and jitter, not just median speed.

What an FPGA does not fix is anything that requires genuine decision complexity. A matching engine evaluating multiple order types against a book with thousands of price levels, a risk engine running a VaR calculation across a portfolio, a strategy weighing several correlated signals before deciding to act: none of this maps cleanly onto fixed-function hardware. You can implement complex sequential logic in an FPGA, but every added branch and every added state costs you fabric area and timing margin, and past a certain complexity threshold you're better off keeping that logic in software and accepting the software latency for it. The honest way to think about it is that an FPGA is exceptional at doing one narrow, well-defined thing extremely fast and extremely consistently, and gets progressively worse value the more general-purpose you ask it to be. This is also why FPGA development timelines routinely surprise teams coming from software. A change that would be a ten-line diff and a five-minute CI run in C++ can mean a multi-hour resynthesis and a new round of timing closure in hardware, and that iteration cost shapes

what you choose to put on the FPGA in the first place, not just how you implement it.

Set realistic expectations against that baseline. If your pure-software wire-to-decision path sits at 3 to 7 microseconds, moving market data ingestion and a bounded, well-defined slice of decision logic onto an FPGA typically gets you into the 500 nanosecond to 1.5 microsecond range for that same path, an improvement of roughly 3 to 6x rather than 10x, and most of what remains is the genuinely complex decision logic still living in software along with the PCIe round trip back out to the wire. Teams that go in expecting sub-microsecond total latency from day one are usually measuring against a marketing number from a vendor's best-case demo, not against what a production matching pipeline with real order types and real risk checks will actually deliver. The gains are real and they compound with every other technique in this series, but they show up at the boundary, not throughout the whole system, and Chapter 1.3 is where we draw that boundary precisely enough to build against it.

Latency budget ledger — chapter 1.2

FPGA-addressable segment identified: NIC-to-parsed-message latency drops from 1 to 3 μ s (software, kernel-bypass) to tens of nanoseconds in fabric, and tail jitter from OS scheduling effects is eliminated for anything running entirely in hardware. Realistic full-path target after offloading ingestion and a bounded decision slice: 500 ns to 1.5 μ s, down from the 3 to 7 μ s software baseline. Complex decision logic remains in software by design, not as a shortfall.

1.3 Defining the interface boundary between software and hardware for this build

A team gets the parsing and the risk checks working in fabric, feels good about the demo, and then spends three weeks debugging a production issue that turns out to be a race condition at the handoff between the FPGA and the software matching engine. Neither side was wrong on its own. The FPGA delivered a decision exactly when its logic said it should, and the software consumed it exactly when its logic said it should, but nobody had written down, in enough precision to catch this in review, what "ready" meant across that boundary. This is the failure mode that defines the rest of this book, and it shows up more often at the interface than inside either side.

The boundary has to be drawn by what each side is actually good at, not by what feels architecturally clean. Fabric wins on anything fixed-format, fixed-latency, and free of branching complexity: parsing a known message layout, validating a checksum, filtering by instrument ID, applying a small number of hard-coded risk limits like a notional cap or a fat-finger bound. Software wins on anything that needs state spanning many messages, floating-point-heavy

computation, or decision logic with enough branches that hardware implementation would cost more fabric and more timing margin than the latency saved is worth. A matching engine walking a multi-level order book against several order types is squarely on the software side of that line, not because hardware can't technically do it, but because the fabric cost and the iteration cost of getting it right in HDL exceed what a well-written C++ implementation running at a few hundred nanoseconds already delivers. Draw the boundary as close to the wire as the workload allows, and no closer than that.

Once the boundary is drawn, it has to be specified as a contract, not as a convention two teams happen to agree on informally. That contract has three parts, and skipping any one of them is what produces the three-week bug from the opening.

First, the wire format: an exact byte layout for whatever crosses from FPGA to software, no variable-length fields, no padding left to compiler discretion.

cpp

```
// Fixed-width, no padding left to compiler discretion – every
// field offset is load-bearing
#pragma pack(push, 1)
struct ParsedMarketEvent {
    uint64_t seq_no;           // monotonic, consumer detects
    // gaps/wraps against this
    uint64_t hw_timestamp;    // FPGA-side capture time, ns since
    // epoch
    uint32_t instrument_id;
    uint32_t price_ticks;     // fixed-point, not float – hardware
    // doesn't do float cheaply
    int32_t qty;
    uint8_t side;             // 0 = bid, 1 = ask
    uint8_t msg_type;
    uint8_t _reserved[2];    // explicit padding, not implicit –
    // keeps the struct 32B aligned
};
#pragma pack(pop)
static_assert(sizeof(ParsedMarketEvent) == 32, "wire format
changed, bump protocol version");
```

The `static_assert` is not defensive boilerplate. It fails the build the moment someone on either the HDL side or the software side changes a field width without updating the other, which is exactly the class of mismatch that produced the race condition this chapter opened with. `#pragma pack(1)` matters for the same reason: the compiler's default struct padding is a convenience for software-only structs, and trusting it here is precisely the informal convention this section argues against. Explicit layout, or it isn't a contract.

Second, the timing contract: a stated maximum latency from event to data-ready, and a stated mechanism for signaling readiness, whether that's a doorbell interrupt, a polled status register, or a ring buffer with a monotonically incrementing sequence number the consumer checks. For this build, that mechanism is the sequence number embedded in the struct above, checked by the consumer rather than assumed from an interrupt alone:

cpp

```
// Software side: lock-free single-producer/single-consumer poll
class MarketEventRing {
public:
    explicit MarketEventRing(volatile ParsedMarketEvent* dma_base,
        size_t depth)
        : ring_(dma_base), depth_(depth) {}

    // Returns false if nothing new has landed since last_seen_seq
    bool try_consume(uint64_t& last_seen_seq, ParsedMarketEvent&
out) {
        const size_t idx = last_seen_seq % depth_;
        const ParsedMarketEvent snapshot = ring_[idx]; // volatile
read, DMA-coherent memory

        if (snapshot.seq_no <= last_seen_seq) {
            return false; // FPGA hasn't written a new entry here
yet
        }
        if (snapshot.seq_no > last_seen_seq + 1) {
            // gap: consumer fell behind, ring wrapped under it –
the failure-contract
            // case this section requires deciding on before
implementation, not after
            handle_ring_overrun(last_seen_seq, snapshot.seq_no);
        }
        out = snapshot;
        last_seen_seq = snapshot.seq_no;
        return true;
    }

private:
    volatile ParsedMarketEvent* ring_;
    size_t depth_;
    void handle_ring_overrun(uint64_t expected, uint64_t got);
};
```

volatile is doing real work here, not decoration. It tells the compiler not to cache ring_[idx] in a register across calls, because the memory underneath changes from outside the C++ abstract machine entirely. This is also the first place in the book where "the FPGA wrote it" and "the CPU can see it" are not the same moment, which is a PCIe ordering problem, not a C++ problem, and

Chapter 6 solves it properly. This version is the naive one that motivates that chapter, not the production one.

Third, the failure contract: what happens when the FPGA can't process a message in time, when a checksum fails, when the software side falls behind and the ring buffer wraps, as flagged by `handle_ring_overflow` above. Every one of these has to have a defined behavior before the first line of the DMA driver gets written, because retrofitting failure handling onto a boundary that was designed only for the happy path means touching both the hardware and software sides again, and a hardware-side change means another trip through synthesis and timing closure.

For this build, the boundary sits at parsed-and-filtered market data plus a narrow set of pre-trade risk checks on the way in, and validated order intent on the way out. The FPGA receives raw packets off the wire, parses fixed-format messages, applies instrument filtering and hard limit checks, and hands a `ParsedMarketEvent` to the software matching engine over the ring buffer above. The matching engine owns book state, order matching logic, and any risk check that requires cross-message context. Decisions flow back to the FPGA as a similarly fixed-width struct, and the FPGA handles final packet construction and transmission. Chapter 5 specifies this contract at the field level. Chapters 6 and 7 build the DMA path and the clock domain crossings that make it real without introducing the exact race condition this section opened with.

Latency budget ledger — chapter 1.3

Interface boundary fixed: fabric owns wire-to-parsed-message plus hard-coded pre-trade limits, software owns book state and matching logic, fabric owns decision-to-wire. Contract requirement locked in for the rest of the book: every crossing needs a byte-exact wire format, a stated maximum latency with an explicit readiness signal, and a defined failure behavior, specified before implementation begins on either side.

Chapter 2

HDL Fundamentals for C++ Engineers

2.1 Verilog and SystemVerilog concepts mapped to familiar C++ analogies

An engineer with fifteen years of C++ experience writes his first Verilog always block, structures it like a function with sequential statements he expects to execute top to bottom, synthesizes it, and gets a design that behaves nothing like what he wrote. Every statement in that block didn't run in sequence. It described hardware that all exists and all operates at once, and the simulator was telling him this the whole time in ways his C++ instincts weren't equipped to read. This is the single most common point of failure for software engineers learning HDL, and it's worth being direct about before going further: the syntax looks like a programming language, and that resemblance actively works against you.

Start with what actually transfers cleanly. A module is close enough to a class that the analogy holds: it has a defined interface, in the form of ports, and internal state, in the form of registers, and you instantiate it multiple times the way you'd instantiate a class multiple times. Parameters in Verilog behave like template parameters in C++, resolved at compile time:

verilog

```
module fifo #(
    parameter int Width = 32,
    parameter int Depth = 16
)()
    input  logic clk,
    input  logic rst_n,
    input  logic wr_en,
    input  logic [Width-1:0] wr_data,
    input  logic rd_en,
    output logic [Width-1:0] rd_data,
    output logic full,
```

```

        output logic empty
    );
    // implementation covered in full in chapter 7
endmodule

// Two instantiations, same logic, different widths, resolved at
// synthesis
fifo #(.Width(64), .Depth(256)) market_data_fifo (.clk(clk),
.rst_n(rst_n) /*...*/);
fifo #(.Width(32), .Depth(64)) order_ack_fifo (.clk(clk),
.rst_n(rst_n) /*...*/);

```

Same relationship to template<int Width, int Depth> class Fifo as you'd expect, except this instantiation happens once, at synthesis, and produces physically separate gates for each instantiation rather than sharing code the way a C++ binary shares a function body across call sites. If you've written template metaprogramming, generate blocks for building repeated hardware structures off a parameter will feel familiar for the same reason: both are compile-time code generation, not runtime behavior.

Where the analogy breaks is execution model, and this is the part that has to be rewired, not extended. C++ code executes one statement after another on one thread of control, and even multithreaded C++ is sequential within each thread. Hardware has no threads and no program counter. Every always block you write describes a piece of circuitry that exists permanently and reacts continuously to its inputs, and every always block in your module is active simultaneously, all the time, not triggered in some order you control. The order you write them in the file has no bearing on execution order, because there is no execution order. There is only what each circuit does in response to its inputs on each clock edge.

verilog

```

module msg_filter #(
    parameter int InstrumentWidth = 16
)()
    input  logic clk,
    input  logic rst_n,
    input  logic valid_in,
    input  logic [InstrumentWidth-1:0] instrument_id,
    input  logic [InstrumentWidth-1:0] allowed_id,
    output logic pass,          // combinational: pure function of
current inputs
    output logic [31:0] accept_count_q // sequential: persists
across cycles
);
    // Combinational: no clock, no memory, re-evaluates
continuously
    always_comb begin
        pass = valid_in && (instrument_id == allowed_id);
    end

```

```

// Sequential: state that only changes on a clock edge
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        accept_count_q <= '0;
    else if (pass)
        accept_count_q <= accept_count_q + 1;
end
endmodule

```

Combinational logic, written here with `always_comb`, describes a network of gates whose output is a pure function of its current inputs, closer to a stateless expression than to a function call, because there's no call happening. Sequential logic, written with `always_ff`, describes state that updates on a clock edge, your closest equivalent to a variable that persists across loop iterations, except every register in your design updates on the same edge at the same time, not one after another.

`always_comb` over the older `always @(*)` is not stylistic. If you forget to assign `pass` on some code path, `always @(*)` silently synthesizes a latch that holds its last value, a latch that will pass timing analysis and then fail in the field on the one input combination your testbench didn't hit. `always_comb` throws a compile warning the moment a signal isn't assigned on every path, which is the difference between catching this mistake in this chapter and catching it in Chapter 8's timing closure report.

Blocking versus non-blocking assignment is where this stops being an abstract distinction and starts being the thing that produces working or broken hardware. Blocking assignment, `=`, updates its target immediately within the simulation step, the way a normal C++ assignment does, and you use it inside combinational blocks, as above. Non-blocking assignment, `<=`, schedules the update to happen at the end of the current time step, after every non-blocking assignment in every `always` block has read its inputs, and you use it inside sequential blocks. Mix them up and you get a design that simulates correctly but synthesizes into something with a different meaning, or worse, simulates fine and fails on real silicon because the simulator was more forgiving about ordering than the gates are.

verilog

```

// Broken: blocking assignment inside a clocked block collapses two
// stages into one
module bad_shift_reg (
    input  logic clk,
    input  logic d_in,
    output logic d_out
);
    logic stage1;
    always_ff @(posedge clk) begin
        stage1 = d_in;    // executes immediately
    end
endmodule

```

```

        d_out = stage1; // reads the value just written above,
same time step
    end
    // Behaves as one register of delay, not two – the pipeline
stage you intended
    // never exists, in simulation or in silicon
endmodule

// Correct: non-blocking assignment defers the update, preserving
two real stages
module good_shift_reg (
    input  logic clk,
    input  logic d_in,
    output logic d_out
);
    logic stage1;
    always_ff @(posedge clk) begin
        stage1 <= d_in;    // scheduled, not applied yet
        d_out  <= stage1;  // reads stage1's value from before this
edge
    end
    // Two genuine stages of delay – a real pipeline register, not
a pass-through
endmodule

```

The rule that matters in practice: non-blocking for anything inside a posedge clk block, blocking for anything inside a combinational block, no exceptions worth remembering, because the exceptions are where the bugs live.

SystemVerilog over plain Verilog earns its place here mainly through two additions worth knowing before Chapter 3's toolchain discussion. `always_comb` and `always_ff`, used throughout the examples above, replace the generic `always @(*)` and `always @(posedge clk)` forms and catch the accidental-latch mistake at compile time. interface blocks let you bundle related signals the way a struct bundles related fields, cutting down on the port-list sprawl a plain Verilog module accumulates as a design grows. Neither changes the execution model. They make it harder to accidentally violate it, which for an engineer new to hardware is worth more than the syntax sugar implies.

Latency budget ledger — chapter 2.1

No latency figures at this stage; this chapter establishes the execution model required to write correct HDL in the first place. The cost of getting blocking versus non-blocking assignment wrong isn't measured in nanoseconds, it's measured in a design that passes simulation and fails timing closure, or produces silently incorrect results on hardware, which is a more expensive class of bug to chase than anything in the software path from Chapter 1.

2.2 Combinational versus sequential logic, and why order matters differently in hardware

A design passes every simulation test, gets synthesized, and then fails intermittently on the bench in a way that never reproduces in the simulator. The engineer spends two days assuming it's a power supply issue or a bad board before finding the actual cause: a long chain of combinational logic computing a price comparison across several pipeline stages, all crammed into a single clock cycle, with a propagation delay that exceeds the clock period under worst-case conditions. The simulator, running at the register-transfer level with idealized zero-delay logic, had no way to catch this. The hardware, bound by the actual physical time it takes a signal to propagate through a chain of gates, caught it immediately, just not in a way anyone could see until the board was under real timing pressure.

The distinction that matters is not just what combinational and sequential logic compute, but what "order" means for each, and this is where the C++ instinct for sequencing has to be replaced with something closer to a physical model. Combinational logic has no concept of before and after within a clock cycle. An expression like `pass = valid_in && (instrument_id == allowed_id)` isn't evaluated once and stored; it's a physical network of gates continuously settling toward its output value every time an input changes, and the "order" you write sub-expressions in has zero effect on when the circuit finishes settling. What does matter is depth: how many gates a signal has to propagate through from input to output, because every gate adds a small, physically real delay, typically in the range of tens to a few hundred picoseconds depending on the gate and the process node, and those delays add up along the longest path through your logic.

verilog

```
// A price comparison built as one long combinational chain – looks
fine functionally,
// hides a timing problem that only shows up under synthesis, not
simulation
module price_check_naive (
    input  logic [31:0] order_price,
    input  logic [31:0] book_bid,
    input  logic [31:0] book_ask,
    input  logic [31:0] notional_limit,
    input  logic [31:0] order_qty,
    output logic        within_spread,
    output logic        within_notional,
    output logic        approved
);
    always_comb begin
```

```

        within_spread    = (order_price >= book_bid) && (order_price
<= book_ask);
        within_notional = (order_price * order_qty) <=
notional_limit; // multiply in the chain
        approved        = within_spread && within_notional;
    end
    // All three signals settle within one clock period, or they
don't. There is no
    // partial credit. The multiply alone can be the deepest path
in the whole design.
endmodule

```

That multiplier is the problem, and it's worth being blunt about why: a 32-bit multiply implemented as combinational logic is a genuinely deep gate structure, and on a mid-range FPGA it can easily eat 4 to 8 nanoseconds of propagation delay on its own. At a 250 MHz clock, your entire period is 4 nanoseconds. Put that multiply in a combinational path alongside a comparison and a boolean AND, and you've built a circuit that cannot physically finish computing before the next clock edge arrives, and synthesis timing analysis will tell you exactly this in Chapter 8. Simulation will not, because RTL simulation without back-annotated timing treats every gate as instantaneous.

Sequential logic is where you buy yourself out of this. Breaking the same computation into registered stages doesn't make the multiply faster, it gives each piece of combinational logic its own clock period to settle in, at the cost of latency measured in clock cycles instead of nanoseconds of pure propagation delay.

verilog

```

// Same check, pipelined – each always_ff stage gets a full clock
period,
// combinational depth per stage is now bounded and small
module price_check_pipelined (
    input  logic      clk,
    input  logic      rst_n,
    input  logic [31:0] order_price,
    input  logic [31:0] book_bid,
    input  logic [31:0] book_ask,
    input  logic [31:0] notional_limit,
    input  logic [31:0] order_qty,
    output logic      approved_q
);
    logic      within_spread_q;
    logic [63:0] notional_q;
    logic      within_notional_q;

    // Stage 1: spread check and multiply, each shallow enough for
one period
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin

```

```

        within_spread_q <= 1'b0;
        notional_q      <= '0;
    end else begin
        within_spread_q <= (order_price >= book_bid) &&
(order_price <= book_ask);
        notional_q      <= order_price * order_qty; // multiply
gets its own stage
    end
end

// Stage 2: compare against limit using the registered product
from stage 1
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        within_notional_q <= 1'b0;
    else
        within_notional_q <= (notional_q <= {32'b0,
notional_limit});
    end

// Stage 3: final AND, trivial depth
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        approved_q <= 1'b0;
    else
        approved_q <= within_spread_q && within_notional_q;
    end
endmodule

```

Notice what changed and what didn't. The logical result is identical. What you paid for correctness is three clock cycles of latency instead of one, roughly 12 nanoseconds at 250 MHz instead of a single period that the naive version couldn't actually meet. This is the fundamental trade this chapter exists to make explicit: in software, adding more sequential steps to a computation almost always costs you time, full stop. In hardware, adding more sequential stages is frequently how you buy back timing margin you'd otherwise lose to combinational depth, at the cost of a small, fixed, and entirely predictable latency addition. A software engineer's instinct is to minimize steps. A hardware engineer's instinct is to minimize the deepest combinational path between any two registers, and those are not the same optimization target, and reaching for the C++ instinct here is exactly what produces the naive version above.

The other consequence worth stating plainly: because every register updates on the same clock edge, the "order" of your always blocks in the source file is not just irrelevant to functionality, it's irrelevant to timing too. What determines timing is purely the combinational depth between one register and the next register that consumes its output, regardless of which block came first in the file or which block you wrote first while designing. This is why hardware engineers draw pipeline diagrams before writing code, tracing register-to-register paths

explicitly, rather than reasoning about a design the way you'd trace a call stack. Chapter 8 gives you the tool that measures this precisely, but the discipline that avoids needing to debug it on a bench starts here, at the point where you decide how many registers sit between your inputs and your outputs.

Latency budget ledger — chapter 2.2

Design principle locked in, not yet a latency number: combinational depth between registers is the constraint, not instruction count. A 32-bit multiply alone can consume 4 to 8 ns of propagation delay, enough to violate a 250 MHz (4 ns) clock period on its own if left unregistered. Pipelining the price-check logic into three stages adds roughly 12 ns of fixed latency in exchange for meeting timing reliably, a trade this book will make repeatedly in the ingestion and matching-boundary logic in Part II.

2.3 Simulating a first module before touching real silicon

A team skips a proper testbench for their first parser module, reasoning that it's simple enough to just synthesize and check on the board with a logic analyzer. Three days later they're staring at waveforms trying to figure out whether the bug is in their parsing logic, their board's clock configuration, a signal integrity issue on the PCB, or a misconfigured pin constraint, because all four look identical from the outside: wrong data coming out. On a real board, every one of those failure modes produces the same symptom and gives you almost no visibility into which layer is responsible. In simulation, those four failure modes are impossible to conflate, because three of them don't exist yet. This is the actual argument for simulating before synthesizing anything, and it has nothing to do with convenience.

A testbench is not a smaller, simpler version of your design. It's a separate piece of Verilog whose only job is to drive inputs into your module and check outputs against known-correct values, and it never gets synthesized onto hardware, which means it can use constructs and behavior that would be meaningless or illegal in real gates: arbitrary delays, file I/O, print statements, and unclocked procedural logic that exists purely for verification. Take the `msg_filter` module from 2.1 as the module under test.

verilog

```
// Testbench for msg_filter – drives inputs, checks outputs, never
synthesized
`timescale 1ns/1ps

module msg_filter_tb;
    logic clk = 0;
    logic rst_n;
```

```

    logic valid_in;
    logic [15:0] instrument_id;
    logic [15:0] allowed_id;
    logic pass;
    logic [31:0] accept_count_q;

    // Instantiate the module under test exactly as it would be in
the real design
    msg_filter #(.InstrumentWidth(16)) dut (
        .clk(clk), .rst_n(rst_n), .valid_in(valid_in),
        .instrument_id(instrument_id), .allowed_id(allowed_id),
        .pass(pass), .accept_count_q(accept_count_q)
    );

    // Free-running 250 MHz clock, exists only in simulation
    always #2 clk = ~clk;

    task automatic check(string name, logic expected_pass, logic
[31:0] expected_count);
        if (pass !== expected_pass || accept_count_q !==
expected_count) begin
            $error("FAIL %s: pass=%b (exp %b) count=%0d (exp %0d)",
                name, pass, expected_pass, accept_count_q,
expected_count);
        end else begin
            $display("PASS %s", name);
        end
    endtask

    initial begin
        rst_n = 0; valid_in = 0; instrument_id = '0; allowed_id =
16'hABCD;
        @(posedge clk); @(posedge clk);
        rst_n = 1;

        // Case 1: matching instrument, should pass and increment
        @(negedge clk);
        valid_in = 1; instrument_id = 16'hABCD;
        @(posedge clk); #1;
        check("matching id", 1'b1, 32'd1);

        // Case 2: non-matching instrument, should not pass, count
unchanged
        @(negedge clk);
        instrument_id = 16'h0001;
        @(posedge clk); #1;
        check("non-matching id", 1'b0, 32'd1);
    end

```

```

        // Case 3: valid_in deasserted, matching id, should not
pass
        @(negedge clk);
        valid_in = 0; instrument_id = 16'hABCD;
        @(posedge clk); #1;
        check("valid deasserted", 1'b0, 32'd1);

        $display("Testbench complete");
        $finish;
    end
endmodule

```

Three things in this testbench are worth pointing at directly because they're where engineers coming from C++ unit testing get the habits wrong. The clock generator, always `#2 clk = ~clk`, has no equivalent in application code; it exists purely to drive the sequential logic under test and would be meaningless in synthesizable RTL, which is exactly why it's fine to write it this loosely here. The `#1` delay after each `@(posedge clk)` before checking outputs is not arbitrary padding, it's giving the non-blocking assignments from 2.1 time to actually complete their scheduled update before you sample them, and forgetting this delay is a common way to read the previous cycle's value and conclude a working design is broken. And the checks themselves use `!=` rather than `!=`, because Verilog's four-state logic includes `x` (unknown) and `z` (high-impedance) as real values, and a comparison that silently treats `x` as unequal to everything, but not equal to `x`, catches uninitialized-signal bugs that a two-state language like C++ has no equivalent concept for. An `x` propagating through your design in simulation almost always means a register was read before it was ever written, and that bug is nearly invisible on real hardware, where an uninitialized flip-flop just powers up to some physical value that looks deceptively like real data. Run this and you get a pass or fail on functional correctness, in seconds, with zero risk to a physical board and zero ambiguity about which layer failed. What it does not tell you is whether the design will meet timing at your target clock frequency, because RTL simulation of this kind treats every gate as having zero propagation delay, which is precisely the gap that let the price-check module in 2.2 pass simulation while being unroutable at 250 MHz. Functional simulation and timing verification are answering two different questions, and conflating them is the second most common mistake after skipping simulation entirely: a design that simulates cleanly has proven its logic is correct assuming infinitely fast gates, and has proven nothing yet about whether real gates, with real delay, can keep up with your clock. Chapter 3 sets up the toolchain that runs this testbench as part of a repeatable build, and Chapter 8 is where the timing question this chapter deliberately leaves open gets answered with real numbers instead of assumptions.

Latency budget ledger — chapter 2.3

No latency figures yet; this chapter establishes functional correctness as a

precondition, not a substitute, for the timing work in Chapter 8. Every module built in Part II gets a testbench of this shape before synthesis, on the principle that a bug caught in simulation costs minutes, the same bug caught on the bench costs days, and the same bug caught in production costs the incident review this book's war stories are built from.