MBSTech FPGA Design Flow

The following section outlines the FPGA design flow implemented by the MBSTech design team (me, myself and I) during the development of FPGA code.

Contents

Step 1: Define Requirements

This is the foundation of any FPGA project. Begin by clearly identifying the functional and non-functional requirements of the design. What should the FPGA do? What data rates and processing throughput are needed? What interfaces does it support (SPI, I2C, Ethernet, etc.)? Define timing constraints, clock frequencies, power limitations, and target device family. Understanding whether the design is for control logic, signal processing, or communication is critical. Also document environmental constraints such as temperature range or radiation tolerance (if applicable). These requirements guide architectural decisions, test planning, and tool selection. Clear, measurable requirements help determine whether the design is successful and reduce the likelihood of costly rework later in the project lifecycle.

Best Practices for Defining FPGA Requirements

✅ 1. Make Requirements Clear, Concise, and Testable
Each requirement must be unambiguous (no vague terms like "fast" or "as needed").
They must be testable or verifiable by simulation, inspection, or hardware test.
Use measurable criteria, e.g., “must support SPI at 10 MHz” or “latency must not exceed 100 ns”.

✅ 2. Categorise Requirements
Group them into:
Functional Requirements: What the FPGA must do (e.g., data encoding, interfacing, timing).
Non-Functional Requirements: Constraints like power, area, latency, clock domains, or safety levels.
Interface Requirements: Protocols, voltages, pinouts, handshaking logic, etc.
Performance Requirements: Throughput, latency, error rates, timing margins.
Verification Requirements: How each requirement will be validated (testbench, hardware test, etc.).

✅ 3. Use Unique Identifiers
Give every requirement a unique ID (e.g., REQ-001) so traceability is possible throughout design and test.

✅ 4. Define Assumptions and Constraints
Record assumptions like clock sources, reset behaviour, or signal integrity limits.

List external constraints (e.g., package size, toolchain version, radiation tolerance).

✅ 5. Use a Requirements Table or Matrix

Helps link each requirement to a test method and design module.

Also supports requirement traceability, useful for regulated industries (avionics, medical, automotive).

✅ 6. Be Iterative but Controlled
Start broad and refine. Use version control to track changes.

Involve stakeholders (e.g., systems engineers, software team) early.

Step 2: Create Block Diagram / Architecture

Once the requirements are clear, design a high-level architecture using a block diagram. This diagram shows major functional blocks (e.g., filters, control FSMs, interfaces), their interconnections, and data/control flows. Include how clocks and resets are managed and define boundaries between modules for code modularity. Consider how the design will be partitioned for simulation, synthesis, and testing. This is also where you decide on reuse of IP cores, how inputs/outputs are interfaced with hardware, and where key timing constraints will lie. A good block diagram keeps your team aligned and makes the overall system easier to verify, debug, and maintain.

Best Practices for Creating Block Diagrams & FPGA Architecture

✅ 1. Start from Requirements
Trace each functional requirement to a logical block.

Group related requirements together to form high-level functional units.

Ensure every function has a home in the block diagram.

✅ 2. Use Hierarchical Design
Break down complex designs into smaller modules (e.g., uart_rx, packet_parser, fsm_ctrl).

Represent major blocks at the top level, with internal sub-blocks encapsulated.

This approach supports divide-and-conquer coding, simulation, and reuse.

✅ 3. Use Clear Naming and Signal Direction
Label each block clearly with its function.

Show inputs on the left, outputs on the right, and clocks/resets at the top or bottom.

Name signals consistently across modules for clarity and integration ease.

✅ 4. Annotate with Clock Domains
If using multiple clocks, indicate which blocks belong to which clock domain.

Clearly show domain crossings and buffers (e.g., FIFOs, synchronisers).

✅ 5. Include Interfaces and Buses
Define standard interfaces such as AXI, SPI, or Avalon at the top-level block.

Show control/data separation and bit widths (e.g., data [31:0], addr [15:0]).

Use interface groupings or color-coding to show related signals.

✅ 6. Plan for Debug and Testability
Include blocks or hooks for internal debug (e.g., ILA cores, debug muxes).

Consider scan chains or observation points early in the architecture.

✅ 7. Document Each Block
Write a brief text description for each block: what it does, inputs, outputs, FSM states (if applicable).

This helps hand-off to other engineers and aligns with formal design reviews.

✅ 8. Use Tool-Aided Diagrams When Possible
Use tools like Microsoft Visio, draw.io, Lucid chart, or even Vivado's IP Integrator.

Keep diagrams under version control (as images or source files if possible).

Step 3: Design Reuse

Design reuse is a core principle within the MBSTech FPGA workflow, enabling faster development, improved consistency, and reduced risk across multiple projects. The reuse of validated modules, IP cores, interface blocks, and scripts not only accelerates the design process but also contributes to improved quality and maintainability of the final system. Equally important is designing for reuse from the outset. This means writing code and building modules with a long-term view, anticipating future adaptation and integration in other projects. To support this, the following best practices are observed:

Best Practices for Reuse

✅ Principles for Designing with Reuse in Mind:
Modular Architecture: Break down functionality into self-contained, independent modules with clear responsibilities and minimal external dependencies.

✅ Parameterized Design: Use generics (VHDL) or parameters (Verilog/System Verilog) to make modules configurable without rewriting code.

✅ Standard Interfaces: Where possible, adopt standard bus protocols (e.g., AXI, Avalon, Wishbone) to ease integration with other IP.

✅ Comprehensive Documentation: Include interface descriptions, signal timing expectations, usage examples, and configuration guidance.

✅ Synthesis-Friendly Code: Avoid constructs that may hinder synthesis or cross-platform compatibility. Aim for portability between toolchains and FPGAs.

✅ Practices for Reusing Existing IP:
Internal IP Libraries: Maintain a structured library of reusable, version-controlled IP with clearly tagged releases and changelogs.

✅ Testbench Preservation: Always retain and update testbenches for reusable modules to facilitate quick regression testing.

✅ Packaging and Abstraction: Package IP using design tools (e.g., Intel Platform Designer, Xilinx IP Integrator) to ease drag-and-drop reuse in future projects, look for these tools in other FPGA designer tools.

✅ Design Reviews: Reusable components undergo stricter code reviews, ensuring robustness, clear interfacing, and scalability.

✅ Designing for reuse not only benefits the current project but builds a strong foundation for long-term capability development. This strategic investment in reusable assets supports MBSTech's goal of rapid iteration without sacrificing reliability or maintainability.

Step 4: Write RTL Code

Write the Register Transfer Level (RTL) code using an HDL such as Verilog, VHDL, or System Verilog. Each module should be a clean implementation of a block from the architecture. Focus on synthesizable coding practices, avoid latches unless needed, and document assumptions. Use parameters or generics to allow for reuse and scalability. Code should be readable, well-commented, and structured for easy simulation and debugging. Modular design with clearly defined interfaces makes testing and integration smoother. Use version control from the start.

Ensure your code follows any project-specific or industry-standard naming and formatting guidelines.

Best Practices for Writing RTL Code (VHDL / Verilog / System Verilog)

✅ 1. Write for Synthesis, Not Just Simulation
Avoid constructs not supported by synthesis tools (e.g., #delays in Verilog).

Use only synthesizable features; always check synthesis warnings.

✅ 2. Use One Process Block per Clock Domain (VHDL) or Always (System Verilog)
Keep combinational and sequential logic clearly separated.

Don't mix latches and flip-flops in the same process.

Always use asynchronous reset only when needed and know your FPGA vendor's best practices for resets.

✅ 3. Infer Flip-Flops and RAMs Properly
Use simple, idiomatic structures for flip-flop inference.

Use templates provided by your FPGA vendor to infer block RAM, dual-port RAM, or FIFO reliably.

✅ 4. Avoid Latches
Unintended latches usually result from incomplete if or case statements.

Always provide a default assignment for outputs in combinational blocks.

✅ 5. Parameterise
Use generics (VHDL) or parameters (Verilog/System Verilog) to make modules reusable and scalable.

Avoid excessive parameterisation that complicates verification.

✅ 6. Use Consistent Naming Conventions
Prefix clocks as clk_, resets as rst_, active-low signals as _n, buses as [N:0].

Clearly differentiate between inputs, outputs, and internal signals.

✅ 7. Encapsulate Logic into Modules
Keep modules small and focused — one clear purpose per module.

Group related functionality into libraries or packages.

✅ 8. Comment Thoughtfully
Explain the why, not just the what.

Document FSM states, parameter meanings, and corner cases.

✅ 9. Simulate Early and Often
Don’t wait to write the testbench — simulate each block as soon as it's written.

Validate waveform behaviour, reset conditions, and edge triggering.

✅ 10. Avoid Clock Gating
Instead, use enable signals and let synthesis tools optimise power and logic safely.

Direct clock gating often leads to unreliable or unsafe timing.

❌ Common Pitfalls to Avoid
Latch inference from incomplete assignments.

Overusing blocking assignments in sequential logic (Verilog).

Unconstrained signals leading to unintended synthesis results.

Clock domain crossings without proper synchronisation.

Wide buses or deeply nested logic that cause timing failures.

Step 5: Write Testbench

A testbench simulates the environment in which your RTL will operate. It drives stimulus to the design-under-test (DUT) and checks outputs, either manually or via assertions. Write separate testbenches for unit-level and system-level testing. Include test vectors for normal operation, edge cases, and fault conditions. Use clocks and resets to mirror hardware behaviour. For complex systems, consider using verification frameworks (e.g., UVM for System Verilog) or co-simulation with Python or C. Effective testbenches are essential for debugging and are often reused for regression testing later in the design cycle.

Best Practices for Writing Testbenches

✅ 1. Think Like a Tester — Not a Designer
The testbench must validate the DUT, not just exercise it.

Write it from the perspective of a user or external environment: focus on what should happen, not how it's implemented.

✅ 2. Build a Realistic Stimulus Environment
Emulate actual signal conditions: bit timings, protocol sequences, and backpressure (e.g., flow control).

Add clock generators, reset sequencing, and realistic input vectors — especially for asynchronous designs.

Use constrained-random stimulus or step through edge cases.

✅ 3. Create Self-Checking Testbenches
Testbenches should automatically flag errors using assert, if, or monitor statements.

Write checkers that validate bit patterns, state transitions, and handshake protocols.

✅ 4. Use Golden Models for Output Comparison
Write a reference model in the testbench (often behavioural code) to mirror expected DUT behaviour.

At each cycle or event, compare the DUT's output to the model.

Print a diff or summary when mismatches occur.

✅ 5. Use Verbose, Informative Simulation Output
Print cycle-by-cycle outputs, error locations, and internal state changes when debugging.

Make output readable and filterable — not just raw waveform dumps.

Add simulation banners to indicate test phase and sub-test progress.

✅ 6. Modularise and Reuse Testbench Code
Use reusable procedures, clock generators, and stimulus tasks.

Split into test controller, stimulus, checker, and monitor blocks for clarity.

✅ 7. Simulate Edge Conditions and Faults
Force invalid inputs, glitches, resets during operation, and timing violations (e.g., pulse-width errors).

Confirm the DUT reacts safely and predictably.

✅ 8. Use a Testcase Framework (Optional)
For large designs, create a testbench infrastructure that runs multiple test cases with a pass/fail result.

Use text input vectors, waveform logging, or even Python-driven co-simulation for higher-level tests.

❌ Common Pitfalls
Relying only on waveform inspection — always write checkers.

Creating trivial stimulus that only covers "happy paths."

Ignoring clock-domain behaviour or glitches on async inputs.

Not synchronising signals properly with respect to clocks in the testbench itself.

Step 6: Simulate Functional Behaviour (Pre-Synthesis)

Run functional simulations using your HDL simulator (e.g., ModelSim, XSIM, Riviera-Pro) to validate logical correctness. This is done before synthesis and reflects the ideal behaviour of your RTL code. Simulations verify logic functionality, control sequencing, and finite-state machines. You observe waveforms, assert outputs, and check timing using simulated clocks. Test cases should cover nominal and edge conditions. This step helps catch design logic bugs before synthesis, which can save time and cost. Use coverage reports to measure how thoroughly the design has been tested and iterate until you reach acceptable coverage.

Best Practices for Simulating Functional Behaviour (Pre-Synthesis)

✅ 1. Run Simulations Early and Often
Simulate as soon as a block is written — don't wait for full integration.

Focus on unit testing individual modules before system-level integration.

✅ 2. Use the Testbench as the Simulation Driver
The testbench should provide clocks, resets, and stimulus, and it should validate outputs in real-time.

Keep it modular and re-runnable.

✅ 3. Simulate All Edge Cases
Include nominal behaviour and edge cases: boundary conditions, illegal inputs, timeouts, and unexpected resets.

Check FIFO underflow/overflow, FSM idle states, or rare race conditions.

✅ 4. Observe Waveforms, But Also Use Assertions
Use a waveform viewer (e.g., GTKWave, Vivado, ModelSim) to visually debug logic.

Supplement with assertions, logs, or error counters for functional correctness.

✅ 5. Validate Bit-Accuracy and Timing Alignment

For data-processing pipelines, ensure bit-accurate output.

Check for off-by-one errors in latency-sensitive designs.

Use signal tap points or history buffers in the testbench for easy debugging.

✅ 6. Verify FSMs and Control Logic Transitions
Manually walk through state machines in waveforms to ensure correct sequencing.

Simulate transitions triggered by data flow or control signals — not just clock edges.

✅ 7. Check Protocol Compliance
For interfaces like SPI, UART, AXI, etc., verify timing relationships, handshakes, and data alignment.

Use protocol monitors if available or write your own in the testbench.

✅ 8. Establish a "Pass" Criteria
Use flags, output checks, or final summary reports to clearly show test pass/fail.

Don't consider a simulation successful just because it "runs."

❌ Common Pitfalls
Only testing one stimulus pattern (e.g., "golden path").

Ignoring reset edge cases or uninitialized conditions.

Overlooking unconnected or floating internal signals.

Assuming simulation speed implies correct function — always verify outputs.

Step 7: Debug and Fix RTL/Testbench Issues

If simulations reveal incorrect behaviour, isolate and fix the RTL or testbench. Use waveform viewers to trace signals and compare expected vs actual behaviour. Common issues include incorrect state transitions, signal initialization, or timing alignment. Modify your code in small increments and re-simulate to confirm fixes. Keep your testbench up to date with changes. Use assertions or logging to identify subtle bugs early. This iterative debugging process is where robust simulation tools and good design documentation are invaluable. Only move on once the pre-synthesis simulation passes all intended test cases.

Best Practices for Debugging and Fixing RTL/Testbench Issues (Section 6)

✅ 1. Use a Structured Debug Approach
Don't start changing code blindly — identify the symptom, then trace back to root cause.

Use the waveform viewer (ModelSim, Vivado, etc.) to track signal transitions and compare expected vs actual logic states.

✅ 2. Use Assertions and Logging
Place assertions inside your RTL to check for forbidden states or protocol violations.

Use report, $display, or assert property constructs to log events and flag errors during simulation.

✅ 3. Validate FSM Behaviour
Debug state transitions carefully — FSM errors are a top source of bugs.

Use named state values and waveform color-coding (where possible) to spot illegal or unexpected transitions.

✅ 4. Trace Signal Propagation
Identify the signal of interest and trace it backward from the output or forward from the stimulus.

Watch for missed resets, wrong initialisations, or misrouted signals in hierarchical designs.

✅ 5. Fix One Issue at a Time
Make one change per simulation cycle and re-run to confirm the effect.

Use version control (e.g., Git) to avoid regressions or logic breakage during debugging.

✅ 6. Focus on Testbench First, RTL Second
Misbehaving outputs may be due to bad stimulus, not bad RTL.

Confirm that clock edges, signal timings, and stimulus logic match intended scenarios.

✅ 7. Use Waveform Markers and Bookmarks
Tag key moments like state transitions, signal toggles, or assertion failures.

Zoom into critical regions to examine clock alignment or bit mismatches.

✅ 8. Log Time-Based Behaviours
Use logging to compare expected vs actual events at simulation time t.

For example: if (output!= expected) $display("Mismatch at time %t", $time);.

✅ 9. Cross-Check with Requirements
Each failure should be traceable to a violated requirement or missed edge case.

Helps avoid “random patching” and keeps design aligned to goals.

❌ Common Pitfalls
Changing multiple parts of RTL without isolating root cause.

Fixing symptoms (e.g. adding delays) instead of identifying design flaws.

Ignoring reset issues or metastability in asynchronous paths.

Assuming passing simulation = bug-free design — always verify with multiple test cases.

Step 8: Synthesize Design

Synthesis translates your RTL into a gate-level netlist using your FPGA vendor’s toolchain (e.g., Xilinx Vivado, Intel Quartus, Lattice Radiant). It maps your design onto the primitives of the target FPGA, such as LUTs, flip-flops, and BRAMs. During synthesis, constraints like clock definitions and timing budgets are applied. Review synthesis reports to check resource utilization, fan-out warnings, and timing estimates. If your design exceeds area or fails basic timing, you may need to optimize code or reconsider architectural choices. A successful synthesis step produces a netlist ready for implementation.

✅ **9. Use Graphical RTL and Technology Views**
- Most synthesis tools (e.g., **Synplify**, **Vivado**, **Quartus**, **Radiant**) provide **graphical representations** of your design:
  - **RTL Viewer**: Shows your design as it is logically inferred from the HDL.
  - **Technology Viewer**: Shows how the logic is implemented in terms of actual gates, LUTs, flip-flops, and primitives.
- Use these views to verify:
  - Correct FSM encoding
  - Correct RAM/DSP inference
  - Signal propagation and muxing
  - Logic duplication or unexpected optimisations
- This step acts as a visual "sanity check" that what you *wrote* is being correctly *understood* and translated by the toolchain.

ChatGPT said:

Got it — here's a fully rewritten and expanded version of Section 7: Synthesize Design, keeping in line with the style and depth of the previous sections, including best practices, design tips, and use of graphical synthesis tools like Synplify, Vivado, etc.

Best Practices for Synthesizing RTL Designs

✅ 1. Understand the Purpose of Synthesis
Synthesis transforms your high-level RTL (VHDL/Verilog) into a gate-level netlist using the FPGA vendor's primitives — LUTs, flip-flops, BRAMs, DSPs, etc. It also optimises logic and prepares the design for place-and-route. This is not just a black box step — it's where subtle bugs and misinterpretations can creep in, especially when HDL is ambiguous or improperly written.

✅ 2. Clean Synthesis Output is a Goal
Aim to get as close to a warning-free synthesis as possible. Treat warnings like symptoms — most are indicators of bad or unintended design. Never assume a warning is harmless until you understand its cause and consequence.

✅ 3. Always Review Synthesis Reports Thoroughly
Check:

Resource utilisation (LUTs, FFs, RAMs, DSPs): Are usage figures in line with your expectations?

Inferred components: Are your intended RAMs, multipliers, and logic blocks inferred properly?

Critical path estimates and maximum achievable clock frequency

Fanout issues, hierarchy flattening, and unexpected optimisations

Use these to spot mistakes like unintended latches, redundant logic, or un-synthesizable constructs early.

✅ 4. Document Justified Warnings
Some warnings (e.g., inferred latches in test logic) may be intentional and unavoidable. In these cases:

Add a comment in your code

Create a synthesis warning audit in your documentation

Explain why the warning is acceptable and how it was reviewed

This is particularly important in regulated or safety-critical design flows.

✅ 5. Use Graphical RTL and Technology Views

Most synthesis tools — Synplify, Vivado, Quartus, Radiant — offer two extremely helpful graphical tools:

RTL Viewer: Shows how your HDL is interpreted logically.

Technology Viewer: Shows how it's implemented physically using LUTs, MUXes, registers, RAMs, etc.

Use these to:

Verify FSM encoding (e.g., one-hot vs binary)

Check that RAM and DSP blocks are inferred correctly

Confirm that combinational logic is not overly deep or duplicated

Trace how input signals propagate through logic layers

This acts as a visual sanity check — ensuring your intent matches tool interpretation.

✅ 6. Maintain Meaningful Hierarchy
Use synthesis settings that preserve module hierarchy, and ensure all instances have meaningful names. This helps with:

Post-synthesis simulation

Implementation analysis

Constraint mapping and debugging

✅ 7. Constrain the Design
Even during synthesis, basic constraints (e.g. create_clock, set_input_delay) help tools guide optimisation. Proper constraints prevent over-optimisation or incorrect assumptions about signal behaviour.

✅ 8. Repeat Synthesis Often
Don't wait for full design completion. Synthesize frequently to:

Check for logic bloat or incorrect inference

Catch unintended logic growth due to testbench residue or legacy code

Maintain design quality and structure

❌ Common Pitfalls
Ignoring synthesis warnings due to "it worked in simulation"

Unexpected resource overuse due to wide buses or deep logic trees

Allowing synthesis to silently infer latches or combinatorial loops

Failing to cross-check between RTL and technology views

`Assuming synthesis "optimised away" your mistake`

Step 9: Run Post-Synthesis Simulation (Optional for Sum 😊)

This step simulates the synthesized netlist to check functional equivalence and timing approximation. Although optional, it is useful for complex designs with tight timing or critical IP. Post-synthesis simulation includes propagation delays and logic mapping effects, which can reveal problems not seen in RTL-level simulations. It's slower but closer to real hardware. Compare its output to the pre-synthesis simulation results. Differences may indicate synthesis-altered behaviour or asynchronous issues. This step can increase confidence before committing to place and route, especially in safety- or mission-critical systems.

Best Practices for Post-Synthesis (Gate-Level) Simulation

✅ 1. Understand What This Simulation Really Is
Post-synthesis simulation, often referred to as gate-level simulation, simulates the netlist output from synthesis, including propagation delays and low-level optimisations. Unlike RTL simulation, this simulation reflects what will actually go onto the silicon (or fabric). It allows you to catch subtle timing-related, logic-flattening, and mapping issues before implementation.

✅ 2. Treat This as a Required Step
While many treat it as optional, robust FPGA design processes mandate gate-level simulation, especially for:

Safety-critical or certified systems (e.g., avionics, automotive)

Timing-critical designs with tight setup/hold windows

High-complexity blocks (e.g. arithmetic pipelines, multipliers, custom memories)

✅ 3. Use Identical Testbench (Where Possible)
Reuse the same testbench from RTL simulation to drive this simulation. This allows direct comparison between RTL and gate-level behaviour, ensuring synthesis did not change logic meaningfully.

✅ 4. Understand What's Different

Gate-level simulation includes:

Netlist timing delays (may be zero-delay or estimated from synthesis)

Real LUT, MUX, and flip-flop behaviour

Glitches and hazards that don't appear at RTL

Synthesis remapping or FSM encoding optimisations

✅ 5. Simulate With and Without Timing Delays
Use functional simulation (no timing) to verify logic equivalence.

Then simulate with back-annotated delays (SDF) to observe timing violations and races.

✅ 6. Check for New Problems Post-Synthesis
Look for:

Unintentional logic simplification

FSM encoding mismatches

Clock domain crossings behaving differently

Hold/setup violations emerging from gate-level delay modelling

Initialisation issues, particularly in reset-less registers

✅ 7. Use Waveform Tools with Hierarchy Mapping
Gate-level simulations tend to flatten design hierarchy. Ensure the synthesis tool preserved enough structure to trace signal flow and debug effectively in waveform viewers.

✅ 8. Confirm Bit-Accuracy and Timing
In data pipelines (e.g., FIR filters, CRC, AXI stream data), validate bit-accurate output.

Use assertions or checks to confirm timing alignment — e.g., is valid data presented in the correct clock cycle?

✅ 9. Run at Slower Clock Rates If Needed
Gate-level simulations can be very slow, especially with SDF delays. For long tests:

Reduce clock frequency or test vector count.

Focus on corner cases, timing stress tests, and functional coverage points.

❌ Common Pitfalls

Skipping post-synthesis simulation and missing a synthesis-induced bug.

Relying solely on timing reports instead of waveform verification.

Failing to back-annotate the SDF delay file (or ignoring SDF warnings).

Overlooking FSM encoding mismatches or RAM inference artefacts.

Assuming RTL simulation “proves correctness” — synthesis can and does change logic!

Step 10: Implement (Place and Route)

Implementation, or place and route, maps the synthesized netlist to actual FPGA resources. The tool determines where logic blocks, routing interconnects, memory elements, and I/O buffers are placed. It then routes the signal paths to meet timing and design constraints. At this stage, physical constraints such as pin assignments and floor planning come into play. The tool produces a detailed report of routing congestion, timing violations, and power estimates. Review these carefully. Successful implementation is a prerequisite for generating a bitstream. Optimising logic placement and adding timing constraints can help if the design fails timing at this stage.

Best Practices for Implementation (Place and Route)

✅ 1. Understand What Implementation Actually Does
Place and route takes your post-synthesis netlist and maps it physically onto the FPGA’s logic fabric. This includes:

Placing logic elements (LUTs, registers, DSPs, RAMs) into actual physical tiles.

Routing all nets and clock trees across the fabric.

Applying timing-driven optimisation to ensure your design meets setup and hold requirements.

This stage is sensitive to tool defaults, floorplan layout, and timing constraints — and is the final chance to influence performance before bitstream generation.

✅ 2. Review Timing and Congestion Reports
After each implementation run, examine:

Slack (for each clock domain and critical path)

Routing congestion and resource hotspots

Clock skew, fanout, and delay balancing

A design that meets timing with margin across process/voltage/temp (PVT) corners is considered robust.

✅ 3. Leverage Physical Constraints to Improve Layout
Most modern tools (Vivado, Quartus, Radiant, etc.) allow you to use physical constraints to guide placement. These include:

Pblocks or Logic Lock Regions: Define specific physical areas where certain modules or entities must reside.

I/O Planning: Pin placement can be fixed manually or via automated rules to group related signals near interface logic.

Floor planning Views: Use the GUI to drag, drop, and anchor modules spatially like a Lego layout.

These techniques:

Reduce routing delays

Improve timing closure

Help repeatability between builds

Allow designers to “architect” the layout rather than leave it to automation

✅ 4. Place Interface Logic Near Pins
Place serializer/deserializer logic, I/O controllers, or interface FSMs close to relevant I/O banks.

This minimises delay and improves reliability for high-speed standards like DDR, LVDS, or gigabit Ethernet.

✅ 5. Group and Isolate Critical Blocks
Keep high-speed paths, control FSMs, and clock domain crossings in logically and physically distinct areas.

This isolates critical timing from large or noisy logic regions and prevents unintended coupling.

✅ 6. Use Incremental Compilation / Implementation
When only parts of the design change, use incremental place-and-route to preserve previous layout and speed up iteration.

This supports faster compile times and consistent results across versions.

✅ 7. Align Floorplan with Clocking
Clock buffers (e.g., BUFGs, PLL outputs) have limited reach. Floorplan your design to minimise skew and ensure efficient clock tree usage.

✅ 8. Use GUI Tools to Inspect and Refine
After implementation:

Use the device view to inspect placement and congestion.

Verify that logic is packed efficiently and critical paths are not fragmented across the chip.

Adjust constraints and rerun as needed.

❌ Common Pitfalls
Allowing automated tools to place everything by default, especially in timing-sensitive designs.

Ignoring placement warnings about density or congestion.

Over-constraining regions (e.g., locking too much logic into a small Pblock).

Not re-running timing analysis after making physical changes.

Step 11: Run Timing Analysis

Timing analysis ensures your FPGA design meets setup, hold, and clocking requirements so it can operate reliably at the intended clock frequency. After place-and-route, a static timing analysis (STA) tool evaluates all possible signal paths, without needing simulation vectors. The main types of checks include:

- Setup Time Analysis: Ensures that signals arrive at flip-flops with enough time before the active clock edge. Violations here mean the circuit may not latch correct data.
- Hold Time Analysis: Ensures that signals remain stable for a required time after the clock edge. Hold violations typically result from clock skew or excessively fast paths.
- Clock Skew and Jitter Analysis: Measures variation in clock arrival times across the chip. Excessive skew can cause setup or hold violations. Jitter (timing uncertainty) must also be accounted for in timing margins.
- Multi-cycle Path Checks: For paths that intentionally span multiple clock cycles, STA must be guided to avoid false violations.
- False Path Constraints: Paths that don’t affect functionality (e.g., test logic) can be marked to exclude them from analysis.

- Timing Margins: All paths are analysed for slack — the difference between required and actual arrival times — with negative slack indicating a failure.

Overall, timing closure is a critical milestone before generating the final bitstream.

Best Practices for Static Timing Analysis (STA)

Timing analysis is what determines whether your design will operate correctly at your target clock frequency. This step doesn't rely on simulation vectors — instead, it exhaustively checks all possible paths for violations in setup, hold, and other timing constraints. A design that "simulates" correctly but fails timing is not viable in hardware.

✅ 1. Setup Time Analysis
Purpose: Ensures data arrives before the clock edge, with sufficient margin.

Best Practices:

Prioritise the worst negative slack (WNS) paths first.

Use register balancing and pipelining to meet timing.

Adjust logic to reduce long combinational chains on critical paths.

Set realistic timing constraints for each clock domain (create_clock, set_clock_groups).

✅ 2. Hold Time Analysis
Purpose: Ensures data remains stable after the clock edge.

Best Practices:

Investigate hold time violations early — they often appear even when setup timing is fine.

Use delay balancing (e.g., inserting buffer chains).

Avoid excessively fast combinational paths from flip-flop to flip-flop.

Always validate hold timing after placement and routing — this is when real delays are known.

✅ 3. Clock Skew and Jitter Analysis
Purpose: Measures variation in clock arrival times due to routing or PLL behaviour.

Best Practices:

Minimise skew by keeping synchronous logic close together.

Use dedicated clock routing resources (e.g., BUFGs, global nets).

Keep jitter budgets in mind for high-speed I/Os or SERDES blocks.

Analyse launch vs latch clock edges carefully for multi-clock paths.

✅ 4. Multi-Cycle Path Checks
Purpose: Handles timing paths that are intentionally allowed to take more than one clock cycle.

Best Practices:

Clearly declare multi-cycle paths using set_multicycle_path.

Be explicit about launch and latch clock edges (start/hold setup adjustments).

Document these paths to avoid confusion during peer review or certification.

✅ 5. False Path Constraints
Purpose: Excludes paths from timing analysis that do not affect functionality.

Best Practices:

Identify test paths, status buses, and async strobes that don't need timing closure.

Use set_false_path to prevent false errors in reports.

Regularly audit false paths to ensure they are truly non-functional.

✅ 6. Timing Margins and Slack Management
Purpose: Slack is the difference between required and actual arrival times — positive slack means success.

Best Practices:

Aim for 0.2–0.5 ns or more positive slack in typical designs for robustness.

Use timing reports and timing paths visualisation tools to analyse violations.

Don't just close timing "on paper" — understand why your design just passes or fails.

✅ 7. Clock Domain Crossing (CDC) Checks
Purpose: Ensures safe data transfer between unrelated or asynchronous clocks.

Best Practices:

Use double-synchroniser flip-flops or asynchronous FIFOs for all CDCs.

Run formal CDC analysis tools (many FPGA vendors provide them).

Never trust cross-domain paths to behave correctly without validation.

✅ 8. Review Reports Iteratively
Use Post-Place and Route timing reports — not pre-synthesis estimates — for final sign-off.

Track worst path slack, total violating paths, and top-level summary metrics.

Repeat the full static timing analysis after:

Design changes

Constraint updates

Floorplan/placement tweaks

❌ Common Pitfalls
Blindly trusting tools without validating constraints.

Overusing set_false_path or set_multicycle_path to "silence" violations.

Ignoring small hold violations (they often worsen in silicon).

Forgetting to constrain generated clocks, derived clocks, or asynchronous resets.

Failing to review hold/setup behaviour across clock domain crossings.

Step 12: Hardware Testing and Debug

Once your design passes timing analysis and implementation, the next phase moves from the digital realm into the physical world — programming and validating on actual hardware. This transition is critical: even a design that simulates perfectly and passes all static checks can behave incorrectly on an FPGA if real-world interactions, I/O timing, or hardware quirks were not fully accounted for.

✅ Bitstream Generation and FPGA Programming
Use official toolchains (e.g., Vivado, Quartus) to generate a bitstream (.bit or .sof) file.

Always version-control your bitstreams and maintain logs of synthesis and implementation settings for reproducibility.

Document which HDL commit/hash corresponds to which bitstream version.

✅ Perform Controlled Hardware Bring-Up
Apply power, clocks, and reset signals while monitoring current and temperature.

Program the FPGA and look for “heartbeat” indicators (e.g., blinking LEDs or startup UART messages).

Begin by testing the most basic functionality — clock domains, resets, and I/O pin direction — before running the full application.

✅ Use Real-Time Debug Tools
Insert Integrated Logic Analysers (e.g., Xilinx ILA, SignalTap) into your design to probe internal signals without re-synthesis.

Use these tools to capture waveform snapshots in real-time and correlate against expected behaviour.

Monitor FSM states, counters, and internal bus traffic at key points in your design.

✅ Log and Validate All Outputs
Check actual I/O behaviour using an oscilloscope, protocol analyzer, or logic analyzer.

Use checksum, CRC, or known-good vectors to validate communication links (e.g., SPI, UART, Ethernet).

Automate hardware tests where possible with Python or LabVIEW host-side scripts.

✅ Iterate Intelligently
Log all issues discovered during hardware testing.

Update RTL, constraints, or testbenches to reflect these findings.

Re-run synthesis and implementation — and re-verify that the fix didn’t break timing or logic elsewhere.

Use incremental builds were supported to reduce turnaround time.

❌ Common Pitfalls
Assuming the bitstream “just works” because the simulation passed.

Forgetting to validate I/O standards, pull-ups, or drive strengths.

Ignoring power sequencing or overcurrent symptoms.

Failing to verify reset behaviour across asynchronous domains.

Not documenting changes made during on-board debug, leading to non-repeatable designs.

Step 13: Requirement Compliance

Requirement compliance is a key pillar of the MBSTech FPGA design methodology, ensuring that all design outputs meet defined functional, performance, and regulatory expectations. Early alignment with system requirements reduces the risk of late-stage rework and promotes traceability throughout the project lifecycle.

Key Practices for Ensuring Compliance:

✅ Requirement Traceability: Maintain a bi-directional traceability matrix linking each system-level requirement to specific design modules, test cases, and verification activities.

✅ Design Specification Review: Before coding begins, ensure that the functional and timing requirements are clearly understood, reviewed, and approved.

✅ Assertions and Coverage: Use formal assertions and functional coverage to validate compliance during simulation and runtime verification.

✅ Constraints and Timing Checks: Apply accurate timing constraints (e.g., .sdc, .xdc) to enforce setup/hold timing and clock domain separation.

✅ Simulation Against Expected Behaviour: Simulate the design using testbenches that model real-world use cases, boundary conditions, and edge cases.

✅ Validation with Hardware-in-the-Loop (HIL): Where possible, test the design against hardware prototypes or emulators to ensure requirements are met in real time.

Common Pitfalls

Despite best intentions, teams can fall into traps that undermine effective requirement compliance. MBSTech actively works to avoid the following common issues:

❌ Lack of Requirement Traceability

Designs often evolve beyond the original specification without updating the documentation or traceability matrix. This can result in "orphan" features or missing functionality.

❌ Ambiguous or Incomplete Requirements Vague language like “as fast as possible” or “should operate reliably” makes it difficult to verify compliance. Requirements must be specific, measurable, and testable.

❌ Late Validation

Waiting until hardware testing to check compliance can result in expensive rework. Requirement verification should be incremental throughout the simulation and synthesis phases.

❌ Unconstrained Design

Missing or incorrect timing constraints can lead to false positives in simulation and timing analysis, causing real hardware to fail even if simulations pass.

❌ Over-reliance on Manual Testing

Manual test procedures can overlook edge cases or regressions. Automated testbenches and coverage tools help ensure thorough and repeatable validation.

❌ Insufficient Coverage Metrics

Passing a few test cases doesn't guarantee full compliance. Without functional and code coverage metrics, gaps may go undetected.

❌ Non-Compliance with Standards

Failing to align with coding or industry-specific safety standards (e.g. DO-254, ISO 26262) can lead to audit failures or product certification issues.

❌ Lack of Configuration Management

Changes to IP, tool versions, or constraint files without proper version tracking can lead to inconsistent compliance results across builds.

Step 14: Design Archival and Environment Preservation

The final — and often overlooked — phase of an FPGA development lifecycle is archival. Archiving your work properly ensures that future updates, maintenance, certification

reviews, or product variants can be undertaken confidently and without painful reverse engineering.

✅ Archive the Entire Design Flow
Save all RTL code, testbenches, constraints, scripts, and simulation data.

Include:

*.vhd / *.v / *.sv

*.xdc / *.sdc constraint files

Synthesis, implementation, and bitstream artifacts

Golden test vectors and waveforms

Coverage reports, timing reports, and build logs

Archive both source and generated outputs in case regeneration becomes impossible due to environment changes.

✅ Version and Tag Releases
Use semantic versioning (v1.2.0, etc.) and tag each release in version control (e.g., Git).

Include a CHANGELOG.md and README.md summarising each release's state and known limitations.

✅ Preserve the Design Environment
Document:

Toolchain versions (e.g., Vivado 2022.1, Quartus Prime 21.3)

Operating system version (e.g., Windows 10 Build 19044)

Python/Perl versions if test automation was involved

Archive:

Tool installer packages or download links

Docker images or virtual machines (if used)

Batch files or environment setup scripts

⚠️ Note: Tool Version Drift Is Real
FPGA toolchains often do not guarantee forward-compatibility.

A design built in Vivado 2018.3 may not open or synthesise identically in Vivado 2024.1.

Likewise, licensing changes, OS-level DLL compatibility, and Java/driver dependencies can break older projects on newer PCs.

✅ Keep Project Snapshots and Configuration Files
Store .qsf, .xpr, .prj, .tcl, and any auto-generated environment files.

These contain settings that affect optimisation, timing, IP block generation, and more.

✅ Use Long-Term Storage Best Practices
Compress projects using consistent formats (.zip, .tar.gz)

Store copies on:

Internal file servers

Off-site/cloud backup

Write-once media (e.g., Blu-ray M-DISC for extreme longevity)