BLINKNBUILD ELECTRONICS · EMBEDDED SYSTEMS · MAKER CONTENT

FPGA Master Guide

Complete Internal Architecture & How an FPGA Actually Works — LUTs, Flip-Flops & Routing → Memory, DSP & Clocking → RTL → Synthesis → Placement → Routing → Bitstream → Physical Operation

PRIORITY MUST must know HIGH high probability GOOD good to know BONUS bonus // SCOPE GEN true of FPGAs generally AMD INTEL LATTICE MICROCHIP family-specific example, never a universal rule

How to read this sheet. Every architectural claim carries a scope chip. GEN means it holds for FPGAs in general. A vendor chip means **this is one family's answer**, quoted so you have a concrete number — not a definition. The single fastest way to fail an FPGA interview is to state one vendor's slice structure as though it were the definition of an FPGA.

// §1 WHAT AN FPGA ACTUALLY IS

1.1 Reading the name MUST

Field	Out in the world, after the chip has been manufactured, packaged and soldered down. Not in a fab.
Programmable	Its **internal structure** is set by data you supply. Not its instruction stream — it has none.
Gate Array	A large, regular array of identical logic resources, laid out in rows and columns, with programmable wiring between them.

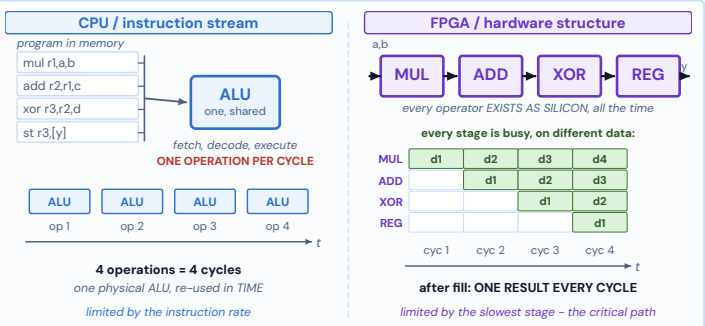
SIMPLE A chip full of blank logic blocks and blank wires. You supply a file that decides what each block computes and which wires connect to which. After that it behaves like the circuit you described.

ENGINEERING A configurable hardware fabric: an array of programmable logic elements embedded in a programmable interconnect network, with dedicated hard blocks for memory, arithmetic, clocking and I/O. Configuration state is held in on-chip memory that a bitstream writes.

PHYSICAL INTUITION The transistors are all fixed at manufacture. What the bitstream changes is the **stored value** at millions of tiny memory points, and each of those values holds a multiplexer select or a pass transistor gate either on or off. The chip does not rewire itself; it switches between wiring options that were etched in silicon years ago.

1.2 The question everyone gets wrong MUST

[TRAP] "Does an FPGA execute my Verilog?" No. Nothing inside an FPGA reads, decodes or executes your HDL — or the bitstream, for that matter. Your HDL **describes** a circuit. The tools build that circuit out of the resources this particular device happens to have, and write out the settings for every configurable point. The FPGA loads those settings once and then simply **is** that circuit, continuously, in parallel, until you power it down or load a different bitstream.



The one sentence to keep. A CPU is a fixed circuit that changes its **behaviour over time** according to an instruction stream. An FPGA is a **configurable circuit** that is built once and then behaves the same way every cycle, everywhere on the die, all at once.

1.3 What changes when you "program" an FPGA MUST

What is loaded	A **bitstream**: the value of every configuration bit in this exact device.
Where it goes	Into **configuration memory**, which is physically distributed across the whole die — beside every LUT, every routing switch, every I/O.
What it decides	LUT truth tables, routing multiplexer selects, flip-flop vs latch, reset polarity, I/O standards and drive, BRAM mode and initial contents, clock-manager divide and phase settings.
What does not change	Any transistor, any wire, any physical connection. Only stored bits change.

// §2 FPGA VS EVERYTHING ELSE

2.1 The comparison that gets asked MUST

Feature	FPGA	CPU	MCU	GPU	ASIC
Hardware structure	**configurable** fabric	fixed, general datapath	fixed core + fixed peripherals	fixed array of many simple cores	fixed, custom-built
Programmed by	a **bitstream** (structure)	machine code	machine code	kernels + machine code	nothing — it is what it is
Execution model	spatial: everything at once	temporal: one stream, fast	temporal, deterministic	SIMT: one instruction, many data	spatial, purpose-built
Parallelism	as much as fits on the die	a few cores, ILP	usually one core	thousands of threads	whatever was designed in
Latency	**very low, and predictable**	low, but cache-dependent	low and deterministic	high — batching is the point	lowest
Throughput	high for the right shape of work	moderate	low	very high for regular data	highest for its one job
Power / op	moderate — config overhead	high	very low absolute	high absolute	**lowest**
Flexibility	reconfigurable in the field	fully general software	general software	general within its model	**none**
Unit cost	high	moderate	very low	high	very low at volume
Development cost	moderate; long build times	low	low	moderate	**enormous** NRE
Time to first silicon	minutes to hours	n/a	n/a	n/a	months
Typical use	protocol bridging, real-time DSP, low-latency trading, emulation, aerospace, prototyping	general compute, OS	embedded control	graphics, dense linear algebra, ML training	high-volume products

Never say "an FPGA is faster than a CPU". It is a different *shape* of machine. An FPGA clocks far **slower** than a CPU — a few hundred MHz against several GHz — and wins only when the work is wide, regular and parallel enough that doing a thousand things at 200 MHz beats doing one thing at 4 GHz. On branchy, irregular, pointer-chasing code the CPU wins easily and the FPGA is a poor choice.

2.2 The honest decision table HIGH

Choose an MCU	Control, sequencing, a bit of maths, low cost, low power, and the timing is measured in microseconds.
Choose a CPU	Irregular work, complex decisions, an operating system, or when the answer is "just write software".
Choose a GPU	Huge batches of identical arithmetic where you can tolerate the latency of batching.
Choose an FPGA	Hard real-time deadlines, custom or odd interfaces, wide bit-level parallelism, deterministic sub-microsecond latency, low-to-medium volume, or a spec that is still moving.
Choose an ASIC	The design is frozen, the volume is large, and power or unit cost dominate everything else.

// §3 THE DIGITAL LOGIC YOU NEED FIRST

Only the parts that FPGA architecture is built on. If all of this is already familiar, skip straight to §4.

3.1 Combinational: output depends only on inputs now GOOD

Boolean logic	AND, OR, NOT, XOR over 0/1. Any function of N inputs can be written as a truth table with 2^N rows — which is precisely why a LUT is built the way it is.

Multiplexer	Selects one of several inputs. `y = sel ? a : b`. Universal: any Boolean function can be built from muxes alone. FPGA routing is built from them because a mux is the cheapest way to make a programmable choice.
Decoder	N inputs to one-of-2^N outputs. Address decode, one-hot state.
Adder	Sum and carry per bit. The carry is what makes it slow, and what dedicated FPGA carry logic exists to fix (§7).
Propagation delay	`tpd`: how long after an input changes before the output is finally stable. Nothing is instant.

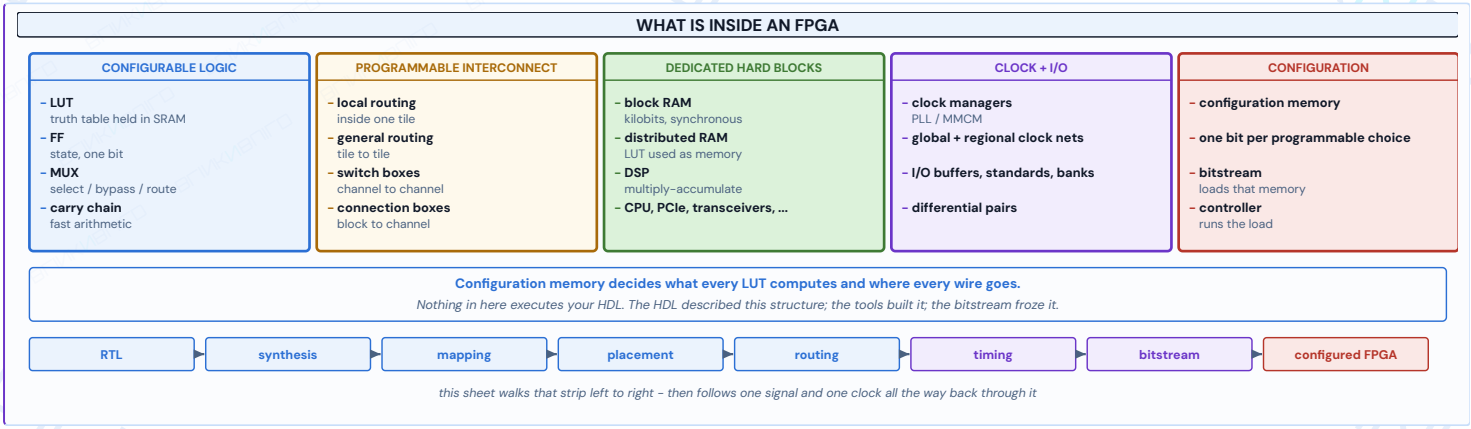
3.2 Sequential: output depends on history MUST

D flip-flop	On the active clock edge, Q takes the value of D. Between edges it holds. This is the only storage element that matters in FPGA design.
Register	N flip-flops sharing a clock. A bus-wide flip-flop.
Counter	A register whose next value is its current value plus one. Maps to registers plus a carry chain (§48.1).
FSM	A state register plus next-state logic plus output logic. Maps to flip-flops plus LUTs (§48.4).
Setup time `tsu`	How long D must already be stable **before** the active edge.
Hold time `th`	How long D must stay stable **after** the active edge.
Clock-to-Q `tcq`	From the active edge until Q is valid.

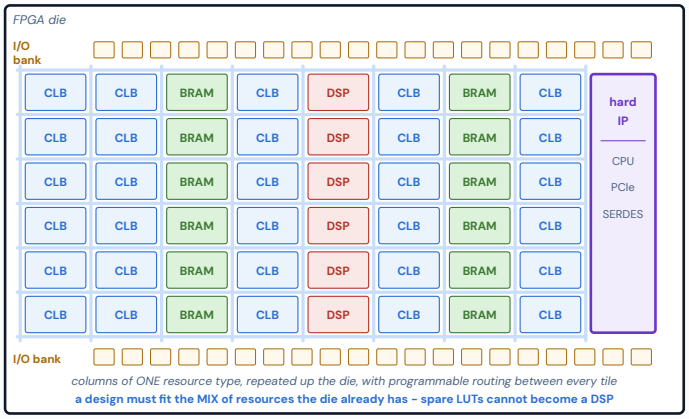
Why this is the floor. Every mainstream FPGA fabric is, at bottom, "a lookup table feeding a flip-flop, with programmable wires between them". (A few specialised families - notably antifuse parts - build their logic block from multiplexers instead, but the LUT is what you will meet.) Combinational logic becomes LUT contents; sequential logic becomes flip-flops; and setup, hold and propagation delay become the timing analysis that decides whether your design closes. Everything else in this sheet is detail on top of those three facts.

// §4 INSIDE THE DEVICE

4.1 The whole inventory, on one page MUST

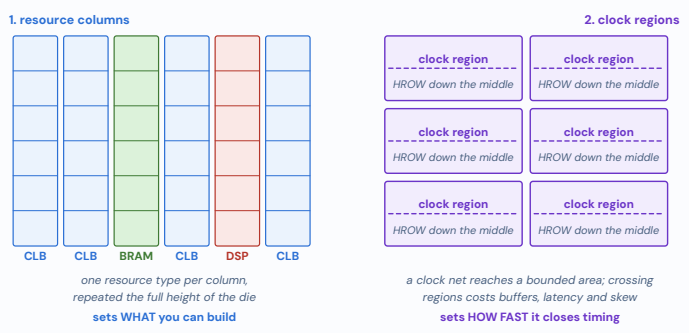


4.2 The physical arrangement MUST



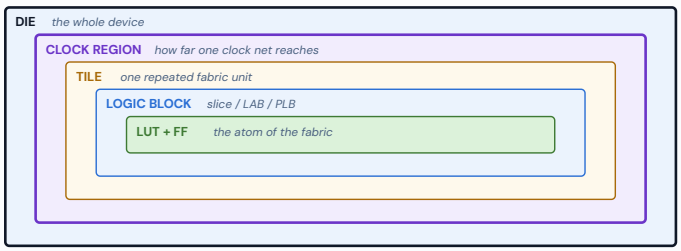
Why columns. Modern FPGAs are built from a small number of tile types repeated up the die in columns — logic, memory, arithmetic, I/O, transceivers. It makes the silicon cheap to lay out and easy to scale to a bigger part, and it is the reason a device is specified as "so many LUTs, so many BRAMs, so many DSPs": you are buying a fixed **mix**, and a design that needs a different mix will not fit even if the total area looks fine.

4.3 The two ways the die is divided HIGH



These two divisions are independent, and both constrain you. Resource columns decide **what** you can build; clock regions decide **how fast it will close timing**. A design that fits comfortably on resources can still fail because one clock has to reach across more regions than the network comfortably serves.

4.4 The containment hierarchy HIGH



Die	The whole piece of silicon. Some large parts are several dice on an interposer.
Region	A bounded area served by one set of clock resources. Crossing regions costs buffers, latency and skew.
Tile	The repeated unit: one logic tile, one BRAM tile, one DSP tile, each with its slice of the routing network attached.
Logic block	The vendor's grouping of LUTs, flip-flops, muxes and carry logic. Called a slice/CLB, a LAB/ALM, a PLB or a logic element depending on whose datasheet you are reading.
LUT + FF	The atom. Everything above is repetition of this.

4.5 Vendor vocabulary — a translation warning MUST

Term	Whose	What it actually names
`CLB`, `Slice`, `SLICEL`, `SLICEM`	AMD	the logic block and its two halves; SLICEM adds memory and shift-register modes
`LUT6`, `CARRY4`, `CARRY8`	AMD	a 6-input function generator; the 4- and 8-bit carry elements
`ALM`, `LAB`, `MLAB`, `ALUT`	INTEL	the adaptive logic module, the block of ten of them, and a LAB used as memory
`M20K`, `M10K`	INTEL	embedded memory blocks, named by their capacity in bits
`Block RAM`, `UltraRAM`, `LSRAM`, `uSRAM`	various	on-chip memory blocks — different sizes, different families
`DSP48`, `DSP58`, `Math block`, `variable-precision DSP`	various	hard arithmetic blocks — **different multiplier widths**
`MMCM`, `PLL`, `DLL`, `CMT`	various	clock management; the capabilities genuinely differ
`PFU`, `PLB`, `Logic Element`	LATTICE, MICROCHIP	their logic blocks — mostly LUT4-based

[TRAP] "How many LUTs does this design use?" is only meaningful **within one family**. A LUT6 fabric and a LUT4 fabric need different LUT counts for the same function, and an Intel ALM fractures differently again — so "one ALM" and "one slice" are not comparable units. Vendors publish their own "logic cell" or "logic element" figures precisely because raw LUT counts do not translate. `AMD` documents a ratio of **1.6 logic cells per 6-input LUT** for the 7-series, which tells you the marketing number and the physical number are deliberately different things.

4.6 What is *not* inside an FPGA `HIGH`

No instruction fetch	Nothing sequences through your design. There is no program counter unless you build one.
No operating system	Unless a hard processor is present (§23), and then it is the processor's OS, not the fabric's.
No dynamic allocation	Every resource is assigned at build time. A design cannot ask for another LUT at run time.
No implicit sequencing	Two `always_ff` blocks are two pieces of hardware running simultaneously — not two statements in an order.
Usually no on-die DRAM	On-chip memory is kilobits to a few tens of megabits. Bulk storage is off-chip, through a memory controller. Some high-end parts integrate HBM.

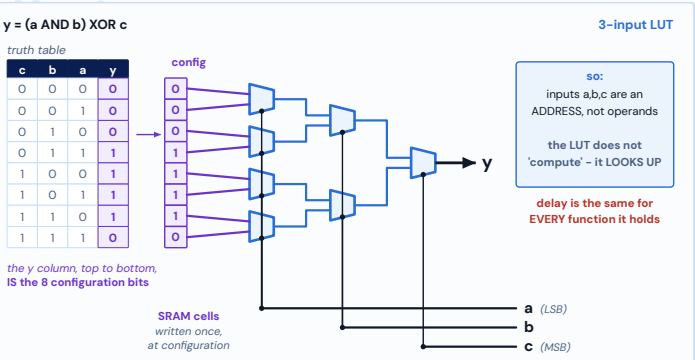
// §5 THE LUT — THE MOST IMPORTANT BLOCK IN THE DEVICE

5.1 What a LUT is `MUST`

`SIMPLE` A tiny memory holding a truth table. The inputs are the address; the stored bit at that address is the output.

`ENGINEERING` An N-input LUT is `2^N` bits of configuration memory read out through a multiplexer tree that the N logic inputs select. It implements **any** Boolean function of N inputs, because a truth table *is* any Boolean function of N inputs.

`PHYSICAL INTUITION` The configuration bits are the mux **data**; your logic signals are the mux **select**. Nothing computes. A path through a tree of transistors is opened, and one stored charge reaches the output.

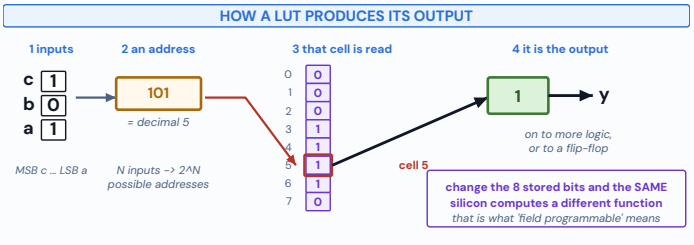


5.2 The arithmetic that follows from that `MUST`

Input combinations	`2^N` — a 4-input LUT has 16, a 6-input LUT has 64.
Configuration bits	**exactly** `2^N`, one per row of the truth table.
Functions it can hold	`2^(2^N)` — for N=4 that is 65,536; for N=6 it is about `1.8 x 10^19`.
Delay	**the same for every one of them.** The signal traverses the same mux tree whatever the stored bits are.

The consequence engineers actually use. `AMD`'s own CLB user guide states it plainly: the propagation delay through a LUT is **independent of the function implemented**. So on an FPGA a 6-input XOR costs exactly what a 6-input AND costs. "Simplify the Boolean expression" — the reflex you learned for gate-level design — buys you **nothing** unless it reduces the number of LUTs in series. What costs time is **LUT DEPTH**: how many LUTs a signal must pass through before it reaches a flip-flop.

5.3 How one evaluation actually happens `MUST`

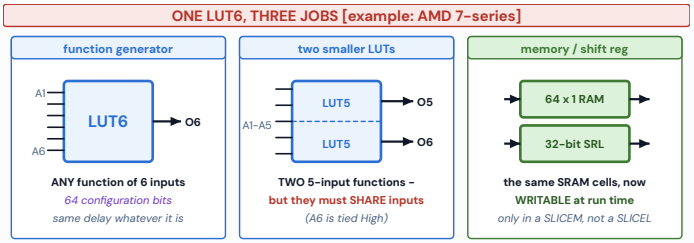


5.4 LUT size is an architectural choice `HIGH`

LUT size	Config bits	Trade	Seen in
LUT4	16	small, cheap, low power per LUT; needs **more** of them (and more routing hops) for a wide function	`LATTICE` iCE40, ECP5, Nexus; `MICROCHIP` PolarFire
LUT6	64	absorbs more logic per level, so shallower designs; each LUT is 4x the memory and often partly wasted	`AMD` 7-series, UltraScale, Versal
8-input fracturable	—	one 6-input function, or two independent 4-input functions, or selected 7-input ones — the tool packs to suit	`INTEL` ALM (Arria/Stratix/Agilex)

Bigger is not simply better. A LUT6 holding a 2-input AND uses 4 of its 64 bits to say anything; the other 60 are forced copies. Vendors answer that with **fracturing**: splitting one large LUT into two smaller independent ones. Research on FPGA architecture has long settled around 4–6 inputs as the sweet spot between logic depth and wasted silicon — which is why every mainstream fabric sits in that range.

5.5 A LUT is not only a function generator `HIGH`



Function generator	The normal case. Configuration bits are written once and then only read.
Fractured	Two smaller functions from one LUT. `AMD` 7-series: two 5-input functions on outputs O5 and O6 — but they must **share their inputs**, which is the catch.
Distributed RAM	The same SRAM cells, now writable at run time. Small, fast, and it consumes logic resources.
Shift register	The cells chained into a delay line. Extremely efficient — a 32-deep shift register in one LUT instead of 32 flip-flops.

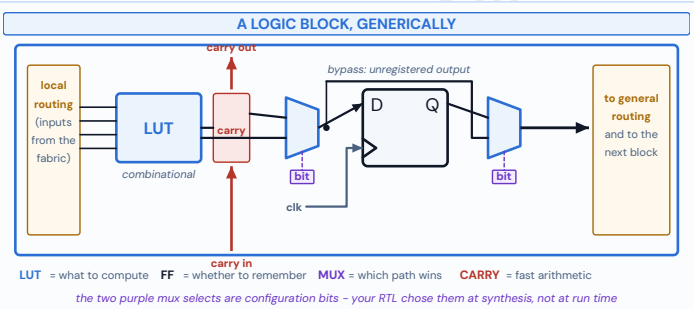
Worked example `AMD`. In the 7-series, roughly two-thirds of slices are **SLICEL** (logic only) and the rest are **SLICEM**, whose LUTs can additionally become 64-bit distributed RAM or a 32-bit shift register (SRL32, or two SRL16s). One CLB — two slices — therefore offers 8 LUTs, 16 flip-flops, 2 carry chains, and, if it is a SLICEM CLB, 256 bits of distributed RAM or 128 bits of shift register. That asymmetry is why a design heavy in small memories can run out of SLICEMs while LUT utilisation still looks low.

5.6 The mental model to carry forward `MUST`

LUT → the truth table your Boolean expression compiled to
purpose → implement any combinational function of N inputs in one delay
internal structure → `2^N` configuration cells feeding a multiplexer tree
signal flow → inputs address the tree, one stored bit reaches the output
RTL that produces it → any `assign`, any combinational `always`, any expression feeding a register
physical effect → a fixed, function-independent delay, plus routing to get in and out
timing consequence → what matters is the number of LUTs **in series**, not how complicated each one is

// §6 THE LOGIC BLOCK

6.1 What every vendor's logic block contains `MUST`

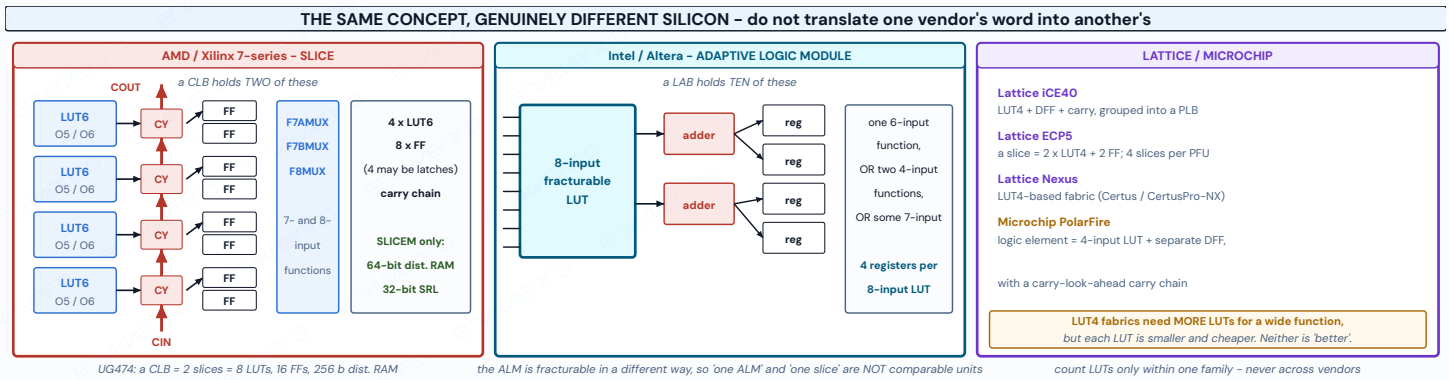


LUT(s)	the combinational function
Flip-flop(s)	the storage, one per LUT output or better
Multiplexers	registered or bypassed output, which input feeds what, wide-function building

Carry logic	a dedicated fast path from bit to bit, never through general routing

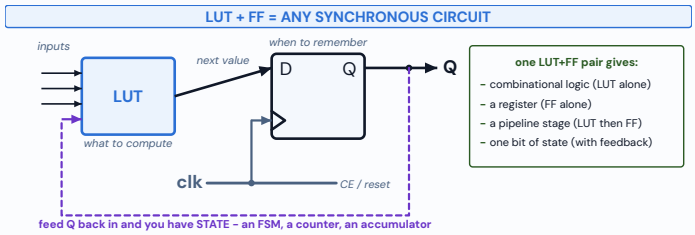
Local routing	short, cheap connections inside and between neighbouring blocks

6.2 The same concept, three vendors MUST



Do not manufacture equivalences. "One ALM equals one slice" is false. "A LUT6 equals one and a half LUT4s" is false. The only honest statements are structural: `AMD` 's 7-series slice holds four 6-input LUTs and eight storage elements, and a CLB holds two of them; `INTEL` 's ALM holds one 8-input fracturable LUT, two dedicated adders and four registers, and a LAB holds ten of them. Those are different silicon answering the same question.

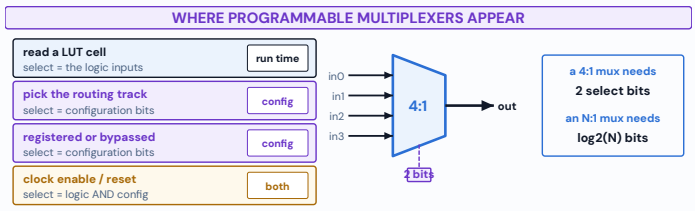
6.3 The flip-flop MUST



D, Q, clock	The core. On the active edge Q takes D.
Clock enable (CE)	A synchronous "hold". This is a **dedicated pin** on the flip-flop — using it costs nothing, whereas building the same hold as a feedback mux costs a LUT input.
Set / reset	Configurable synchronous or asynchronous, set or reset, per flip-flop — but often shared across a group, which is why mixing reset styles inside one block wastes resources.
Latch mode	Some fabrics allow it. `AMD` 7-series: four flip-flops per slice may become latches, and if you do, **the other four must go unused** — an expensive way to discover you inferred a latch by accident.

Flip-flops are nearly free; LUTs and routing are not. Every LUT comes with at least one flip-flop beside it, and most designs use far fewer flip-flops than LUTs. That asymmetry is the standing argument for pipelining (§34) and for one-hot state encoding (§48.4): both spend the resource you have spare to buy the one you do not.

6.4 Programmable multiplexers HIGH



Where the configuration memory really goes. It is tempting to picture configuration memory as mostly LUT contents. It is not. The great majority of configurable points in an FPGA are **routing multiplexer selects**, because the interconnect — not the logic — dominates both the die area and the bit count. This is also why bitstreams are large, and why routing is where the delay lives.

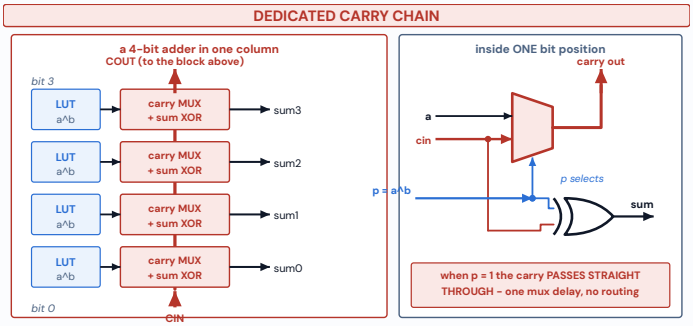
6.5 Wide functions GOOD

A function of more inputs than one LUT has must be built from several. Fabrics provide dedicated multiplexers to combine adjacent LUTs cheaply rather than paying for a routing hop.

`AMD` **7-series**	Three dedicated muxes per slice: `F7AMUX` and `F7BMUX` combine LUT pairs into any 7-input function; `F8MUX` combines all four into any 8-input function.
`AMD` **UltraScale**	Each LUT6 can act as a 4:1 mux; the eight LUTs of a slice can be combined into a 32:1 mux.
`INTEL`	The ALM's fracturable 8-input LUT covers all 6-input functions and selected 7-input ones directly.

// §7 THE CARRY CHAIN

7.1 Why it exists MUST



Without this, every carry would cross a switch box – four routing hops, not four mux delays.

It is why adders, counters and comparators are cheap on an FPGA – and why a chain cannot leave its column.

`SIMPLE` Adding is special enough that FPGAs have a private fast lane for it.

`ENGINEERING` A ripple adder built only from LUTs would send every carry out into the general interconnect and back — a switch-box round trip per bit. Dedicated carry logic gives each bit position a hard-wired path to the next, so an N-bit add costs N LUTs plus N cheap mux delays instead of N routing hops.

`PHYSICAL INTUITION` When the propagate term is true the carry passes straight through a single transmission gate. That is about as fast as a signal can move in the fabric, and nothing programmable is in the way.

7.2 What it is used for HIGH

Addition / subtraction	The obvious case. Subtraction is addition with an inverted operand and carry-in of 1.
Incrementers and counters	One operand is the constant 1, so the LUTs are trivial and the chain does the work.
Comparators	`a < b` is a subtraction whose carry-out you keep and whose sum you throw away.
Wide AND / OR	Some tools map very wide reductions onto the chain, because propagating along it beats a LUT tree.

7.3 The constraint it imposes MUST

[TRAP] A carry chain cannot leave its column. The dedicated wires only run vertically between adjacent logic blocks. A 64-bit adder is therefore a rigid vertical stack that the placer must fit somewhere, in order, with nothing in the way. That is why very wide arithmetic constrains placement far more than its LUT count suggests — and why breaking a huge adder into pipelined chunks often improves timing twice over: shorter chains *and* more placement freedom.

Design consequence. Prefer `+` , `-` , `<` and `==` written plainly in RTL. Synthesis recognises those shapes and reaches for the carry chain. Hand-built adders out of Boolean expressions frequently **lose** the chain and end up slower and larger than the operator you were trying to avoid.

// §8 PROGRAMMABLE ROUTING

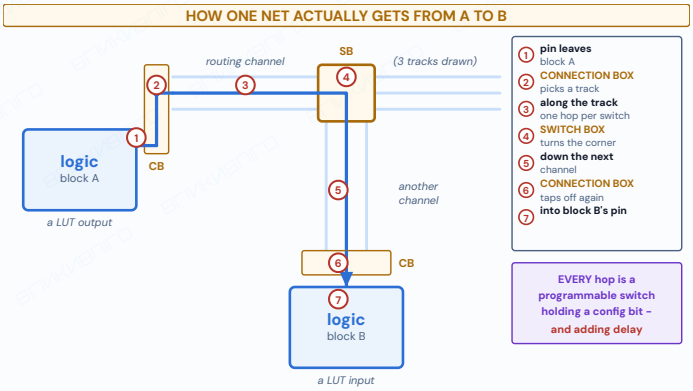
Routing gets more attention here than logic, because it consumes most of the die, most of the configuration bits and most of the delay — and because it is what every "my design fits but will not close timing" problem turns out to be about.

8.1 What routing is MUST

SIMPLE A grid of pre-built wires with programmable switches at the junctions. You do not draw wires; you choose which existing wires to switch together.

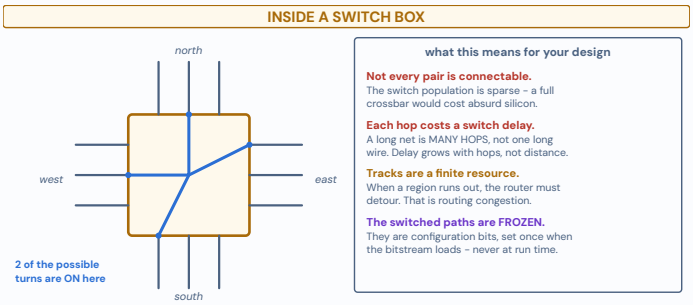
ENGINEERING Logic blocks sit as islands in a sea of routing channels. **Connection boxes** attach a block's pins to some of the tracks in the adjacent channel; **switch boxes** attach tracks in one channel to tracks in another. Both are populated sparsely, because a full crossbar is unaffordable.

PHYSICAL INTUITION Every "connection" is a transistor held open or closed by a configuration bit. A net is a chain of those, each contributing resistance and capacitance — so delay grows with the number of **hops**, not simply with distance.



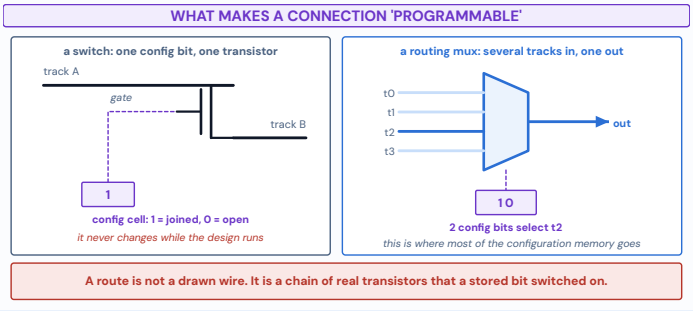
The tools chose this exact path. A different placement would choose a different one, with different timing.

8.2 Inside a switch box HIGH



A switch box joins channels; a connection box joins a block's pins to a channel. Different silicon.

8.3 What a programmable connection physically is MUST



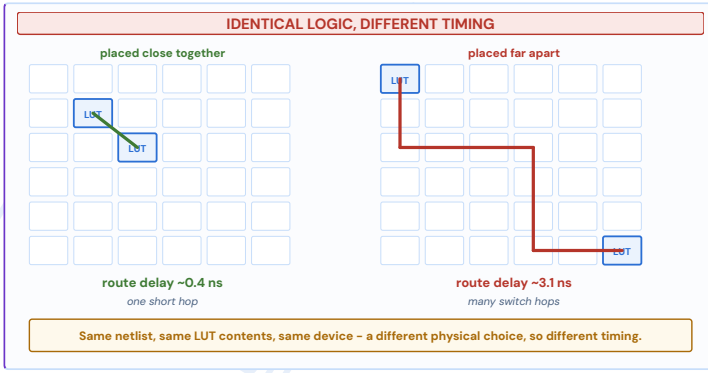
The distinction that matters. Routing is **part of the configured hardware**, not a software-level list of connections. After configuration there is no routing table anywhere; there are transistors that a stored bit turned on. This is why a connection cannot change while the design runs – short of reconfiguring that part of the device – why routing costs real delay, and why the router can fail even when there is plenty of logic left.

8.4 The routing hierarchy HIGH

Level	Reach	Cost	Used for
Local / direct	inside a block, or to an immediate neighbour	lowest delay, no switch box	carry chains, LUT-to-FF, packing related logic together
Single / double	a few tiles	one or two hops	most ordinary nets
Long lines	across a region or the die	fewer hops for the distance, but heavily loaded	buses, high-fanout control
Dedicated	fixed, point-to-point	fastest; not general purpose	carry chains, DSP cascades, clock networks

// §9 WHY ROUTING DOMINATES TIMING

9.1 The same logic, twice MUST



9.2 What actually makes a route slow MUST

Number of hops	Every switch is a transistor in series: resistance, plus the capacitance of the next segment. This is usually the dominant term.
Physical distance	Longer wires mean more capacitance to charge, and more hops to get there.
Fanout	Every extra load slows the edge. A net with 500 sinks is a different animal from one with 2 (§36).
Congestion	When a region's tracks are used up, the router detours — sometimes far. Congested designs have wildly variable timing between runs.
Placement	The root cause of most of the above. The router can only work with the distances the placer left it.

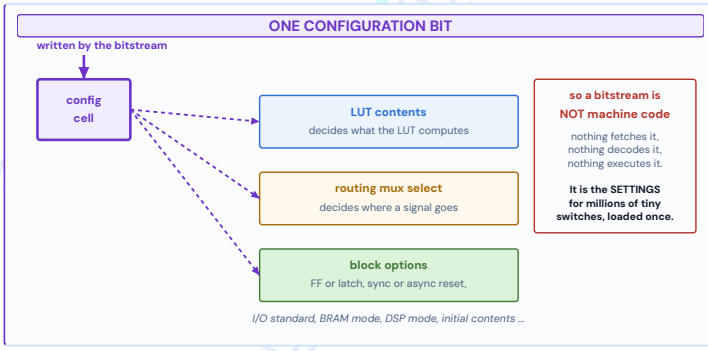
9.3 The consequences you must be able to state MUST

× Two logically identical RTL descriptions must have the same timing	✓ They can differ by a factor of several. Timing is a property of the **physical implementation**, not of the logic.
× Low utilisation means good timing	✓ A 30%-full design can fail badly if the logic is spread out, the critical path is deep, or one net has enormous fanout (§36).
× Re-running the tools should give the same result	✓ Placement and routing are heuristic searches. A different seed, a different tool version, or a trivial RTL edit can move timing either way.
× The LUT delay is what I should optimise	✓ On a long path the routing is usually the bigger half. Reducing **logic depth** helps mainly because it removes hops, not because LUTs are slow.

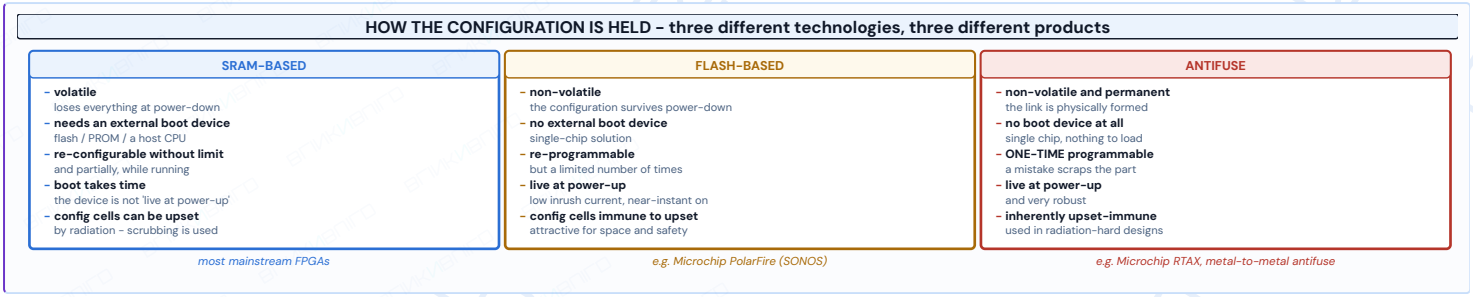
What to do about it. Shorten paths in **logic depth** first (pipeline, restructure), then in **physical distance** (keep related logic together, use hierarchy and floorplanning if the tool supports it), then in **fanout** (replicate drivers). Only after that is it worth arguing with the router.

// §10 CONFIGURATION MEMORY

10.1 One bit, three jobs MUST



10.2 The three technologies MUST



[TRAP] "All FPGAs are SRAM-based." Most mainstream ones are, and that shapes everything about them: they are volatile, they need an external boot device, they take time to come up, and their configuration cells are vulnerable to radiation-induced upsets. But flash-based and antifuse parts exist and are chosen precisely because they are none of those things. `MICROCHIP` PolarFire, for instance, is built on a 28 nm non-volatile SONOS process: its configuration cells are documented as single-event-upset immune, it needs no external boot device, and it is live at power-up with low inrush current. That is a different product, not a different brand of the same product.

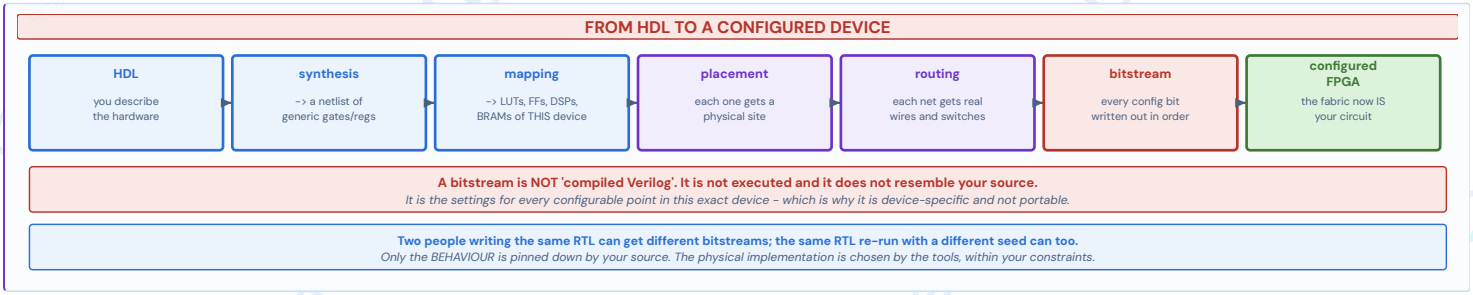
10.3 What configuration bits control `HIGH`

LUT contents	2^N bits per LUT — the truth table.
Routing	Every switch and every routing mux select. **The largest share by far.**
Block options	Flip-flop or latch; set or reset; synchronous or asynchronous; clock-enable polarity; which output is registered.
I/O	Direction, standard, drive strength, slew rate, termination, pull-up or pull-down.
Clocking	Which buffers are enabled, multiply and divide values, phase settings.
Memory	Width, depth, port mode, write mode — and the **initial contents**, so a BRAM can come up as a ROM.
DSP	Which pipeline registers are enabled, which operation is selected, cascade routing.

The exact bit-level format is architecture-specific and largely proprietary. You do not need it — you need to know that **everything programmable is a stored bit**, and that the total is measured in millions to hundreds of millions.

// §11 THE BITSTREAM

11.1 What it is, and what it is not `MUST`



× A bitstream is compiled Verilog	✓ It is the **settings** for every configurable point in one specific device. Your source is not recoverable from it in any meaningful sense.
× A bitstream is executed	✓ It is **loaded**, once. Nothing fetches or decodes it afterwards.
× A bitstream is portable	✓ It is tied to the exact device and often the exact package. A bitstream for one part will be rejected by another — the device ID check exists for that reason.
× The same RTL gives the same bitstream	✓ A different tool version, a different seed, a different constraint, even reordered source files can produce a different — equally correct — implementation.

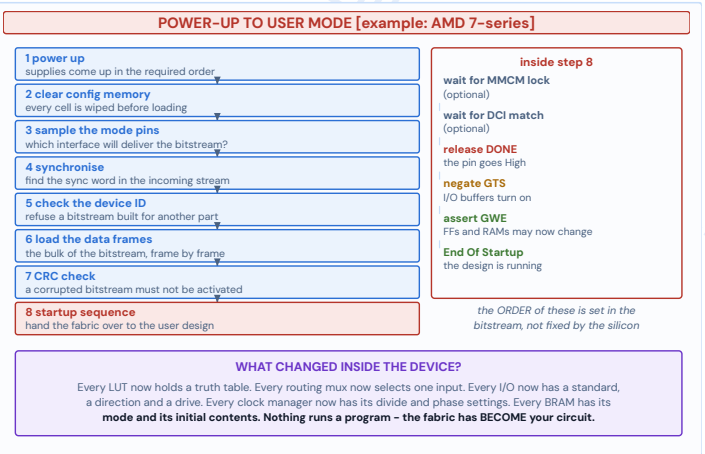
What is preserved and what is not. Your source pins down the **behaviour**. Everything about the physical realisation — which LUT, which site, which track, how the carry chain packed — is chosen by the tools inside the freedom your constraints leave them. That is the whole reason timing closure is an iterative activity rather than a compile step.

11.2 Bitstream security and integrity `GOOD`

CRC	Checked before the device is activated, so a corrupted load cannot bring up a half-configured, possibly contention-driving circuit.
Encryption	Most families support an encrypted bitstream with a key held in on-chip fuses or battery-backed RAM — this is the standard defence against cloning an SRAM FPGA by snooping its boot flash.
Authentication	Modern families additionally authenticate, so a valid-looking but foreign bitstream is rejected rather than merely garbled.
Readback	Many devices can read configuration memory back — useful for verification and for **scrubbing**: continuously re-checking and repairing cells upset by radiation.

// §12 WHAT HAPPENS WHEN YOU PROGRAM AN FPGA

12.1 Power-up to user mode `MUST`



12.2 Where the bitstream comes from `HIGH`

Mode	Who drives the clock	Typical use
Master SPI / BPI	the FPGA	production: the device boots itself from an attached flash
Master / slave serial	FPGA / external	simple boards, daisy-chained devices
Parallel (e.g. SelectMAP)	external	a host CPU loads the fabric quickly, sometimes at run time
JTAG	external	development, debug, and factory programming
Internal port	the fabric itself	partial reconfiguration — a running design swaps part of itself

`AMD` 7-series names these steps explicitly: power-up, clear configuration memory, sample the mode pins, synchronise, check the device ID, load the data frames, CRC, and then startup. The mode pins are sampled in step 3, which is why they must be strapped correctly before power is applied — you cannot change your mind later.

12.3 The startup sequence, in detail HIGH

The last step is not one event but an ordered handover, and the order is itself set by the bitstream. In AMD 7-series terms the user-selectable events are:

wait for MMCM lock	optional — hold everything until the clock managers are stable
wait for DCI match	optional — hold until on-die termination has calibrated
release DONE	the open-drain DONE pin goes High; the board can watch this
negate GTS	Global 3-State is removed — **the I/O buffers turn on** and the device starts driving pins
assert GWE	Global Write Enable — flip-flops and RAMs may now change state
End Of Startup	the design is running

Why the order matters on a real board. Between "I/O turns on" and "flip-flops may change" there is a window in which the device is driving pins from its **initial** state. Get the ordering or the initial values wrong and you can drive a bus that something else is already driving. This is also why every output should have a defined power-on state, and why boards commonly hold external devices in reset until DONE is seen High.

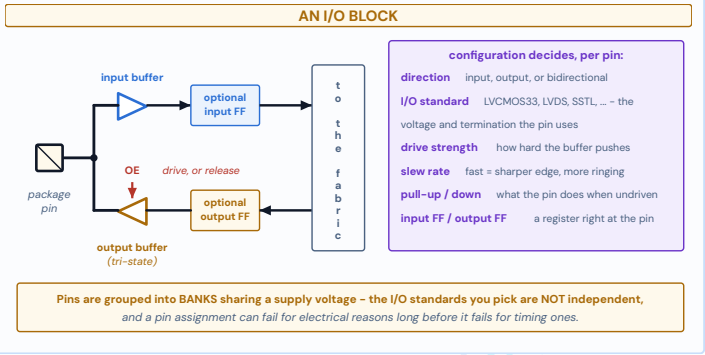
12.4 So what changed inside? MUST

Every LUT now holds a truth table. Every routing multiplexer now selects one input. Every I/O has a direction, a standard and a drive. Every clock manager has its multiply, divide and phase settings. Every BRAM has its mode and its initial contents. Every DSP has its pipeline registers enabled or bypassed.

Nothing is "running a program". The fabric has **become** the circuit you described, and it will behave that way on every clock edge until power is removed.

// §13 I/O BLOCKS

13.1 What sits between a pin and the fabric MUST



13.2 The electrical settings that are yours to choose HIGH

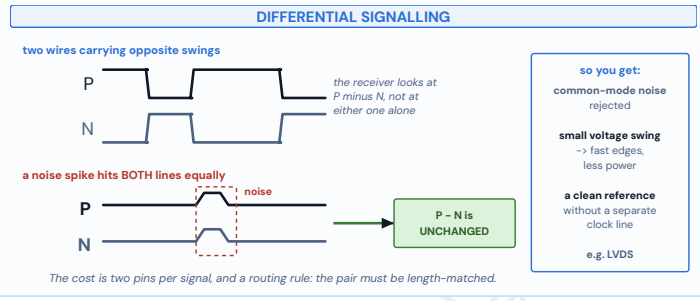
I/O standard	LVCMOS33, LVCMOS18, LVTTL, SSTL, HSTL, LVDS and so on. Sets the switching threshold, the output swing and the termination. **It must match the other end of the wire.**
Drive strength	How hard the output buffer pushes, usually in mA. More drive means faster edges into a load — and more ground bounce.
Slew rate	Fast or slow. Fast edges meet tight timing; slow edges radiate less and ring less. Use slow unless you need fast.
Pull-up / pull-down	What the pin does when nothing is driving it — including before configuration finishes.
Termination	Many families offer on-die termination, calibrated at startup (the "wait for DCI" step in §12.3).
I/O registers	A flip-flop physically in the I/O block. Using it gives a fixed, tightly characterised pin-to-register time; leaving it out puts routing into your I/O timing.

13.3 Banks — the constraint people trip over MUST

[TRAP] Pin assignment is an electrical problem before it is a timing problem. Pins are grouped into **banks**, and every pin in a bank shares the same VCCO supply. So the I/O standards you pick are **not independent**: you cannot put a 3.3 V LVCMOS pin and a 1.8 V one in the same bank. Some pins are further reserved — configuration pins, clock-capable pins, and the pins a hard block is wired to. A pinout that looks fine on the schematic can be rejected outright by the tools.

Clock-capable pins	Only certain pins have a path to the clock resources. Bring a clock in on an ordinary pin and it will have to reach the clock network the slow way, or not at all (§15).
Differential pairs	Fixed P/N pin pairs. You cannot form a pair from any two pins you like.
Simultaneous switching	Many outputs switching together inject noise into the shared supply. Vendors publish SSO limits per bank; exceeding them causes failures that look like nothing else.

// §14 DIFFERENTIAL SIGNALLING

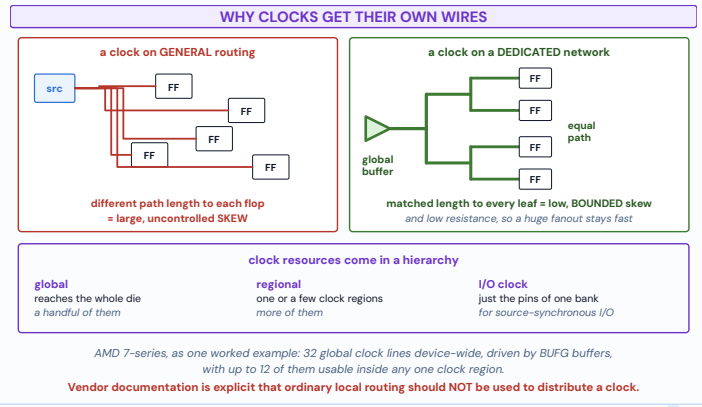


How it works	Two wires carry complementary swings. The receiver responds to the **difference**, so anything common to both lines cancels.
Common-mode rejection	Noise coupled into a closely routed pair lands on both wires nearly equally, so it disappears from the difference.
Small swing	Because the receiver only needs a small difference, the swing can be a few hundred millivolts. Smaller swing means faster edges and less power per transition.
LVDS	The common low-voltage differential standard. Current-mode drive into a termination resistor at the receiver.
The cost	Two pins per signal, a fixed pin pair, and a routing rule: the two traces must be **length-matched** and routed together, or the differencing stops working.

Where you meet it. Camera and display interfaces, high-speed ADCs and DACs, source-synchronous memory interfaces, and every multi-gigabit serial link (§24). Once a signal is fast enough that single-ended signalling would need a clean ground return and a matched impedance anyway, differential costs almost nothing extra and buys a great deal.

// §15 CLOCK ARCHITECTURE

15.1 Why a clock does not travel on ordinary routing MUST



SIMPLE A clock has to arrive everywhere at the same instant. General routing cannot promise that, so FPGAs build a separate, balanced network just for clocks.

ENGINEERING Dedicated clock trees are length-matched, low-resistance and heavily buffered, giving bounded skew across thousands of loads. They are also characterised tightly, so static timing analysis can model them accurately instead of pessimistically.

PHYSICAL INTUITION A clock is the highest-fanout, highest-activity net in the design — it toggles every cycle into every flip-flop. Putting that on the same wires as data would be both slow and a serious power problem.

[TRAP] Do not generate clocks in the fabric. A clock produced by a LUT — a divided clock, a gated clock, a "clock" that is really a control signal — reaches the flip-flops through **general routing**. Vendor documentation is explicit that ordinary local routing should not be used for clock distribution. The result is uncontrolled skew, a clock domain nobody constrained, and failures that appear only on some boards at some temperatures. Use a clock manager, or a clock enable.

15.2 Use a clock enable, not a gated clock MUST

Gated clock (wrong on an FPGA)	Clock enable (right)
The clock passes through a LUT and onto general routing.	One clock, on the dedicated network, reaching every flop.
Creates a new, skewed, unconstrained clock domain.	No new domain. Timing analysis is unchanged.
Glitches on the gate become clock edges.	A glitch on CE is simply sampled like any other signal.
Costs LUTs and routing.	CE is a **dedicated pin** on the flip-flop. It costs nothing.

15.3 The clock resource hierarchy HIGH

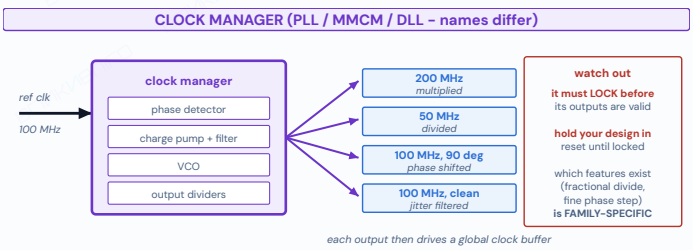
Resource	Reach	How many	For
Global	the whole die	few — a scarce resource	main system clocks
Regional	one or a few clock regions	more	local domains, divided clocks
I/O clock	the pins of one bank	more still	source-synchronous interfaces

AMD 7-series, as a concrete example: the device is divided into 8 to 24 clock regions depending on size; a region spans 50 CLBs and one 50-pin I/O bank with a horizontal clock row through its centre; there are **32 global clock lines** device-wide driven by BUFG buffers, of which any one region can use up to 12. Those numbers are family-specific — the **shape** of the answer is not.

Global clock buffers are countable, and you can run out. Every distinct clock — every PLL output, every divided clock, every recovered clock from a transceiver — wants one. A design with a dozen clock domains can exhaust them, at which point the tools quietly put a clock on ordinary routing and your timing collapses. Check the clock utilisation report, not just the LUT one.

// §16 CLOCK MANAGERS: PLL, MMCM, DLL

16.1 What they do MUST



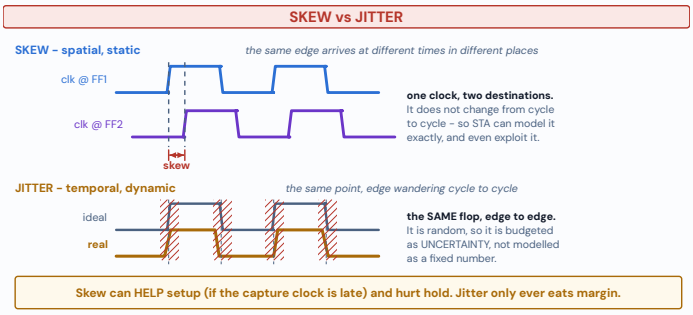
Frequency multiplication	A VCO runs at a high frequency locked to the reference; dividers bring it back down. This is how one 100 MHz crystal becomes 200 MHz, 50 MHz and 33.3 MHz at once.
Frequency division	Cheap and exact. Prefer this to a fabric counter whenever the divided signal is used as a clock.
Phase shift	Fixed or dynamic. Essential for source-synchronous interfaces, where you must place a clock edge in the centre of a data eye.
Jitter filtering	The loop filter attenuates high-frequency noise on the reference.
De-skew	Feeding the network's own output back as the feedback path lets the manager cancel the network's insertion delay.
Duty-cycle correction	Supported on some families and outputs; check before relying on it.

What a clock manager cannot do. It cannot make the output more **accurate** than the reference. Frequency accuracy and long-term stability come from the crystal; the manager only synthesises ratios of it. If the specification needs 50 ppm, that is a crystal decision, not an FPGA one.

Always wait for LOCK. A clock manager's outputs are meaningless until it has locked, and lock takes time after configuration. Hold your design in reset until the lock signal asserts. Designs that ignore this work on the bench and fail intermittently at power-up, which is the worst possible failure mode to debug.

Capabilities differ by family and this is not a place to generalise. AMD 7-series pairs one MMCM and one PLL in each clock management tile, and documents the **PLL as a subset of the MMCM** — the MMCM adds fine phase shift in either direction, dynamic phase shifting, and a fractional counter for finer frequency synthesis. Other vendors divide the capability differently, and some parts also offer delay-locked loops, which correct phase without synthesising a new frequency.

// §17 SKEW, JITTER AND UNCERTAINTY



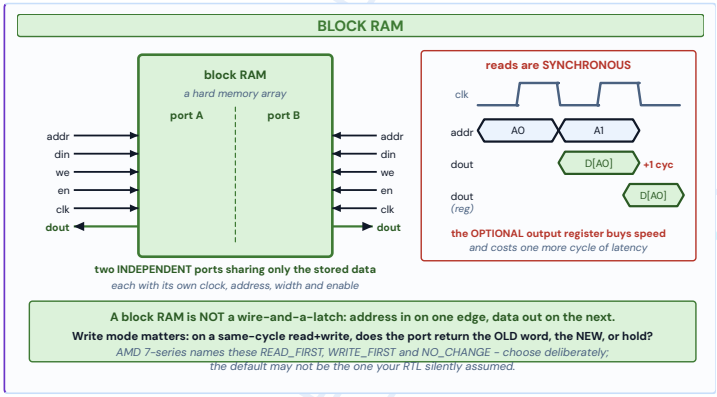
Concept	Meaning	Cause	Design impact
Skew	The same edge reaching two places at **different times**. Spatial and static.	Unequal clock path lengths and loading.	Positive skew (capture clock later) **helps setup, hurts hold**. Modelled exactly by STA; can be exploited deliberately as useful skew.
Jitter	The **same** point in the circuit seeing the edge wander cycle to cycle. Temporal and random.	Reference noise, supply noise, PLL loop behaviour, crosstalk.	Eats setup margin. Cannot be designed out — it is **budgeted** as uncertainty.
Uncertainty	The lump sum STA subtracts from the available period to cover jitter and modelling error.	Tool-derived, plus whatever you declare.	Under-declaring it makes reports optimistic — the design passes on paper and fails on the board.

The clean way to remember it: skew is a difference in **space**, jitter is a difference in **time**. Skew can be your friend; jitter never is.

// §18 BLOCK RAM

18.1 Why dedicated memory exists at all MUST

A LUT can be used as a small memory, so why harden anything? Because the numbers do not work. A 4 KB buffer is 32,768 bits. At 64 bits per LUT that is 512 LUTs used purely as storage, plus the address decoding and multiplexing to select among them, plus the routing to reach them all. A single hard memory block does the same job in a fraction of the area, at a fraction of the power, with a clean synchronous interface — and leaves the LUTs for logic.



18.2 The interface MUST

`addr`	Which word. Its width sets the depth.
`din` / `dout`	Data in and out. Width is configurable, and the two ports may differ.
`we`	Write enable, often per byte lane.
`en`	Port enable — deasserting it saves real power, which matters because memory is a power hotspot.
`clk`	**Every** access is synchronous. There is no asynchronous read.

18.3 Port modes HIGH

Single port	One address, one clock. Read or write per cycle.
Simple dual port	One write port, one read port — usually with independent clocks. The natural shape for a FIFO.
True dual port	Two fully independent ports, each with its own clock, address, width and enable, sharing only the stored data. The natural shape for a shared buffer between two subsystems.

18.4 Read latency and write mode — where bugs live MUST

[TRAP 1] Block RAM reads are synchronous. Present the address on one edge; the data appears **after** that edge. Enable the optional output register and it is one cycle later still. Designs written as though `dout` follows `addr` combinationally simulate fine at RTL and fail the moment they meet a real block.

[TRAP 2] What happens on a same-cycle read and write to the same address? There is no universal answer — it is a configured **write mode**. AMD 7-series names three: `WRITE_FIRST` (transparent — the port returns the new data, and it is the default), `READ_FIRST` (the port returns the old data), and `NO_CHANGE` (the output holds). Choose deliberately. Assuming the wrong one gives an off-by-one-cycle bug that is very hard to see in a waveform.

Collisions on a true dual-port block. Writing the same address from both ports in the same cycle gives an **undefined** stored result, and reading while the other port writes can return undefined data. The hardware does not arbitrate for you. If two domains can touch the same word, you need a protocol, not hope.

18.5 Sizes are family-specific GOOD

Family	Block	Notes
AMD 7-series	**36 Kb**	usable as one 36 Kb block or two independent 18 Kb blocks; many width/depth configurations; optional ECC and a hard FIFO mode
AMD UltraScale+	36 Kb + **UltraRAM 288 Kb**	a second, much larger memory type for deep buffers
INTEL	**M20K**, plus **MLAB**	MLAB is a LAB used as memory — the ALM equivalent of distributed RAM
MICROCHIP PolarFire	**LSRAM 20 kbit**, **uSRAM 64x12**	two sizes, plus non-volatile uPROM and sNVM on the same die

// §19 DISTRIBUTED RAM VS BLOCK RAM

You get whichever the synthesis tool infers: small arrays in LUTs, big ones in block RAM –
and you can force the choice with an attribute when the tool guesses wrong for your timing or area budget.

Feature	LUT / distributed RAM	Block RAM
Capacity	tens of bits per LUT	kilobits per block
Resource used	**logic** — it competes with your LUTs	a separate, fixed pool
Read	**asynchronous** on many families	always **synchronous**
Write	synchronous	synchronous
Latency	can be zero cycles	one cycle, or two with the output register
Placement	anywhere there are SLICEMs / MLABs	only in the memory columns
Best for	register files, tiny FIFOs, delay lines, wide-and-shallow tables	frame and line buffers, packet FIFOs, coefficient tables, anything deep

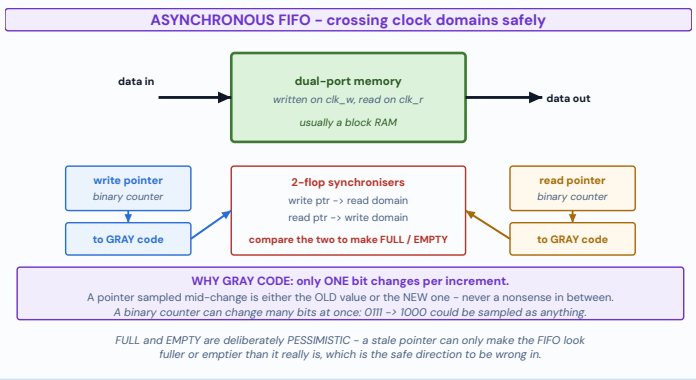
You mostly get whichever one the tool infers — small arrays land in LUTs, large ones in block RAM, with a threshold you can move. Every vendor also provides an attribute to force the choice. Reach for it when the tool's guess costs you timing, or when a shallow memory is eating LUTs you need elsewhere.

// §20 FIFOS

20.1 Synchronous FIFO HIGH

One clock. A write pointer, a read pointer, and a memory between them. Full and empty come from comparing the two pointers directly. The only real subtlety is that "pointers equal" means both full and empty, so you need one extra bit of pointer, or a separate count, to tell the two apart.

20.2 Asynchronous FIFO MUST



The problem	Each pointer lives in its own clock domain, but the flag logic needs to compare them. So one pointer must be sampled by the other domain's clock — a clock domain crossing of a multi-bit value, which is exactly the dangerous case (§38).
Gray coding	Only one bit changes per increment. A pointer sampled mid-change therefore reads as either the old value or the new one — never a nonsense combination. A binary counter going `0111 -> 1000` changes four bits at once and can be sampled as any of sixteen values.
Two-flop synchronisers	Each Gray-coded pointer is registered twice in the destination domain before it is used.
Pessimistic flags	The synchronised pointer is always slightly stale, so FULL can only ever assert early and EMPTY can only ever de-assert late. Both errors are in the **safe** direction — the FIFO may under-use its depth, but it will never overflow or under-run.

Why this is the standard answer for CDC. An asynchronous FIFO turns "how do I move a data stream between two unrelated clocks?" into a well-understood, verifiable, vendor-supplied component. Use the vendor's FIFO generator unless you have a specific reason not to. Hand-written asynchronous FIFOs are a classic source of bugs that appear once a week in the field and never on the bench.

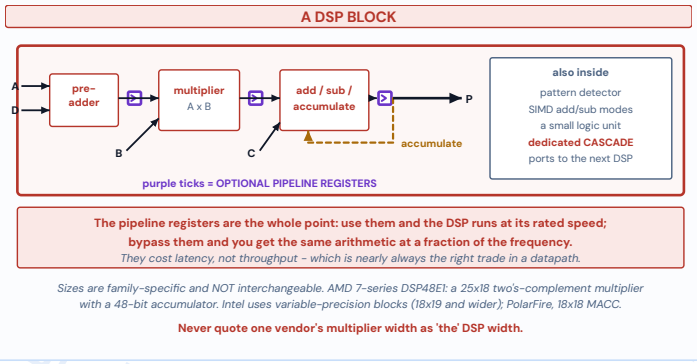
// §21 DSP BLOCKS

21.1 What a DSP block is MUST

`SIMPLE` A hard multiplier with an adder and a register bolted on, because multiplying in LUTs is expensive and everybody wants to multiply.

`ENGINEERING` A fixed-function arithmetic pipeline: an optional pre-adder, a multiplier, a multi-input adder/subtractor/accumulator, optional pipeline registers at every stage, and dedicated cascade ports to the neighbouring block.

`PHYSICAL INTUITION` A multiplier is a large, regular, heavily used structure. Hardening it buys roughly an order of magnitude in area, speed and energy over the same function built from LUTs and carry chains.

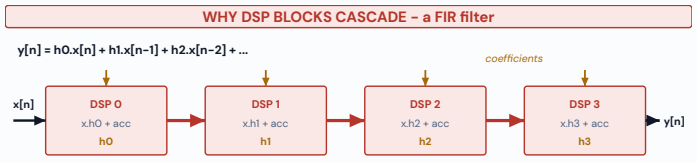


The pipeline registers are the whole point: use them and the DSP runs at its rated speed; bypass them and you get the same arithmetic at a fraction of the frequency.
They cost latency, not throughput - which is nearly always the right trade in a datapath.

Sizes are family-specific and NOT interchangeable. AMD 7-series DSP48E1: a 25x18 two's-complement multiplier with a 48-bit accumulator. Intel uses variable-precision blocks (18x19 and wider); PolarFire, 18x18 MACC.

Never quote one vendor's multiplier width as 'the' DSP width.

21.2 Why it cascades HIGH



The red links between blocks are DEDICATED CASCADE WIRES inside the DSP column –
they do not use general routing at all, which is why a long filter stays fast.

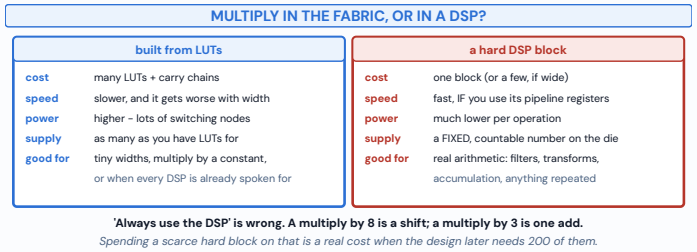
The same structure underlies FIR and IIR filters, FFT butterflies, matrix multiply, convolution layers and control-loop maths. It is one shape, reused - which is exactly why the silicon is worth hardening.

A cascade only works if the tools can PLACE the chain in one column - a huge filter can still fail to fit.

21.3 The pipeline registers are the point MUST

[TRAP] Instantiating a DSP does not make anything fast. The block's rated frequency assumes its internal pipeline registers are **in use**. Write `p = a * b + c;` as one combinational expression and the tool will happily map it to a DSP with every register bypassed — the same arithmetic, at a fraction of the frequency, and now you have also spent a scarce hard block to get there. Register the inputs, the product and the accumulator. The registers exist whether you use them or not.

21.4 LUT multiplier vs DSP multiplier HIGH



'Always use the DSP' is wrong. A multiply by 8 is a shift; a multiply by 3 is one add.
Spending a scarce hard block on that is a real cost when the design later needs 200 of them.

"Always use the DSP" is wrong. Multiplying by 8 is a shift. Multiplying by 3 is `x + (x << 1)` — one adder. Multiplying by a constant generally collapses into a handful of shifts and adds that the fabric does very cheaply. Spending an irreplaceable hard block on that is a real cost in a design that later turns out to need two hundred of them. Conversely, a 32x32 multiply may need **several** DSPs stitched together, so "one multiply, one DSP" is not a reliable count either.

21.5 Widths are family-specific MUST

Family	Block	Core arithmetic
AMD 7-series	DSP48E1	**25 x 18** two's-complement multiplier, 48-bit accumulator, pre-adder, SIMD (dual 24-bit or quad 12-bit), a ten-function logic unit, and a pattern detector
AMD UltraScale	DSP48E2	a wider multiplier than the E1, plus a wider pre-adder
AMD Versal	DSP58	wider again, with floating-point and complex-arithmetic modes on some variants
INTEL	variable-precision DSP	configurable — e.g. 18x19 modes, with two 9x9 multipliers per 18x19, and wider modes built from them
MICROCHIP PolarFire	Math block	**18 x 18** MACC

Never quote one vendor's multiplier width as "the" DSP width. The asymmetry in AMD's 25x18 is itself a design decision — it suits filter coefficients on one input and data on the other. Intel's variable precision is a different decision for the same market. If an interviewer asks "how wide is a DSP block?", the correct answer names a family.

21.6 What DSP blocks are used for GOOD

Filtering	FIR and IIR — the canonical multiply-accumulate cascade.
Transforms	FFT butterflies, DCT, correlation.
Image and video	Convolution kernels, colour-space conversion, scaling.
Machine learning	Dense matrix-vector products; low-precision modes are why SIMD exists in these blocks.
Motor and power control	Park/Clarke transforms and PID loops, at rates and with a determinism a processor struggles to match.
Not just DSP	The 48-bit accumulator makes an excellent wide counter; the pattern detector makes a comparator; the logic unit does wide bitwise operations. Tools sometimes map non-arithmetic logic into spare DSPs to relieve LUT pressure.

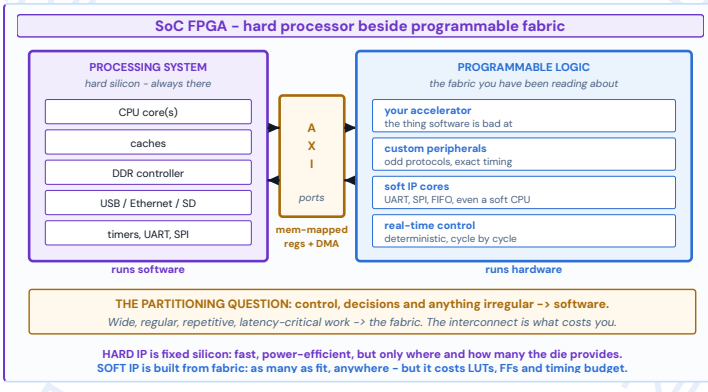
// §22 HARD IP, SOFT IP, REUSABLE IP

22.1 The three meanings of "IP" MUST

Hard IP	Dedicated silicon, laid down at manufacture. Fast, small, power-efficient — and **fixed in quantity and position**. If the die has two PCIe blocks, you get two, wherever they are.
Soft IP	Built out of fabric: LUTs, flip-flops, BRAM, routing. As many instances as will fit, anywhere, in any configuration — but it consumes the resources and the timing budget you would otherwise spend on your own logic.
Reusable IP	A parameterised, verified design block you drop into a project — a UART, a FIFO, an AXI interconnect, an FFT. Almost always soft, delivered as RTL or as an encrypted netlist.

Function	Usually hard	Usually soft
UART, SPI, I2C	—	soft: tiny, and every application wants it slightly different
FIFO	some families harden FIFO control around a BRAM	soft control logic over a BRAM
Ethernet MAC	on many devices	soft is common too, and more flexible
PCIe	**hard** — the protocol stack and the analog front end are far too costly in fabric	rarely
DDR controller	**hard** — timing-critical and calibration-heavy	rarely
Processor	hard on SoC FPGAs	soft cores are common on any device
Multiply-accumulate	**hard** — the DSP block	soft for small or constant widths

// §23 SOC FPGA



23.1 Partitioning — the actual engineering decision MUST

Put it in software	Control flow, configuration, protocol state machines above the wire level, error handling, user interfaces, anything irregular, anything that changes often.
Put it in the fabric	Wide regular arithmetic, bit-level manipulation, hard real-time deadlines, custom or fast interfaces, anything that must happen every cycle without fail.
Watch the boundary	The interconnect between the two is where designs bog down. Moving one word costs far more than computing on it. Batch, stream, and use DMA rather than having the processor poke registers in a loop.

23.2 Interconnect, conceptually HIGH

Whatever the bus is called, it carries three things: an **address** (what), **data** (the value), and **control** (direction, size, validity, completion). AXI, the family most FPGA users meet, comes in three flavours worth telling apart:

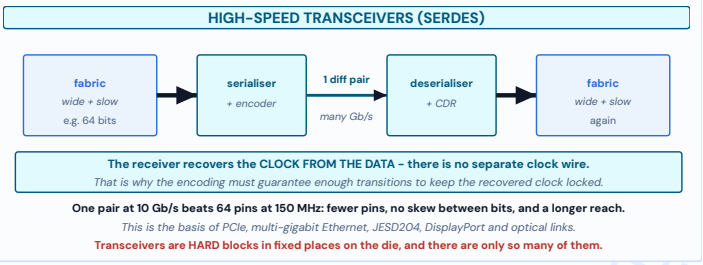
AXI-Lite	Simple memory-mapped register access. One transfer at a time. Use it for control and status registers — it is easy to get right and fast enough for configuration.
AXI (full)	Memory-mapped with bursts, multiple outstanding transactions and independent read and write channels. Use it for DMA into and out of DDR.
AXI-Stream	No address at all: just data, `tvalid`, `tready` and a couple of side signals. The natural shape for a datapath — a filter, a codec, a packet processor.

The one idea to take from any of these buses: `valid` / `ready` handshaking. The source asserts valid when it has data; the sink asserts ready when it can take it; a transfer happens on the cycle both are high. That single convention composes safely, allows either side to stall, and is why streaming IP from different sources can be chained together at all.

23.3 Soft processors GOOD

A processor built entirely from fabric. Slower and larger than a hard core, but you can have several, place them where you like, and pick a device with no hard processor at all. They earn their place for housekeeping — initialising peripherals, running a slow control loop, decoding a command protocol — where the alternative is a large and fiddly state machine. RISC-V is now a common choice for new designs, alongside the vendors' own soft cores.

// §24 HIGH-SPEED TRANSCEIVERS



Serialiser	Takes a wide, slow word from the fabric and clocks it out one bit at a time at many gigabits per second.
Deserialiser	The reverse, at the far end.
Clock and data recovery	There is **no separate clock wire**. The receiver extracts the clock from the data transitions themselves.
Line encoding	8b/10b, 64b/66b and similar schemes guarantee enough transitions to keep CDR locked, and keep the average DC level balanced.
Equalisation	Pre-emphasis at the transmitter and equalisation at the receiver compensate for the frequency-dependent loss of the channel.

Why serial beats parallel above a certain speed. A parallel bus needs every bit to arrive within a fraction of a bit period, which becomes impossible as the period shrinks — you are fighting trace-length skew across a connector. One differential pair has no inter-bit skew to fight, uses two pins instead of sixty-four, and travels much further. This is why PCIe, multi-gigabit Ethernet, SATA, USB 3, DisplayPort and optical links are all serial.

Transceivers are hard blocks in fixed places. They come in quads, they are wired to specific pins, they need clean reference clocks and careful board design, and there are only so many. A design that needs eight lanes on a device with four is not a routing problem — it is the wrong device.

// §25 MODERN HETEROGENEOUS DEVICES

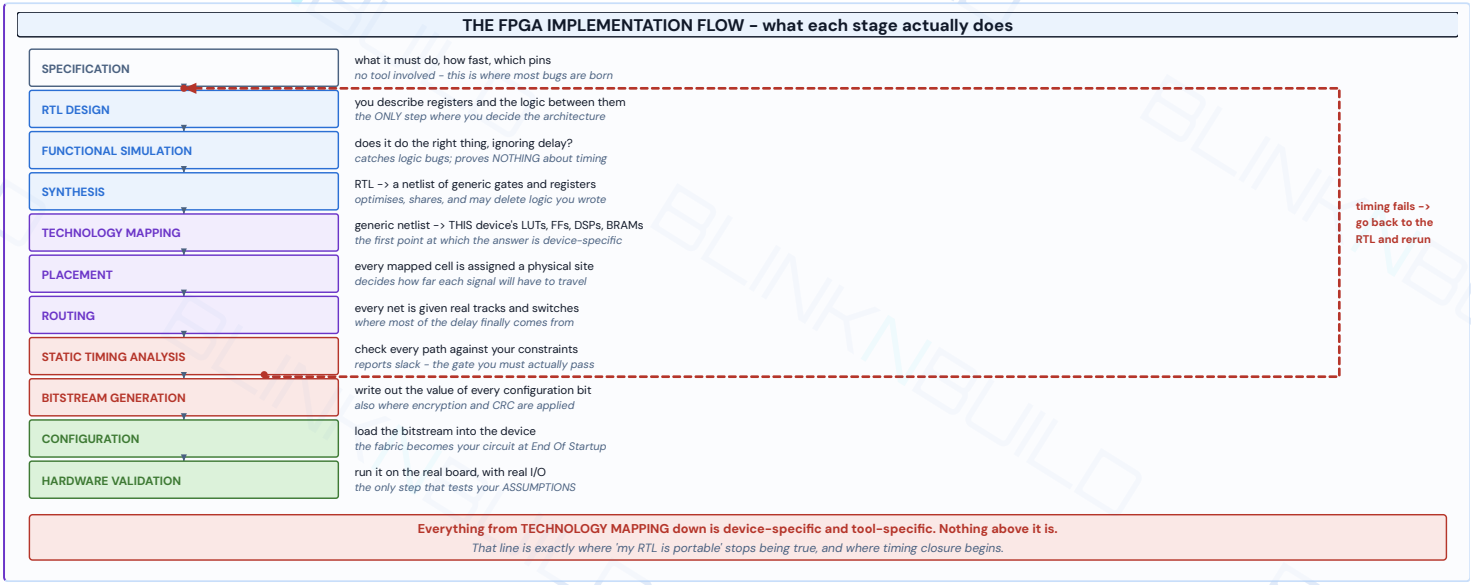
High-level orientation only, and emphatically **not** a description of every FPGA.

General fabric	Still LUTs, flip-flops, carry and routing. Everything else is built around it.
Hardened processors	Application-class and real-time cores, with their own memory system.
AI / vector engines	Arrays of small programmable processors optimised for dense low-precision arithmetic, on some recent families.

Network-on-chip	A hardened packet-switched interconnect replacing the fabric routing for bulk data movement between blocks and memory.
HBM	Stacked DRAM on the same package, offering memory bandwidth an ordinary DDR interface cannot approach.
Registers in the routing	Some architectures place bypassable registers on interconnect segments so the tools can retime aggressively — `INTEL`'s HyperFlex is the well-known example.

Do not assume a resource exists. The devices most students actually use — the parts on affordable boards — usually have fabric, BRAM, DSP, clock managers, I/O, and perhaps a hard processor and a few transceivers. AI engines, NoCs and HBM belong to specific high-end families. Read the family's own documentation before designing around anything on this list.

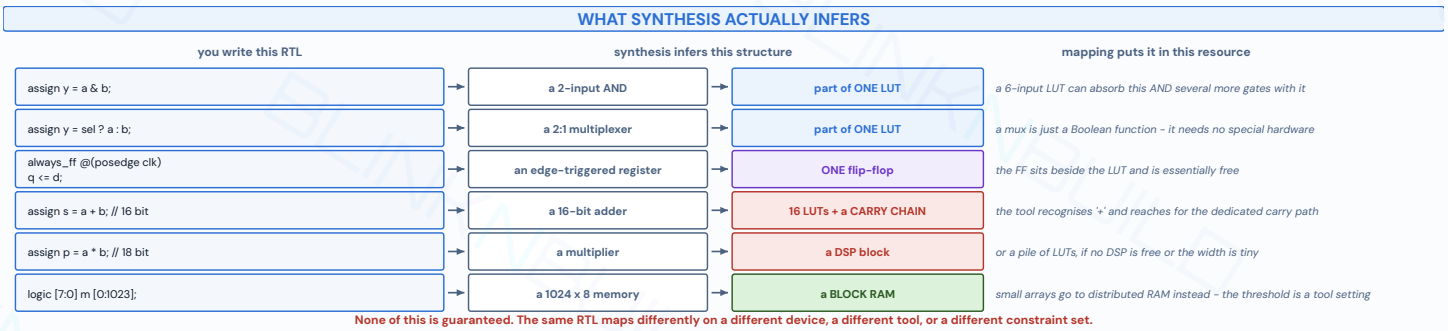
// §26 THE DESIGN FLOW, END TO END



The line that matters. Everything from technology mapping downwards is specific to **this device and this tool**. Everything above it is not. That is exactly where "my RTL is portable" stops being true, and it is where timing closure begins.

// §27 SYNTHESIS — WHAT ACTUALLY HAPPENS

27.1 RTL to inferred structure to resource MUST



27.2 Worked examples MUST

```
assign y = a & b;
```

HDL → Boolean logic → LUT. A 2-input AND. On a LUT6 fabric this uses a small corner of one LUT, and the tool will almost certainly pack several more gates into the same LUT alongside it. You did not get "an AND gate"; you got part of a truth table.

```
assign y = sel ? a : b;
```

HDL → MUX → LUT. A multiplexer is a Boolean function like any other: `y = (sel & a) | (~sel & b)`. It needs no special hardware in the fabric — although wide muxes do use the dedicated wide-function muxes described in §6.5.

```
always_ff @(posedge clk)
    q <= d;
```

HDL → flip-flop. One storage element, taken from the pair that sits beside every LUT. Essentially free.

```
always_ff @(posedge clk)
    if (en) q <= d;
```

HDL → flip-flop with its clock enable used. The `en` lands on the flip-flop's dedicated CE pin. No extra LUT, no feedback mux. This is why a clock enable is the right way to hold a value.

27.3 What synthesis is allowed to do to your code MUST

Constant propagation	A signal tied to a constant is folded through everything it feeds. Whole subsystems can vanish because one parameter was set to zero.
Dead logic elimination	Logic whose output reaches no register and no pin is **deleted**. It does not appear in the netlist and you cannot probe it.
Boolean optimisation	The expression is restructured freely, as long as the function is preserved. Do not expect your gate structure to survive.
Resource sharing	Two multipliers in mutually exclusive branches may become one multiplier and a mux — a large area saving, and sometimes a timing cost.
Register duplication	A heavily loaded register may be replicated so each copy drives fewer loads (§36).
Retiming	Registers may be moved across combinational logic to balance stage delays, preserving cycle-by-cycle behaviour at the boundaries.
FSM re-encoding	Your state constants may be replaced with a one-hot or Gray encoding the tool prefers.
Inference	Recognising that a pattern is an adder, a multiplier, a memory, a shift register, or a counter, and reaching for the matching hard resource.

Why "my signal disappeared" is usually not a bug. If a debug signal has no path to a pin or a register, synthesis removes it — correctly. Mark it with the vendor's keep or preserve attribute, or drive something with it. Similarly, an unconnected module output, an unused generate branch and an unreachable FSM state all legitimately vanish.

// §28 TECHNOLOGY MAPPING

28.1 The mapping table MUST

RTL concept	Typical FPGA resource	Notes
Boolean expression	**LUT**	packed several gates per LUT where the inputs allow
Multiplexer	**LUT**, or dedicated wide muxes	a wide mux may span several LUTs plus the F7/F8-style muxes
Register / `always_ff`	**flip-flop**	one per bit; CE and reset use dedicated pins
Adder, subtractor, comparator, counter	**LUT + carry chain**	the operator is recognised; a hand-built one may not be

RTL concept	Typical FPGA resource	Notes
Multiplier	**DSP block**, or LUTs + carry	depends on width, DSP availability and constant-ness
Multiply-accumulate	**one DSP block**	if written in the shape the tool recognises
Large array, synchronous read	**block RAM**	read must be registered, or it cannot be a BRAM
Small array, asynchronous read	**distributed RAM**	uses LUTs; fine when it is genuinely small
Shift register / delay line	**SRL**, BRAM, or flip-flops	an SRL is dramatically cheaper than N flip-flops, where available
Clock division / phase shift	**PLL / MMCM / DLL**	never a fabric counter, if the result is used as a clock
Tri-state inside the design	**a multiplexer**	internal tri-state buses do not exist in modern fabrics; only I/O buffers are genuinely tri-state
ROM / constant table	**BRAM initial contents**, or LUTs	a BRAM configured with initial data and never written

Every row here is "typically". The actual mapping depends on the architecture, the synthesis tool and its version, your coding style, your constraints, and how full the device already is. This table tells you what to **expect**; the utilisation report tells you what you **got**. When they disagree, the report is right and you should find out why.

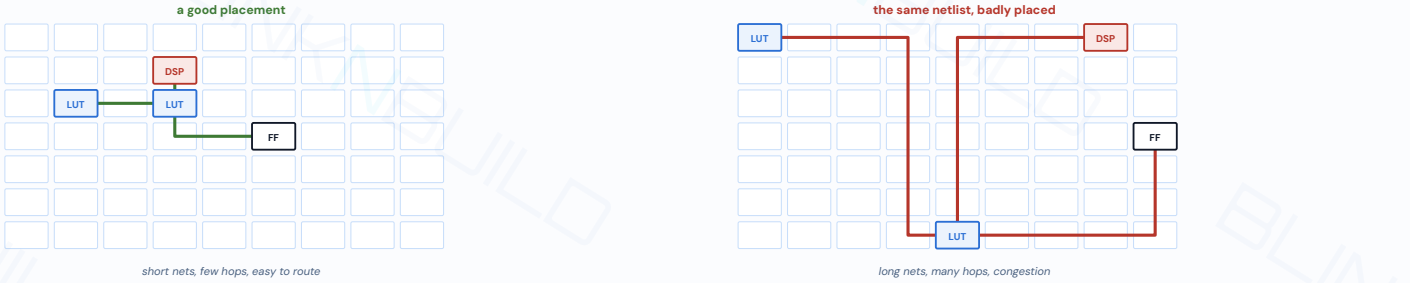
28.2 Getting the mapping you wanted HIGH

Write the operator	Use `+`, `*`, `<`. Inference works on recognisable shapes; a hand-built ripple adder often loses the carry chain.
Register memory reads	A synchronous read is the price of admission to block RAM.
Register DSP inputs and outputs	Otherwise the pipeline registers are bypassed and the block runs slowly.
Use attributes when you mean it	`ram_style`, `use_dsp`, `keep`, `max_fanout` and their equivalents exist for the cases where the tool's default is wrong for you.
Read the report	Every tool prints what it inferred. If you expected a BRAM and got 4,000 LUTs, that message was in the log.

// §29 PLACEMENT AND ROUTING

29.1 The two physical decisions MUST

PLACEMENT DECIDES WHERE. ROUTING DECIDES HOW IT GETS THERE.



Placement is chosen to MINIMISE the routing that will be needed - which is why the placer is timing-driven, not just area-driven.

The router then has to live with that choice. A bad placement cannot be rescued by a good router - which is why timing problems are usually placement problems.

29.2 Placement MUST

What it decides	Which physical site each mapped cell occupies — this LUT in that slice, this flip-flop in that site, this memory in that BRAM column.
What constrains it	Resource type and availability; carry chains that must stay vertical and contiguous; DSP cascades that must stay in one column; I/O that must reach its assigned pin; clock regions.
What it optimises	Estimated routing delay on the critical paths first, then congestion, then total wirelength. Modern placers are **timing-driven**, so your constraints change the answer.
Why it is hard	It is a huge combinatorial optimisation, solved heuristically. That is why results vary between runs, seeds and tool versions.

29.3 Routing MUST

What it decides	Which tracks and which switches carry each net, given where the placer put everything.
What constrains it	The switches that physically exist, the number of tracks in each channel, and every other net competing for them.
What goes wrong	**Congestion**: a region runs out of tracks and nets detour, adding hops and delay to paths that were fine on the estimate.
The hard truth	The router cannot rescue a bad placement. If two connected blocks are at opposite corners, no routing choice makes that path short.

So timing problems are usually placement problems, and placement problems are usually design problems. Deep logic, huge fanout nets, a design with no register boundaries, or logic that must span the die because of where its I/O is — those force the placer into bad choices, and the router then inherits them. Fixing timing at the routing stage almost never works; fixing it in the RTL almost always does.

// §30 UTILISATION — AND WHY IT MISLEADS

30.1 What gets counted HIGH

LUT	Usually reported both as LUTs used and as slices/blocks **occupied** — a slice with one LUT in use is fully occupied for placement purposes.
FF	Nearly always lower than LUT usage. Spare flip-flops are the resource you should be spending.
BRAM	Counted in whole blocks. A 100-bit memory that lands in a 36 Kb block consumes the whole block.
DSP	Whole blocks. A wide multiply may take several.
I/O	Often the first thing to run out on small devices, and it is fixed by the package.
Clocking	Global buffers and clock managers. Scarce, and easy to overlook until the tools start putting clocks on ordinary routing.
Routing	Rarely a single headline number, but congestion maps show it. This is the one that actually kills designs.

30.2 The insight MUST

[TRAP] Resource percentage does not predict performance. A design at 30% LUT utilisation can fail timing badly, and a design at 85% can close comfortably. What matters is **logic depth**, **fanout** and **locality** — how many LUTs a signal passes through between registers, how many loads each net drives, and how far apart the connected pieces ended up. Utilisation only tells you whether it fits.

That said, very high utilisation does hurt — indirectly. Above roughly 80–90% the placer runs out of good sites and the router runs out of tracks, so both start making compromises. The timing degradation is real, but the cause is congestion and lost placement freedom, not the percentage itself.

30.3 Reading a utilisation report properly HIGH

Compare LUT and FF counts	Far more LUTs than FFs suggests deep combinational logic — a pipelining opportunity.
Check LUTs used as memory	A large number here means distributed RAM you may not have intended. Should it be a BRAM?
Check DSP count against expectation	More than you expected means a multiply you forgot about. Fewer means one was built in LUTs instead.
Check the control-set report	Too many distinct clock/reset/enable combinations prevents flip-flops from packing together and inflates slice usage. Standardising your reset and enable style can free 20% of the fabric.
Check clock buffer usage	If it is near the limit, find out which clocks are real.

// §31 STATIC TIMING ANALYSIS

31.1 What STA is MUST

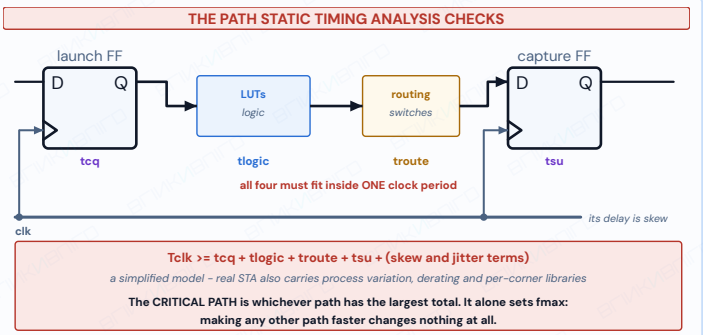
`SIMPLE` The tool walks every path from every register to every register and asks: does the data get there in time?

`ENGINEERING` An exhaustive, vector-free analysis. It does not simulate; it sums delays along every timing path and compares arrival against requirement, over process, voltage and temperature corners.

`PHYSICAL INTUITION` Every LUT, every routing switch, every buffer has a characterised delay. STA adds them up along real, post-route paths. This is why timing numbers before routing are only estimates.

Vector-free is the whole point. Simulation only tests the stimulus you thought of. STA checks **every** path, whether or not your testbench ever exercises it. It is the only tool that can tell you the design will work at frequency — and passing simulation tells you nothing about it.

31.2 The path MUST

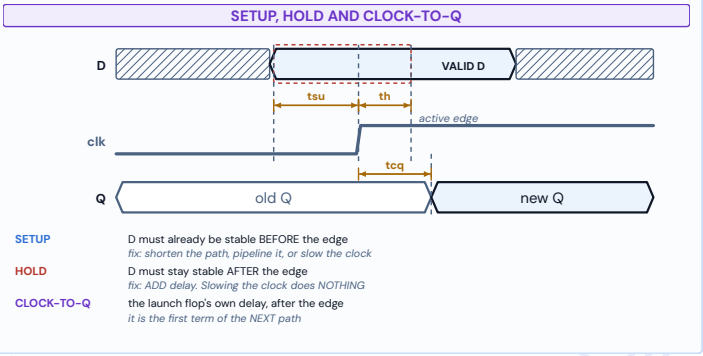


31.3 The four kinds of path HIGH

Register to register	Entirely internal. The one you control completely, and the one that usually sets fmax.
Input to register	From a pin to the first flip-flop. Needs an **input delay** constraint describing what the outside world does, or the tool has no idea when data arrives.
Register to output	From the last flip-flop to a pin. Needs an **output delay** constraint describing what the outside world requires.
Input to output	Purely combinational through the device. Rare, and needs its own constraint.

Unconstrained paths are not analysed. They are silently reported as unconstrained and then ignored. A design can report "all timing met" while half of it was never checked. Always look at the count of unconstrained endpoints before believing a clean report.

// §32 SETUP, HOLD AND CLOCK-TO-Q



32.1 The two checks MUST

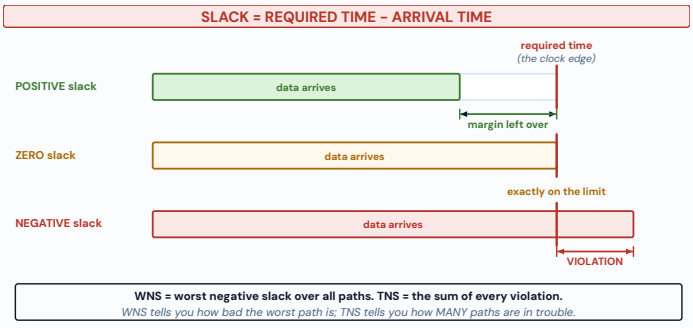
	SETUP	HOLD
The question	Is the data there **early enough**?	Does the data stay put **long enough**?
Cause of failure	the path is too **slow**	the path is too **fast**
Uses which delay	the **longest** path	the **shortest** path
Depends on the clock period?	**Yes**	**No** — the period is not in the equation
Fixed by slowing the clock?	**Yes, always**	**Never**
Real fixes	pipeline, reduce logic depth, reduce fanout, improve placement, use dedicated resources	**add delay** on the data path, or reduce clock skew
Severity on an FPGA	recoverable — run slower	serious, but the tools normally fix it for you by inserting delay during routing

[FORMULA] The simplified conceptual model.
SETUP: `Tclk >= tcq + tlogic + troute + tsu + tuncertainty - tskew`
HOLD: `tcq + tlogic(min) + troute(min) >= th + tskew`

`fmax = 1 / Tclk(min)`. Convention here: `tskew` is the capture clock's arrival minus the launch clock's, so positive skew means the capture edge is **later**.

This is a teaching model. Real FPGA STA additionally carries multiple PVT corners, on-chip variation, clock-network pessimism removal, per-cell derating and vendor-specific timing arcs. Use it to reason; use the tool's report to decide.

// §33 SLACK AND THE CRITICAL PATH



33.1 Slack MUST

Definition	`slack = required time - arrival time.`
Positive	The requirement is met with margin to spare.
Zero	Exactly on the boundary. It passes, but there is nothing left for variation.
Negative	A **violation**. The design does not meet timing at this frequency.
WNS	Worst negative slack — how bad the worst single path is.
TNS	Total negative slack — the sum over all violating paths. Tells you how **many** paths are in trouble.

Read both numbers. WNS of -0.2 ns with TNS of -0.2 ns is one path: find it and fix it. WNS of -0.2 ns with TNS of -400 ns is thousands of paths: the clock is simply too fast for this architecture, and no local fix will help.

33.2 The critical path MUST

Definition	The path with the worst slack. It alone determines the maximum clock frequency.
The consequence	Speeding up any **other** path changes nothing at all. Optimisation effort spent away from the critical path is wasted effort.
It moves	Fix the worst path and a different one becomes critical. Timing closure is a sequence of critical paths, not one.
What it is made of	Look at the report's breakdown. If it is mostly **logic**, pipeline it. If it is mostly **routing**, the problem is placement, fanout or distance — and more pipelining may not help.

33.3 A worked example HIGH

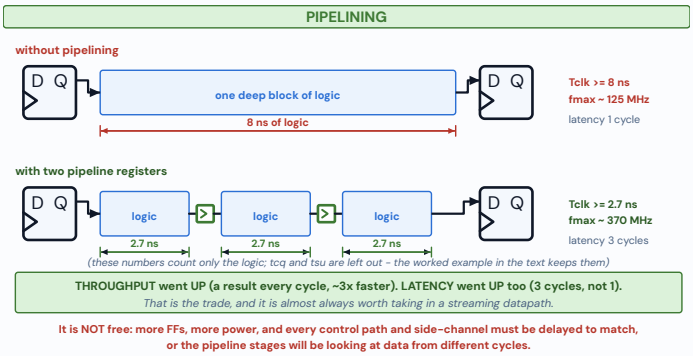
```
tcq    = 0.35 ns    launch flip-flop
tlogic= 3.10 ns    four LUTs in series
troute= 4.20 ns    the hops between them
tsu    = 0.25 ns    capture flip-flop
unc    = 0.10 ns    jitter and uncertainty
                    ------
Tclk(min) = 8.00 ns   ->  fmax = 125 MHz

Split it into two pipeline stages of roughly half:
tcq 0.35 + 1.55 + 2.10 + 0.25 + 0.10 = 4.35 ns -> 230 MHz

Note WHERE the time went: routing (4.20) exceeded logic (3.10).
Halving the logic alone would NOT have halved the period.
```

The lesson in that example. Routing was the larger term, which is typical. Pipelining helped because it split the routing too — the two shorter paths span less distance. If you pipeline without shortening the physical span, you get much less than you expected.

// §34 PIPELINING



34.1 The trade, stated precisely MUST

Throughput	**Up.** After the pipeline fills, one result leaves every cycle, at the higher clock rate.
Latency	**Up.** Any single input now takes more cycles to reach the output.
fmax	**Up.** The longest register-to-register path is shorter.

Area	Up a little — more flip-flops, which are usually the resource you have spare.
Power	Up: more registers toggling, and a faster clock. Down per unit of work, if the throughput gain outweighs it.

The distinction interviewers probe. Pipelining does **not** make one operation faster — that operation now takes *more* wall-clock time end to end. It makes the **rate** of operations higher. For a streaming datapath that is everything; for a tight feedback loop where the next input depends on the last output, it can be useless or actively harmful.

34.2 Where pipelining does not help MUST

Feedback loops	An accumulator, a CRC, an IIR filter: the next value needs this cycle's result. You cannot register your way out of it — you have to restructure the maths (unrolling, look-ahead, C-slowing).
Routing-dominated paths	If the path is long because the two ends are far apart, adding a register in the middle helps only if the register can be **placed** in the middle.
Paths through hard blocks	A BRAM or DSP block has its own registers. Use those before adding fabric ones around it.
Already-shallow paths	If the path is `tcq + one LUT + tsu`, there is nothing left to split.

The bookkeeping cost is where bugs come from. Add a stage to the datapath and **every** signal that travels alongside it — valid flags, tags, addresses, control decisions — must be delayed by the same number of cycles. Miss one and your pipeline stages are looking at data from different transactions. This is far more often the source of pipelining bugs than the arithmetic itself.

// §35 PARALLELISM AND RTL

35.1 Three kinds of parallelism HIGH

Spatial	N copies of the same hardware, each working on different data. Costs N times the resources; gives N times the throughput.
Pipeline	One copy split into stages, all busy at once on different data. Costs registers; gives a higher clock rate.
Concurrent datapaths	Different functions running simultaneously because they are different hardware. This is free — it is simply what the fabric does.

The mental shift from software. In software, two functions run one after the other unless you deliberately parallelise. In RTL, two `always_ff` blocks are two separate pieces of silicon and they both run **every cycle**, whether you want them to or not. Sequencing is something you must **build** — with a state machine, a counter, a valid flag — not something you get for free.

35.2 Register-transfer level MUST

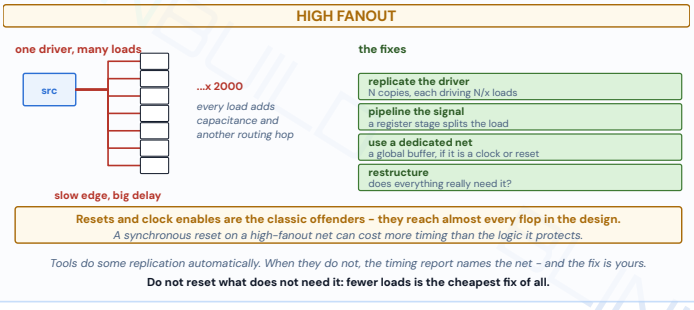
RTL is a description in exactly the terms the hardware provides: **registers**, and the **combinational logic between them**. Every clock edge, every register simultaneously captures whatever its logic is presenting.

Datapath	Where the data flows and is transformed. Wide, regular, mostly LUTs, carry chains, DSPs and BRAMs.
Control path	What decides when things happen. Narrow, irregular, mostly a state machine and some counters.
Register boundaries	Your architectural choice, and the main lever you have over timing. Where you put them decides your critical paths.

```
// this is the shape of essentially all synthesisable RTL:
always_ff @(posedge clk) begin
    if (!rst_n)  state_q <= IDLE;       // sequential: what to keep
    else         state_q <= state_d;
end

always_comb begin                       // combinational: what to do
    state_d = state_q;
    case (state_q)
        IDLE:  if (start) state_d = RUN;
        RUN:   if (done)  state_d = IDLE;
        default:          state_d = IDLE;
    endcase
end
```

// §36 HIGH FANOUT



What it is	One driver feeding many loads. "Many" starts to matter in the hundreds and becomes serious in the thousands.
Why it hurts	Every load adds capacitance to the driving buffer and, usually, another routing hop to reach it. The net becomes slow, and its delay becomes hard for the placer to control because the loads are scattered.
The usual offenders	Resets, clock enables, "global" mode and configuration bits, and any signal that feeds a wide datapath.
Fix: replicate	Make N copies of the driving register, each serving a fraction of the loads. Costs flip-flops, which you have.
Fix: pipeline	Insert a register stage so the fanout is split across cycles. Costs a cycle of latency on that control signal — check that it still lines up.
Fix: reduce the load	Do not reset flip-flops that do not need it. Do not distribute a signal that only three places actually use.
Fix: use dedicated resources	If it genuinely must reach everything and is timing-critical, a global buffer network exists for exactly that.

// §37 POWER

37.1 The two components HIGH

[FORMULA] Dynamic power: `P = alpha * C * V^2 * f`
`alpha` = switching activity (how often a node toggles), `C` = capacitance being driven, `V` = supply voltage, `f` = frequency. Voltage is squared, so it dominates — but the core voltage is set by the device, not by you (a few families offer a low-power core-voltage option). What you **do** choose is activity, capacitance and frequency.

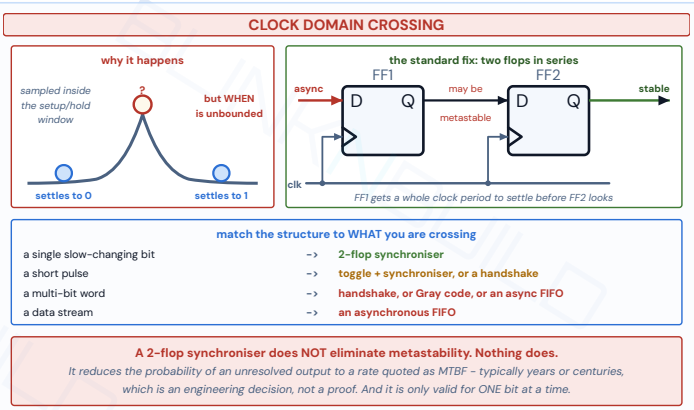
Static / leakage	Drawn whenever the device is powered, whether or not anything is switching. It scales with die size, process and temperature — which is why a large FPGA idles warm, and why non-volatile fabrics advertise low static power.
Dynamic	Everything that toggles. On an FPGA the biggest contributors are usually the **clock networks** (they toggle every cycle into everything), then the **routing** (long, capacitive), then logic, memory, DSP and I/O.

37.2 What actually reduces it HIGH

Use clock enables	Stops registers toggling without creating a gated clock. The single most effective fabric-level technique.
Turn off memory	Deassert BRAM enables when you are not accessing them — memory arrays are power-hungry when enabled.
Reduce data movement	Every long net you avoid is capacitance you do not charge. Keep related logic together.
Use hard blocks	A DSP multiply uses far less energy than the same multiply built from LUTs and carry.
Slow the clock where you can	Not everything needs the fast domain. Frequency is a linear term.
Mind the I/O	Drive strength, slew rate and termination are real power. Do not use fast, strong I/O where slow and weak would do.
Do not over-reset	A global reset network is enormous fanout that toggles at startup and holds capacitance for ever after.

// §38 CLOCK DOMAIN CROSSING AND METASTABILITY

38.1 Metastability MUST



SIMPLE If data changes at exactly the wrong moment relative to the clock edge, the flip-flop cannot decide, and its output sits between valid levels for an unpredictable time.

ENGINEERING A flip-flop is a bistable feedback loop. Violating its setup/hold aperture can leave it balanced at the metastable point. It will resolve — but the resolution time is a random variable with an unbounded tail.

PHYSICAL INTUITION A ball balanced on a hilltop. It will roll off; when, and which way, is not something you can bound with certainty.

When it happens	Whenever you sample a signal that is asynchronous to your clock — a button, another clock domain, a recovered clock, an off-chip interface.

Why it is dangerous	Different loads on the metastable output can resolve it **differently**. Half your state machine sees a 1 and half sees a 0, and your one-hot state vector now has two bits set.
What does not fix it	Faster flip-flops, buffers, careful routing, or a longer clock period. You cannot prevent it when sampling a genuinely asynchronous signal.
What does help	Giving the flip-flop **time to settle** before anything looks at it — which is exactly what a synchroniser does.

38.2 The two-flop synchroniser MUST

Two flip-flops in series in the destination domain. The first may go metastable; it then has a full clock period to resolve before the second samples it. Downstream logic only ever sees the second flop's output.

[TRAP] A 2-flop synchroniser does NOT eliminate metastability. Nothing does. It reduces the probability of an unresolved output reaching your logic to a rate expressed as **MTBF** — typically years or centuries, depending on the clock frequencies, the data rate and the flip-flop's resolution time constant. That is an engineering judgement about acceptable risk, not a proof of correctness. Add a third stage where the consequences of failure are severe, or the clocks are very fast.

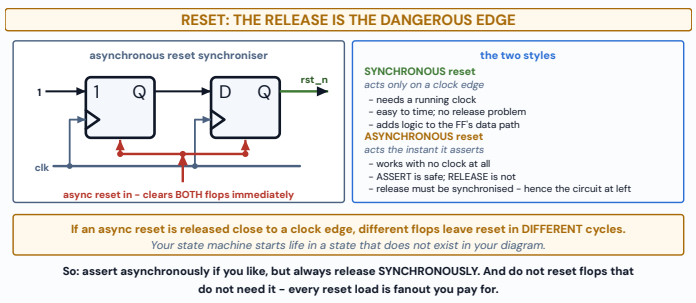
And it is only valid for ONE BIT. Synchronise a multi-bit bus with parallel 2-flop synchronisers and each bit resolves independently, so you can sample a value that never existed — the classic `0111 -> 1000` read as `1111`. This is the single most common CDC bug.

38.3 Match the structure to what you are crossing MUST

What you are crossing	Correct structure	Why
A single slow-changing bit (a status flag, a mode)	**2-flop synchroniser**	one bit, and the value is stable long enough to be sampled
A short pulse, possibly narrower than the destination clock	**toggle** in the source, synchronise, edge-detect in the destination — or a full handshake	a pulse can be missed entirely; a level change cannot
A multi-bit value that changes occasionally	**handshake**: hold the data stable, synchronise a request, wait for an acknowledge	the data is guaranteed stable while it is being sampled
A multi-bit counter or pointer	**Gray code** plus synchronisers	only one bit changes per step, so a mid-change sample is still a valid value
A continuous data stream	**asynchronous FIFO** (§20.2)	the standard, verified answer; do not invent your own
Reset release	**reset synchroniser** (§39)	assert whenever; release in step with the destination clock

Run the CDC report. Every serious toolchain has one. It finds crossings you did not know you had — including the ones created accidentally by a LUT-generated clock or by a signal that quietly travels between two supposedly related domains. Treat its warnings as findings, not noise.

// §39 RESET ARCHITECTURE



39.1 Synchronous vs asynchronous MUST

Synchronous reset	Asynchronous reset
Takes effect only on a clock edge.	Takes effect the instant it asserts.
Needs a running, stable clock — useless before the clock manager locks.	Works with no clock at all.
Easy to time; it is just another synchronous signal.	Assertion is safe; **release** is a timing hazard.
Adds logic to the flip-flop's data path on some fabrics.	Uses the flip-flop's dedicated asynchronous pin.
Filters glitches — a short spike between edges is ignored.	A glitch resets the design.

39.2 The release problem MUST

[TRAP] It is the RELEASE of an asynchronous reset that breaks designs. If reset de-asserts close to a clock edge, some flip-flops leave reset on this cycle and others on the next. Your state machine begins life in a state that does not appear in your diagram, and your counters start from mixed values. The standard fix is a **reset synchroniser**: assert the reset asynchronously (so it works with no clock), but release it through two flip-flops clocked by the destination clock, so every flop in that domain leaves reset on the same edge.

39.3 Practical reset rules HIGH

One reset style per clock domain	Mixing synchronous and asynchronous resets inside one block prevents flip-flops packing together and inflates area.
One reset synchroniser per clock domain	Each domain needs its release aligned to **its own** clock.
Do not reset what does not need it	Datapath pipeline registers usually do not need a reset — they flush naturally. Every reset load is fanout you pay for, for ever (§36).
Do reset control logic	State machines, counters that gate behaviour, and anything whose initial value affects correctness.
Remember configuration already cleared everything	On an SRAM FPGA, GWE releases flip-flops into their configured initial state (§12.3). A reset is for **re**-initialising at run time, not for reaching a known state at power-up.

The interview answer. "Assert asynchronously so it works before the clock is stable; release synchronously so every flop leaves reset on the same edge; use one reset synchroniser per domain; and reset only what actually needs it."

// §40 TIMING CONSTRAINTS

40.1 What you must tell the tools MUST

Clock definition	Period and duty cycle for every clock entering the device. Without this the tools have **no timing goal at all** and will report success on a design that cannot run.
Generated clocks	Any clock derived on-chip — a PLL output, a divided clock — declared in terms of its source.
Input delay	When external data arrives relative to the clock at the pin. Describes the **outside world**, which the tool cannot see.
Output delay	What the outside world requires of your output relative to the clock.
Clock uncertainty	Jitter and margin you want subtracted from the available period.
Pin assignment	Which signal goes to which package pin, and its I/O standard.
Clock groups	Which clocks are genuinely asynchronous to each other, so the tool stops trying to time paths between them.

40.2 Exceptions — powerful and dangerous MUST

False path	"This path can never matter." Correct for a truly static configuration signal or a genuinely unrelated domain.
Multicycle path	"This path is allowed N cycles, not one." Correct when the design genuinely does not sample the result until N cycles later.
Max/min delay	A direct bound, usually for asynchronous interfaces and CDC nets.

[TRAP] Never use a false path to silence a timing failure. A false path does not make the path fast — it makes the tool **stop checking** it. If the path is real, you have converted a build failure into a field failure. The same applies to declaring a multicycle path on something that is actually sampled every cycle. Constraints are a **description of design intent**; if the description is a lie, everything downstream is worthless.

Under-constraining is the commonest real-world mistake. A design with only a clock constraint and no I/O delays will report timing met while every path to and from the pins goes unanalysed. Always check the report of **unconstrained endpoints** — a clean timing summary next to a thousand unconstrained paths means nothing.

// §41 SIMULATION VS SYNTHESIS VS HARDWARE

Stage	What it does	What it proves	What it cannot tell you
RTL simulation	executes your HDL as a program, with a delta-cycle model of time	the **logic** is right for the stimulus you wrote	anything about delay, timing closure, resources, or inputs you did not think of
Synthesis	turns RTL into a netlist of real resources	the design is **synthesisable** and roughly what size	whether it will meet timing, or fit after placement
Implementation	maps, places and routes	it **fits**, and what the real delays are	whether your assumptions about the outside world were right
Static timing analysis	checks every constrained path	it will run at frequency, **if the constraints are honest**	that your constraints described reality

Stage	What it does	What it proves	What it cannot tell you
Hardware	runs the real thing on a real board	the whole system, including everything you assumed	little — but it is the hardest place to see inside

Passing simulation is not passing. Simulation tests the logic you thought of, with no delay. Timing analysis tests the delays, with no notion of whether the logic is right. Hardware tests your assumptions. All three are necessary and none substitutes for another.

41.1 Why simulation and hardware disagree MUST

Timing violations	Simulation at RTL has no delays, so a setup violation simply does not exist there.
Uninitialised state	Simulation shows `X`; real flip-flops come up at their configured value, often 0 — so a bug that shows as `X` in simulation can look "fine" in hardware, and vice versa.
Metastability	Cannot occur in an RTL simulator. It is invisible until hardware, and then intermittent.
Synthesis transformations	Logic removed, shared or retimed changes what is observable, and occasionally exposes a latent bug.
Inferred latches	A latch inferred from an incomplete `case` or `if` may simulate acceptably and behave very differently once it is real transparent hardware on a routed net.
The testbench is wrong	Frequently. The real interface does not behave the way your model of it does.
Race conditions in the HDL	Blocking assignments in sequential blocks, or reading and writing the same signal in two blocks, give simulator-dependent results.

// §42 SYNTHESISABILITY

Construct	Synthesisable?	Note
`if` / `else`	**yes**	becomes a multiplexer. Incomplete `if` in combinational logic infers a **latch**.
`case`	**yes**	always include `default`, for the same reason.
`for` loop	**yes, if bounds are static**	it is **unrolled** into parallel hardware — it is not a loop at run time.
`while` / `forever`	**no**	no static bound, so no fixed hardware.
`always_comb` / `always @(*)`	**yes**	combinational. Assign every output on every path.
`always_ff @(posedge clk)`	**yes**	flip-flops. Use non-blocking (`<=`) assignments.
`#10` delays	**no**	ignored by synthesis, so simulation and hardware diverge silently.
`initial`	partly	ignored by many tools; some accept it for RAM/register initial values on FPGAs. Do not rely on it for logic.
`$display`, `$finish`, `$random`	**no**	simulation only.
File I/O (`$readmemh`)	partly	widely accepted for initialising memory contents at synthesis time.
`real`, dynamic arrays, classes, `mailbox`	**no**	verification constructs.
Assertions	not as logic	hugely valuable in simulation; some tools can synthesise a subset for on-chip checking.
Multiple drivers on one signal	**no**	an error in hardware, whatever the simulator does.

The general rule. If you cannot draw the hardware, the tool cannot build it. And "the tool accepted it" is not the same as "the tool built what you meant" — always read the inference messages.

// §43 DEBUGGING AN FPGA

RTL simulation	First and cheapest. Full visibility, no timing. Where logic bugs should die.
Assertions	Encode your assumptions as checks. They fire the moment an assumption breaks, which is far earlier than the symptom.
Synthesis and implementation logs	Inference messages, removed logic, inferred latches, unconnected ports. Most "mysterious" behaviour was announced here in a warning.
Timing report	The critical path, its slack, and the logic/routing split. Read the actual path, not just the summary.
Utilisation report	Did you get the resources you expected? Unexpected LUT-as-memory or missing DSPs point straight at an inference problem.
CDC report	Finds unsafe crossings automatically, including ones you created by accident.
On-chip logic analyser	An instrumented core (ILA-style) captures internal signals into BRAM and reads them back over JTAG. The most powerful hardware debug tool you have.
Route a signal to a pin	Crude, fast, and sometimes the only way to see something with an oscilloscope at full speed.
An LED or a counter	Never underestimate a heartbeat LED and an error counter. They answer "is it alive and is it unhappy?" in seconds.

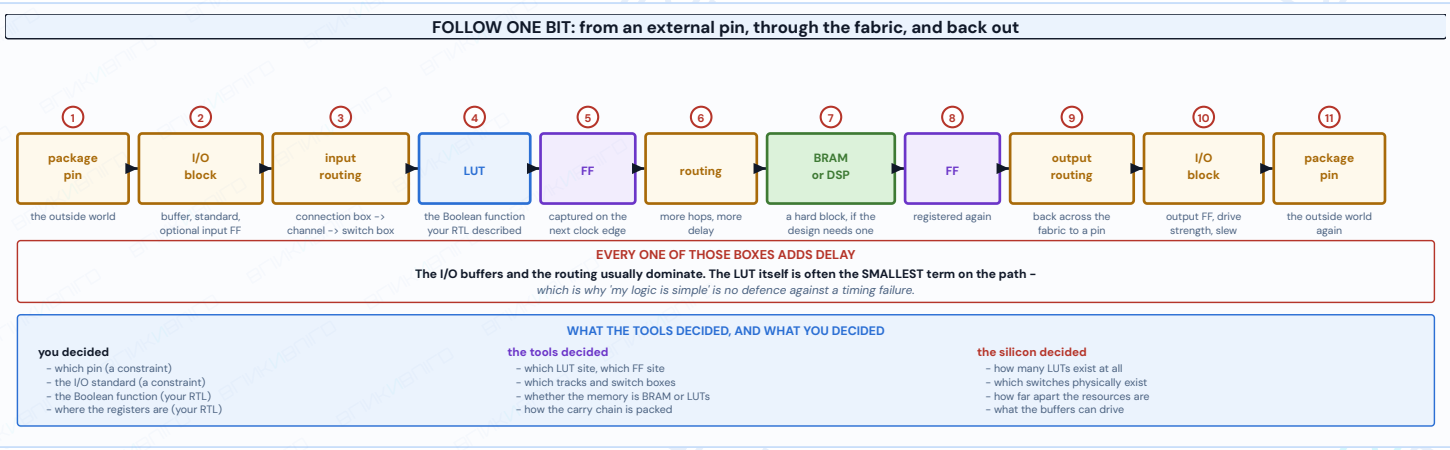
The observer effect is real. Adding a debug core consumes LUTs and BRAM, adds routing, and changes placement — so the timing changes and the bug can move or vanish. Add instrumentation early and leave it in, rather than bolting it on to chase a symptom.

// §44 "WHY IS MY FPGA DESIGN NOT WORKING?"

Symptom	Likely cause	What to check first
Nothing happens at all; DONE never asserts	configuration failed	mode pins, boot flash contents, supply sequencing, INIT/DONE pins on a scope
Works in simulation, dead in hardware	clock or reset never arrives	is the clock on a clock-capable pin? did the MMCM lock? is reset stuck asserted?
Works at 10 MHz, fails at 100 MHz	**setup** violation	timing report: WNS, and the critical path's logic/routing split
Fails at every frequency, including very slow	**hold** violation, or a logic bug	hold slack; then re-examine the RTL — slowing the clock proves nothing about hold
Works on one board, fails on another	marginal timing, or signal integrity	slack margin, I/O standard and drive, termination, temperature
Fails rarely and unrepeatably	**CDC / metastability**	CDC report; every asynchronous input; multi-bit crossings without a handshake
State machine reaches an impossible state	reset release, or an unsynchronised input	reset synchroniser; `default` in the case statement; one-hot corruption
Timing "met" but hardware is wrong	under-constrained design	unconstrained endpoints; missing I/O delays; a false path that should not be there
Output pin does nothing	wrong pin, wrong standard, or optimised away	pin constraint; bank voltage; synthesis log for removed logic
Memory reads return the wrong data	read latency or write mode assumption	is the read registered? which write mode? off-by-one cycle?
Design suddenly grew and slowed after a small edit	an inference was lost	utilisation diff: did a BRAM become LUTs, or a DSP become carry chains?
Multiplier or memory is far slower than expected	pipeline registers bypassed	are the DSP/BRAM internal registers enabled?

// §45 FOLLOW ONE SIGNAL

The question this whole sheet has been building towards: **where does one bit actually go?**

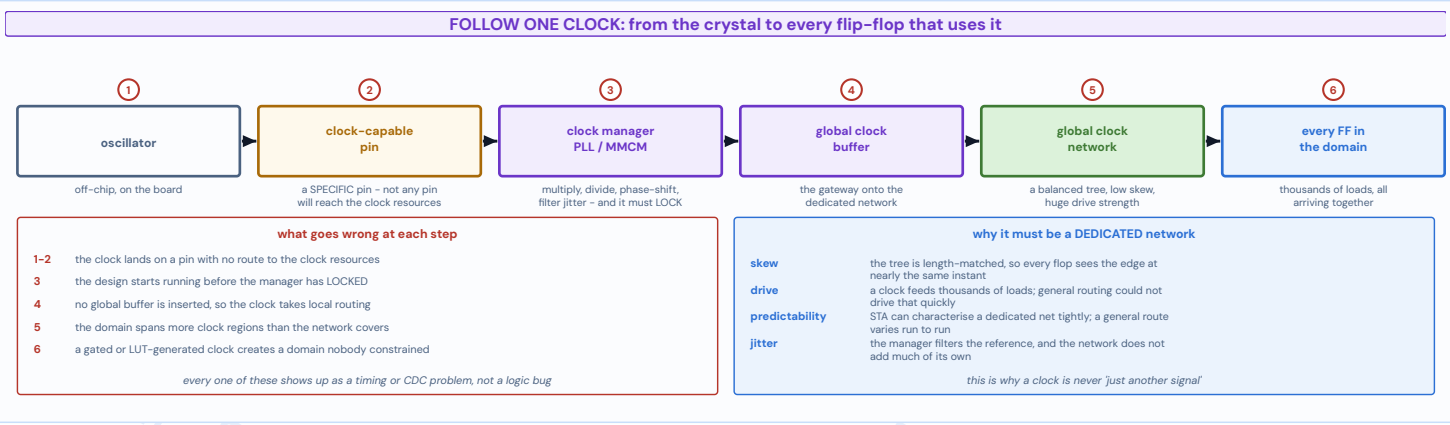


45.1 Stage by stage MUST

1. The pin	A package ball or pin, bonded to a pad on the die. Its electrical behaviour was fixed by your I/O standard constraint.
2. The I/O block	The input buffer converts the external signalling level to the internal one. If you used the input flip-flop, the signal is captured right here, before it ever meets the routing — which is what makes pin-to-register timing tight and predictable.
3. Input routing	A connection box puts the signal on a track; switch boxes carry it across the fabric to wherever the placer put the logic. **Several hops, each a real transistor.**
4. The LUT	The signal is one address bit into a truth table. The stored bit at that address appears at the output after a delay that does not depend on the function.
5. The flip-flop	On the next rising clock edge, the LUT's output is captured — provided it arrived `tsu` before that edge and held for `th` after it.
6. More routing	Out of the slice, back into the interconnect, on to whatever comes next.
7. A hard block	If the design needs memory or arithmetic, the value enters a BRAM or a DSP, with that block's own latency and its own registers.
8. Another flip-flop	Registered again. Each register boundary is a new timing path, checked separately.
9. Output routing	Back across the fabric to the I/O column — often a long way, because I/O is at the edges and logic is in the middle.
10. The I/O block	The output flip-flop (if used), then the output buffer with your chosen drive strength and slew rate, then the tri-state control.
11. The pin	Out into the world, where the board's trace, connector and receiver take over.

The two things to take from this. First, the **LUT is usually the smallest delay on the path** — the I/O buffers and the routing dominate. "My logic is simple" is no defence against a timing failure. Second, at every single stage the physical behaviour was decided by a configuration bit: which buffer, which standard, which track, which truth table, which register. The design is not *running* anywhere; it simply **is** the settings.

// §46 FOLLOW ONE CLOCK



46.1 Stage by stage MUST

1. The source	An oscillator or crystal on the board. All of your frequency accuracy comes from here and nowhere else.
2. A clock-capable pin	Only certain pins have a dedicated path to the clock resources. Land the clock on an ordinary pin and it reaches the fabric the slow way — or the tools refuse the design.
3. The clock manager	PLL or MMCM. Multiplies, divides, phase-shifts and filters jitter. It must **lock** before its outputs mean anything, and your reset should hold the design until it does.
4. The global clock buffer	The gateway onto the dedicated network. Without one, a clock is just another signal on general routing.
5. The clock network	A balanced, low-resistance, heavily buffered tree. Length-matched so every leaf sees the edge at almost the same instant, and strong enough to drive thousands of loads quickly.
6. Every flip-flop in the domain	They all sample on the same edge, within the skew the network guarantees. That guarantee is what makes synchronous design work at all.

46.2 What each stage contributes to your timing budget HIGH

Skew	From stage 5: the small differences in arrival across the tree. Static, characterised, and modelled exactly by STA.
Jitter	From stages 1 and 3: the source's noise, partly filtered by the manager, plus supply noise picked up along the way. Random, so it is budgeted as uncertainty.
Insertion delay	The time from the pin to the leaves. Matters for I/O timing, and can be cancelled by feeding the network output back as the manager's feedback path.
Domain boundaries	Anything clocked by a **different** stage-3 output, or by a different source, is a separate domain — and every signal crossing between them needs §38 treatment.

Every failure in this chain looks like something else. A clock on the wrong pin looks like "the design is dead". A design that starts before lock looks like "it works after a reset". A missing global buffer looks like "it fails timing for no reason". A gated clock looks like "it works on my board". None of them look like clocking problems until you go and check the clocking.

// §47 FOLLOW ONE RTL DESIGN

The central mental model of the whole sheet. If you can narrate this, you understand FPGAs.

'I WROTE THIS VERILOG. WHAT HAPPENS NEXT?'

```
always_ff @(posedge clk)
q <= (a & b) ^ c;
```

1 ELABORATION	the tool works out what you described: one register, one Boolean expression feeding it *no hardware chosen yet – this is still just meaning*
2 LOGIC SYNTHESIS	the expression becomes a Boolean network; constants fold, duplicates merge, dead logic goes *your two operators may not survive as two operators*
3 TECHNOLOGY MAPPING	(a & b) ^ c is a 3-input function -> it fits entirely inside ONE LUT. q becomes ONE flip-flop *on a LUT6 fabric there are 3 inputs spare; the tool may pack more logic into the same LUT*
4 PLACEMENT	that LUT+FF pair is assigned a specific site in a specific slice, in a specific tile *chosen so a,b,c and q have short routes – this is where timing is really decided*
5 ROUTING	a, b and c are routed from wherever they come from; q is routed to wherever it goes; clk comes from the clock network *the LUT delay is fixed; the ROUTING delay is what varies*
6 STATIC TIMING ANALYSIS	tcq + LUT + routing + tsu is compared against your clock constraint *pass -> continue. Fail -> back to step 1 with a different architecture*
7 BITSTREAM	64 bits set the LUT's truth table; more bits set the FF's mode and every routing mux on those nets *your Boolean expression is now literally a set of stored bits*
8 CONFIGURATION	the bitstream is loaded; at End Of Startup the FF is released and the I/O turns on *nothing is 'started' – the circuit simply now exists*
9 OPERATION	on every rising clock edge, that one flip-flop samples whatever the LUT is presenting *for ever, in parallel with every other flop, at one result per cycle*

One line of RTL. One LUT, one flip-flop, a handful of routing switches, and about a hundred configuration bits.
Scale that to a hundred thousand lines and you have an FPGA design – and every step above still applies to every one of them.

Say this out loud in an interview. "I write RTL describing registers and the logic between them. Synthesis turns that into a Boolean network and optimises it. Technology mapping packs that network into **this device's** LUTs, flip-flops, carry chains, DSPs and BRAMs. Placement assigns every one of them a physical site; routing gives every net real tracks and switches. Static timing analysis then checks every path against my constraints. If it passes, bitstream generation writes out the value of every configuration bit, and configuration loads them. At End Of Startup the fabric **is** my circuit, and it runs every stage in parallel, one result per cycle, until power is removed."

// §48 FOUR MORE WALKTHROUGHS

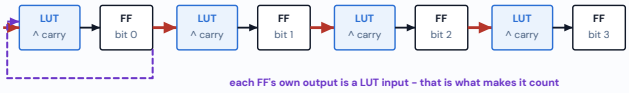
48.1 A counter MUST

WALKTHROUGH: A COUNTER

```
always_ff @(posedge clk)
count <= count + 1;
```

→ an INCREMENTER, not an adder: *one operand is the constant 1*

the red hops are the DEDICATED CARRY CHAIN



each FF's own output is a LUT input – that is what makes it count

So an N-bit counter costs roughly N LUTs, N flip-flops and one carry chain – and it is FAST,
because the only long path is the dedicated carry, not general routing.

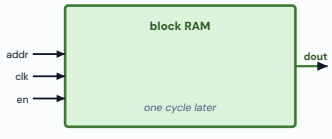
The carry chain must stay in one column, so a very wide counter constrains the placer.
A terminal-count comparator (count == 999) is a separate wide AND – often the real critical path, which is why a free-running counter plus a small decode is usually faster than a loadable one.

48.2 A memory MUST

WALKTHROUGH: A MEMORY

```
logic [7:0] mem [0:1023];
always_ff @(posedge clk)
dout <= mem[addr];
```

the tool sees	→ an array, read synchronously
it counts	→ 1024 x 8 = 8192 bits
it decides	→ too big for LUTs -> a BRAM
it wires up	→ addr, clk and en to the ports
you get	→ one cycle of read latency, free



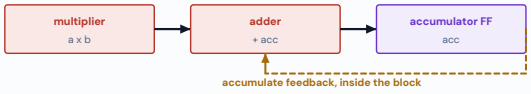
Write it ASYNCHRONOUSLY (assign dout = mem[addr];) and the block RAM cannot be used at all –
the tool falls back to distributed RAM in LUTs, and a big array will then simply fail to fit.
Coding style is what selects the resource – this is the commonest memory-inference mistake there is.

48.3 A multiply-accumulate HIGH

WALKTHROUGH: MULTIPLY-ACCUMULATE

```
always_ff @(posedge clk)
acc <= acc + a * b;
```
a and b are 16-bit

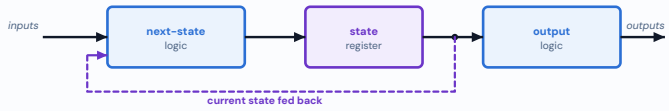
→ **the tool recognises the MAC shape and maps it to ONE DSP block**
16x16 fits inside a 25x18 multiplier



If you PIPELINE it (register a, b, the product and the accumulator) the DSP runs at its rated speed.
If you write it as one combinational lump, you get the same arithmetic at a fraction of the frequency – the block's registers exist whether you use them or not.

48.4 A state machine MUST

WALKTHROUGH: A STATE MACHINE



next-state logic -> LUTs. state register -> flip-flops. output logic -> more LUTs.
There is no 'state machine' resource on the die. It is the same LUTs and FFs as everything else.

ENCODING is your lever: binary uses fewest FFs but deeper decode logic; one-hot uses one FF per state
with very shallow logic – which usually wins, because FFs are plentiful and LUT depth costs time.

A Mealy machine also routes the inputs into the OUTPUT logic, so its outputs can glitch mid-cycle – register them before they leave the chip.

48.5 The pattern behind all four MUST

Every one of these follows the same three steps, and so will anything else you write:

1. What shape did I describe? An increment, an array read, a multiply-accumulate, a state register with two logic clouds.
2. What resource does that shape map to? Carry chain, block RAM, DSP block, LUTs and flip-flops.
3. What does the coding style have to look like to get it? Use the operator; register the read; register the DSP's inputs and outputs; give the case statement a default.

When a design comes out bigger or slower than expected, one of those three answers was wrong — and the utilisation and inference reports will tell you which.

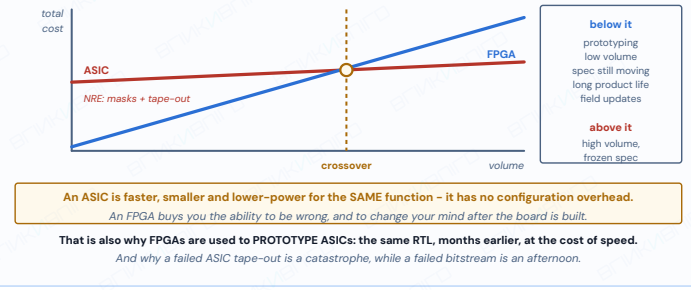
// §49 FPGA VS CPLD VS ASIC

49.1 FPGA vs CPLD GOOD

	CPLD	FPGA
Logic style	a few large blocks, typically wide sum-of-products arrays	many small blocks: LUTs and flip-flops
Interconnect	a central, relatively uniform switch matrix	a rich, hierarchical, distributed routing fabric
Timing	very **predictable** pin-to-pin — few paths, all similar	depends entirely on placement and routing
Capacity	small: glue logic, address decode, board management	small to enormous
Configuration	usually non-volatile and instant-on	usually SRAM, needing a boot device
Typical job	power sequencing, level shifting, "make these three chips talk"	the actual system

The boundary has blurred: small non-volatile FPGAs now occupy much of what used to be CPLD territory, and some vendors market the same silicon under either label. Treat these as tendencies, not definitions.

49.2 FPGA vs ASIC MUST



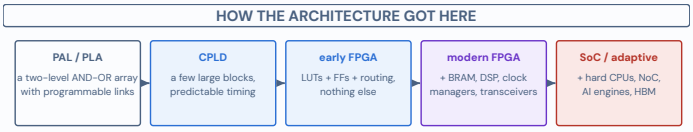
An ASIC is faster, smaller and lower-power for the SAME function - it has no configuration overhead.
An FPGA buys you the ability to be wrong, and to change your mind after the board is built.

That is also why FPGAs are used to PROTOTYPE ASICs: the same RTL, months earlier, at the cost of speed.
And why a failed ASIC tape-out is a catastrophe, while a failed bitstream is an afternoon.

Dimension	FPGA	ASIC
NRE cost	essentially zero	very large — masks, verification, tape-out
Unit cost	high, and it does not fall much	very low at volume
Performance	limited by the configurable interconnect	several times faster for the same function
Power	configuration overhead is always present	far lower
Area	much larger for the same logic	minimal
Time to hardware	minutes to hours	months
Cost of a mistake	rebuild and reload	a new mask set
Field updates	**yes** — a headline feature	no

The relationship people miss. FPGAs are not only an alternative to ASICs — they are how ASICs get **prototyped**. The same RTL runs on an FPGA months before silicon exists, at lower speed, so firmware and software can be developed and the design exercised against real stimulus. Large designs are split across several FPGAs on emulation platforms for exactly this reason.

// §50 HOW THE ARCHITECTURE GOT HERE



The direction of travel is always the same: whatever people kept building out of LUTs eventually became a hard block.

Multipliers, memories, clock managers, serial links, processors, and now AI engines - each was fabric first, and hardened once it was common enough to justify the silicon.

Not every device has all of it. Read the family's own documentation before assuming a resource exists.

// §51 THE TOOLCHAIN

Role	What it does	Examples
Editor / project	manages sources, IP and constraints	vendor IDEs, or any editor plus a makefile
Simulator	runs your RTL and testbench	vendor simulators; Verilator, Icarus, cocotb
Synthesis	RTL to netlist	AMD Vivado, Intel Quartus, Lattice Radiant/Diamond, Microchip Libero; Yosys
Implementation	map, place and route	the same vendor tools; nextpnr for supported open targets
Timing analysis	checks every constrained path	built into the vendor implementation tools
Programmer	loads the bitstream over JTAG or into flash	vendor hardware managers; openFPGALoader
Hardware debug	on-chip logic analysis over JTAG	ILA-style cores and their vendor front ends

Tool and device support varies enormously. The open-source flow (Yosys plus nextpnr) is excellent on the device families it supports and simply does not exist for others, because the bitstream formats had to be reverse-engineered. Vendor tools support their own devices completely, are very large, and are version-sensitive — a design that closes timing on one release may not on the next. Pin the tool version in any project you expect to rebuild.

// §52 HANDS-ON LEARNING ROADMAP

Each experiment is chosen to make **one** piece of internal architecture visible. Build them in order; the point is not the project, it is what the project forces you to understand.

#	Build	You learn	The architecture it exposes
1	LED blink	clock, counter, output pin, constraints	the clock **network**, a flip-flop, an I/O block, and the fact that a pin needs a constraint before anything works
2	Free-running counter	register banks, carry propagation	the **carry chain**: watch the LUT and FF count scale linearly with width while fmax barely moves
3	7-segment display	combinational decode, time multiplexing	pure **LUT logic**, plus your first real fanout and a clock divider you should build with an **enable**, not a gated clock
4	PWM generator	counter plus comparator	a comparator on the **carry chain**, and the first time a wide compare shows up on your critical path
5	UART transmit and receive	FSM, baud timing, oversampling, serial framing	**LUTs + flip-flops as a state machine**, and your first genuine asynchronous input — the RX line needs a synchroniser
6	Synchronous FIFO	pointers, full/empty, memory inference	**BRAM inference**: write the read asynchronously and watch it turn into hundreds of LUTs instead
7	FSM controller with a datapath	separating control from data	the **control path / datapath** split, and state encoding as a real timing lever
8	BRAM with an initialised ROM	memory ports, latency, initial contents	**synchronous read latency** — the one-cycle delay you must design around, and configuration-loaded memory contents
9	FIR filter	MAC, coefficients, fixed point	**DSP blocks and their pipeline registers**: build it unpipelined, look at fmax, then pipeline it and look again
10	Two clock domains + async FIFO	CDC, Gray code, synchronisers	**metastability** in practice, and why the FIFO is the standard answer
11	AXI-Lite peripheral	memory-mapped registers, handshaking	**interconnect**: valid/ready, address decode, and the boundary between software and fabric
12	SoC FPGA integration	processor plus accelerator, DMA	**hardware/software partitioning**, and how quickly the interconnect becomes the bottleneck

// §53 PROJECT ROADMAP

Longer builds. For each one, ask yourself before you start: **which internal resources will this use, and how will I know?**

Project	Resources it will consume
1. LED blink	1 clock buffer, ~30 flip-flops, 1 carry chain, 1 output pin
2. Binary counter with display	flip-flops, a carry chain, a decode LUT cloud, several output pins
3. PWM / servo driver	counter, comparator, carry chain
4. Multiplexed 7-segment controller	counter, decoder LUTs, a fanout net worth watching
5. UART bridge	two FSMs, a baud counter, a synchroniser on RX
6. SPI master	FSM, shift registers (possibly SRLs), clock enable rather than a divided clock
7. FIFO with flags	BRAM or distributed RAM, pointer counters, comparators
8. Traffic-light / vending FSM	flip-flops, LUTs, one-hot encoding worth measuring both ways
9. BRAM-backed frame or line buffer	several BRAMs, address counters, and real read-latency design
10. FIR / moving-average filter	DSP blocks, their cascade path, coefficient storage, deliberate pipelining
11. Dual-clock video or audio pipeline	async FIFO, Gray-coded pointers, two clock domains, two reset synchronisers
12. AXI-Lite register block	address decode LUTs, register flip-flops, handshake logic
13. Hardware accelerator with DMA	AXI master, BRAM buffers, DSPs, a deep pipeline, and a real throughput measurement
14. Full SoC FPGA application	hard processor, interconnect, custom peripherals, accelerator, DDR controller

The habit that turns projects into understanding. After every build, open three reports before you open the board: **utilisation** (did I get the resources I expected?), **timing** (what is the critical path made of — logic or routing?), and the **synthesis log** (what did it infer, and what did it remove?). Five minutes there teaches more than an hour of poking at LEDs.

// §54 FPGA MYTHS

MYTH	REALITY
× An FPGA executes my HDL	✓ Nothing executes anything. The HDL described a circuit; the tools built it; the bitstream froze it into configuration memory.
× An FPGA is just a big pile of gates	✓ It is LUTs, flip-flops, **routing**, and a large amount of dedicated hard silicon — memory, arithmetic, clocking, I/O, transceivers.
× More LUT usage means slower	✓ Utilisation and speed are almost independent. What costs time is **logic depth**, **fanout** and **distance**.
× Less LUT usage means faster	✓ A 20%-full design with a fifteen-LUT-deep path will be slow. Small does not mean shallow.

MYTH	REALITY
× Simulation passed, so it works	✓ Simulation checked the logic against the stimulus you imagined, with no delays. It says nothing about timing, metastability or your assumptions.
× Timing can be sorted out later	✓ Timing is decided by the architecture you chose in RTL. Late-stage tool options recover a few percent, not a factor of two.
× A clock is just another signal	✓ It is the one net that must reach thousands of loads with bounded skew. It gets a dedicated network for exactly that reason.
× A 2-flop synchroniser solves CDC	✓ It handles **one bit**, and reduces failure probability rather than eliminating it. Multi-bit crossings need a handshake, Gray code, or a FIFO.
× All FPGA architectures are the same	✓ LUT sizes, logic-block structure, memory sizes, DSP widths and clocking all differ. Vendor terms do not translate.
× All FPGAs are SRAM-based	✓ Flash-based and antifuse devices exist and are chosen for non-volatility, instant-on and radiation tolerance.
× A bitstream is compiled Verilog	✓ It is the settings for every configurable point in one specific device. It is not executed and is not portable.
× An FPGA is always faster than a CPU	✓ It clocks far slower. It wins on wide, regular, parallel work and loses badly on branchy, irregular code.
× The synthesis tool builds what I wrote	✓ It builds something with the same **behaviour**. Logic is merged, shared, retimed and deleted along the way.

// §55 THIRTY TRAPS

TRAP	REALITY
× HDL is software	✓ It is a hardware description. Statements are structure, not steps.
× An FPGA is a CPU	✓ No instruction stream, no program counter, no sequencing you did not build.
× A LUT is a software lookup	✓ It is `2^N` SRAM cells read through a mux tree, in one fixed delay.
× A complex function costs more LUT delay	✓ LUT delay is independent of the function. Depth costs time; complexity does not.
× Simulation equals timing closure	✓ Different questions entirely.
× Low utilisation means high performance	✓ Unrelated. Check logic depth and fanout.
× More LUTs means worse timing	✓ Not automatically. Wide and shallow beats narrow and deep.
× Fewer LUTs means better timing	✓ Not automatically. Resource sharing can lengthen a path.
× A clock can use ordinary routing	✓ Use a clock buffer and the dedicated network, always.
× Routing is an ideal zero-delay wire	✓ It is a chain of transistors and usually the largest delay term.
× 2-FF is a universal CDC solution	✓ One bit only, and probabilistic even then.
× A bitstream is source code	✓ It is device-specific configuration data.
× Synthesis equals placement	✓ Synthesis chooses *what*; placement chooses *where*.
× Placement equals routing	✓ Placement chooses sites; routing chooses tracks.
× Routing equals timing closure	✓ Routing is a step; closure is a result you may not reach.
× An FPGA is a cheap ASIC	✓ Higher unit cost, lower speed, more power — bought for flexibility.
× A gated clock saves power safely	✓ On an FPGA it creates skew and an unconstrained domain. Use a clock enable.
× A divided clock from a counter is fine	✓ It is a fabric-routed clock. Use a clock manager.
× Asynchronous reset is simply safer	✓ Assertion is; **release** is a hazard needing a synchroniser.
× Reset everything, to be safe	✓ Every reset load is fanout, area and power for ever.
× Block RAM reads combinationally	✓ It is synchronous — address in on one edge, data out after it.
× Read-during-write behaviour is obvious	✓ It is a configured mode with three possible answers.
× One multiply equals one DSP	✓ A wide multiply needs several; a constant multiply may need none.
× Instantiating a DSP makes it fast	✓ Only with its pipeline registers enabled.
× A false path fixes a violation	✓ It stops the tool checking. The hardware still fails.
× Timing met means fully analysed	✓ Check the unconstrained-endpoint count first.
× Any pin can take a clock	✓ Only clock-capable pins reach the clock resources.
× Any two pins can form a differential pair	✓ Pairs are fixed by the package, and share a bank voltage.
× The same RTL gives the same result every run	✓ Placement and routing are heuristic. Seeds and versions matter.
× The tools will tell me if something is wrong	✓ They will — in a warning, among a thousand others. Read them.

// §56 FIFTY-FIVE FPGA INTERVIEW QUESTIONS

Answers are deliberately short — the length you should actually speak. Where a question invites a vendor-specific answer, the honest response names a family rather than generalising.

56.1 Fundamentals MUST

1. What is an FPGA?	An array of programmable logic blocks embedded in a programmable interconnect, with dedicated hard blocks for memory, arithmetic, clocking and I/O. Its structure — not its behaviour over time — is set by configuration data.
2. Does an FPGA execute HDL?	No. The HDL describes a circuit. Tools synthesise, map, place and route it, then write a bitstream that configures the fabric to **be** that circuit.
3. FPGA vs CPU?	A CPU is a fixed circuit whose behaviour changes over time under an instruction stream. An FPGA is a configurable circuit built once, running everything concurrently. CPUs win on irregular work; FPGAs on wide, regular, latency-critical work.
4. FPGA vs ASIC?	ASIC: far faster, smaller, lower power, huge NRE, no field changes. FPGA: reconfigurable, no NRE, higher unit cost and power. The crossover is a volume question.
5. Why is an FPGA slower than an ASIC?	Everything goes through configurable interconnect — programmable switches with real resistance and capacitance — and generic LUTs instead of purpose-built gates.
6. What is "programmable hardware"?	Configuration memory decides what each LUT computes and which wires are joined. The transistors are fixed; the stored settings are not.

56.2 LUT, logic block, carry MUST

7. What is a LUT?	`2^N` configuration bits read out by a mux tree that the N logic inputs address. It implements any Boolean function of N inputs.
8. How many bits in a 6-input LUT?	64 — one per truth-table row.
9. Does a complex function make a LUT slower?	No. Vendor documentation states the delay is independent of the function. Depth — LUTs in series — is what costs time.
10. Why 4 to 6 inputs?	Bigger LUTs reduce logic depth but waste area on small functions and grow as `2^N`. Four to six is the long-established balance; vendors add fracturing to recover the waste.
11. What is a slice / CLB / ALM / LAB?	Vendor names for the logic block. `AMD` 7-series: a slice is four LUT6 plus eight storage elements, and a CLB is two slices. `INTEL`: an ALM is one 8-input fracturable LUT, two adders and four registers; a LAB holds ten.
12. SLICEL vs SLICEM?	`AMD` terms. Both do logic; a SLICEM's LUTs can additionally be distributed RAM or shift registers. Roughly a third of slices are SLICEM.
13. Why a dedicated carry chain?	Ripple carry through general routing would cost a switch-box round trip per bit. A hard path makes an N-bit add N cheap mux delays instead.
14. What constrains a carry chain?	It runs vertically between adjacent blocks, so it must be placed contiguously in one column. Wide arithmetic therefore constrains placement.
15. Why are FPGA multiplexers everywhere?	Every programmable choice — routing track, registered or bypassed output, LUT readout — is a mux driven by configuration bits.

56.3 Routing MUST

16. What is a switch box?	The structure at a channel intersection that joins tracks in one channel to tracks in another. Sparsely populated — a full crossbar is unaffordable.
17. What is a connection box?	What attaches a logic block's pins to some of the tracks in the adjacent channel. Different job from a switch box.
18. How is routing programmed?	Each switch or routing mux select is a configuration bit. After configuration a route is a chain of transistors that stored bits turned on.
19. Why does routing dominate timing?	Delay grows with the number of switch hops, each contributing resistance and capacitance. On a long path routing typically exceeds logic.
20. Why can identical RTL have different timing?	Timing is a property of the physical implementation. Different placement gives different routing gives different delay.
21. What is routing congestion?	Demand for tracks in a region exceeding supply, forcing detours. It shows up as unstable, placement-sensitive timing.

56.4 Configuration and bitstream MUST

22. What is a bitstream?	The value of every configuration bit in a specific device: LUT contents, routing selects, block options, I/O settings, memory initial contents.

23. Is a bitstream compiled Verilog?	No. It is not executed, it does not resemble your source, and it is device-specific.
24. SRAM vs flash vs antifuse?	SRAM: volatile, needs a boot device, unlimited reconfiguration. Flash: non-volatile, live at power-up, limited write cycles. Antifuse: one-time, permanent, inherently upset-immune.
25. What happens at power-up?	Power up, clear configuration memory, sample mode pins, synchronise, check device ID, load frames, CRC, then a startup sequence that releases DONE, enables I/O and enables writes to registers and RAM.
26. Why clear configuration memory first?	So no stale settings survive; the device must not come up as a mixture of two designs.
27. What is partial reconfiguration?	Reloading part of the configuration memory while the rest keeps running — used to time-multiplex accelerators or update one function in the field.

56.5 Memory and DSP MUST

28. Block RAM vs distributed RAM?	BRAM: kilobits, dedicated blocks, synchronous read, fixed pool. Distributed: tens of bits per LUT, often asynchronous read, consumes logic.
29. Why is a BRAM read synchronous?	It is a clocked memory array. Address in on one edge, data valid after it — plus another cycle if the output register is enabled.
30. What is true dual port?	Two fully independent ports — own clock, address, width, enable — sharing only the stored data.
31. What happens on a same-cycle read and write to one address?	It depends on the configured write mode. AMD names three: WRITE_FIRST (new data, the default), READ_FIRST (old data), NO_CHANGE (output holds).
32. What is in a DSP block?	A pre-adder, a multiplier, an adder/accumulator, optional pipeline registers at every stage, and dedicated cascade ports. AMD DSP48E1: 25x18 multiplier, 48-bit accumulator.
33. Why does a DSP have cascade ports?	So filter chains pass partial results block to block on dedicated wires, never touching general routing.
34. When should you NOT use a DSP?	Constant multiplies (shifts and adds), very narrow multiplies, or when the DSPs are needed elsewhere — they are a fixed, countable resource.

56.6 Clocking and timing MUST

35. Why do clocks need dedicated networks?	Bounded skew across thousands of loads, enough drive to switch them quickly, and delays tight enough for STA to model accurately.
36. What does a PLL/MMCM do?	Frequency multiply and divide, phase shift, jitter filtering, and clock de-skew. It cannot improve on the reference's accuracy.
37. Skew vs jitter?	Skew is a spatial, static difference in arrival between two points — modelled exactly, and can help setup. Jitter is temporal, random variation at one point — budgeted as uncertainty, and only ever costs margin.
38. What is setup time?	How long data must be stable before the capture edge. Violated when the path is too slow; fixable by slowing the clock.
39. What is hold time?	How long data must stay stable after the edge. Violated when the path is too fast; **not** fixable by slowing the clock.
40. What is slack?	Required time minus arrival time. Positive passes, negative is a violation. WNS is the worst path; TNS is the sum of all violations.
41. What is the critical path?	The path with worst slack. It alone sets fmax — speeding up any other path changes nothing.
42. What is static timing analysis?	Vector-free checking of every timing path against constraints, over PVT corners. It checks paths a testbench would never exercise.
43. Why is a false path dangerous?	It stops the tool checking the path. If the path is real, a build failure becomes a field failure.

56.7 CDC, pipelining, practice MUST

44. What is metastability?	A flip-flop sampled inside its setup/hold aperture settling to an unpredictable value after an unbounded time.
45. Does a 2-flop synchroniser eliminate it?	No. It gives the first flop a full period to resolve, reducing failure probability to an MTBF you accept. One bit only.
46. How do you cross a multi-bit value?	Handshake, Gray code, or an asynchronous FIFO. Never parallel synchronisers on independent bits.
47. Why Gray code in an async FIFO?	One bit changes per increment, so a pointer sampled mid-change is either the old or the new value — never a nonsense combination.
48. Why does pipelining raise fmax?	It shortens the longest register-to-register path. Throughput rises; latency rises too.
49. When does pipelining not help?	Feedback loops, routing-dominated paths where the register cannot be placed usefully, and paths that are already one LUT deep.
50. Why is high fanout a problem?	Each load adds capacitance and usually a routing hop. Fix by replicating the driver, pipelining, or reducing the load count.
51. Sync or async reset?	Assert asynchronously so it works before the clock is stable; release synchronously through a reset synchroniser so every flop leaves reset on the same edge. One per clock domain.
52. Hard IP vs soft IP?	Hard: dedicated silicon — fast, efficient, fixed in number and position. Soft: built from fabric — flexible and plentiful, but it consumes LUTs, FFs and timing budget.
53. What is a SoC FPGA?	A hard processor system beside programmable logic, joined by a memory-mapped interconnect. Control goes in software; wide regular real-time work goes in the fabric.
54. Why do transceivers recover the clock from data?	At multi-gigabit rates a separate clock wire would skew relative to the data. Line coding guarantees the transitions CDR needs.
55. Your design passes simulation and fails on hardware. What do you check?	In order: is the clock arriving and locked; is reset released properly; timing report WNS and unconstrained endpoints; the CDC report; then pin and I/O-standard constraints. Only then reopen the RTL.

// §57 ONLINE-ASSESSMENT MCQS

Written to test understanding rather than recall. Cover the answer, commit, then read the reasoning — the reasoning is the point.

Q1. A 6-input LUT holds how many configuration bits?
(a) 6 (b) 12 (c) 36 **(d) 64**

(d) One bit per truth-table row, and a 6-input truth table has `2^6 = 64` rows.

Q2. Implementing a 6-input XOR instead of a 6-input AND in one LUT will:
(a) be slower (b) use more LUTs **(c) cost exactly the same** (d) fail to synthesise

(c) LUT delay is independent of the stored function. Only the configuration bits differ.

Q3. Most configuration bits in a typical FPGA control:
(a) LUT contents **(b) routing multiplexer selects** (c) I/O standards (d) BRAM contents

(b) The interconnect dominates both die area and bit count. Logic is the smaller share.

Q4. A carry chain must be placed:
(a) anywhere (b) near the I/O **(c) contiguously in one column** (d) inside a DSP

(c) The dedicated carry wires only run between vertically adjacent logic blocks.

Q5. Which is **not** normally a function of a clock manager?
(a) frequency multiplication (b) phase shift (c) jitter filtering **(d) improving the reference's frequency accuracy**

(d) Accuracy comes from the crystal. The manager only synthesises ratios of it.

Q6. A block RAM read is:
(a) asynchronous **(b) synchronous, with at least one cycle of latency** (c) combinational (d) tool-dependent

(b) Address in on one edge, data out after it — plus another cycle if the optional output register is enabled.

Q7. Writing `assign dout = mem[addr];` for a 1024-entry array will most likely:
(a) infer a block RAM **(b) infer distributed RAM in LUTs** (c) infer a DSP (d) fail to synthesise

(b) The read is asynchronous, so a block RAM cannot be used. A large array then consumes a great many LUTs — and may not fit.

Q8. A hold-time violation can be fixed by:
(a) lowering the clock frequency (b) adding a pipeline stage **(c) adding delay on the data path** (d) raising the supply voltage

(c) The clock period does not appear in the hold equation. Hold is a race between two paths launched by the same edge.

Q9. Slack is defined as:
(a) arrival - required **(b) required - arrival** (c) setup + hold (d) Tclk - tcq

(b) Positive means margin; negative means violation.

Q10. Positive clock skew (capture clock later than launch):
(a) hurts setup, helps hold **(b) helps setup, hurts hold** (c) affects neither (d) affects only jitter

(b) A later capture edge gives the data more time to arrive, but also more time for the *next* data to race through.

Q11. A design uses 30% of the LUTs and fails timing. The most likely cause is:
(a) not enough LUTs **(b) deep logic, high fanout or poor locality** (c) too few flip-flops (d) the device is too large

(b) Utilisation and timing are nearly independent. Depth, fanout and distance are what cost time.

Q12. Pipelining a datapath:
(a) reduces latency (b) reduces throughput **(c) increases both throughput and latency** (d) has no timing effect

(c) More stages, so more cycles end to end — but a result every cycle at a higher clock rate.

Q13. Gray code is used for asynchronous FIFO pointers because:
(a) it is faster to increment (b) it uses fewer bits **(c) only one bit changes per step** (d) it prevents overflow

(c) A pointer sampled mid-change is therefore either the old or the new value — never an invalid combination.

Q14. A 2-flop synchroniser:
(a) eliminates metastability **(b) reduces its probability to an acceptable MTBF** (c) works for any bus width (d) removes the need for constraints

(b) And it is valid for one bit at a time.

Q15. Dividing a clock with a fabric counter and using the result as a clock is bad because:
(a) it wastes LUTs **(b) it puts a clock on general routing, creating skew and an unconstrained domain** (c) counters are slow (d) it violates hold

(b) Use a clock manager, or a clock enable on the original clock.

Q16. Which is **not** a step in configuring an SRAM FPGA?
(a) clear configuration memory (b) check the device ID (c) CRC check **(d) compile the HDL**

(d) Compilation happened long before, on your workstation. Configuration only loads the result.

Q17. A flash-based FPGA differs from an SRAM one mainly in that it:
(a) is faster (b) has more LUTs **(c) retains its configuration without power and is live at power-up** (d) needs no bitstream

(c) It still needs a bitstream — it just keeps it.

Q18. Declaring a real path as a false path:
(a) makes it faster (b) adds buffers **(c) stops the tool checking it** (d) improves WNS legitimately

(c) The report improves; the silicon does not. This converts a build failure into a field failure.

Q19. Synthesis deleted a signal you wanted to observe. Most likely:
(a) a tool bug **(b) it drives no register and no pin** (c) it was too wide (d) it crossed clock domains

(b) Dead logic elimination is correct behaviour. Mark it with a keep attribute if you need it.

Q20. Which will most reliably map to a DSP block at full speed?
(a) `assign p = a*b;` (b) a hand-built shift-add multiplier **(c) a registered MAC with registered inputs and output** (d) `a * 8`

(c) (a) may map to a DSP with the pipeline bypassed; (b) usually loses the inference; (d) is a shift.

Q21. On an FPGA, an internal tri-state bus is normally implemented as:
(a) tri-state buffers in the fabric **(b) a multiplexer** (c) open-drain LUTs (d) a block RAM

(b) Modern fabrics have no internal tri-state. Only the I/O buffers are genuinely tri-state.

Q22. An asynchronous reset is dangerous mainly at:
(a) assertion **(b) release** (c) power-up (d) configuration

(b) Release near a clock edge lets different flops leave reset in different cycles.

Q23. "One ALM equals one slice" is:
(a) true (b) true for logic only **(c) false — they are different vendors' units and do not translate** (d) true after a scaling factor

(c) Different LUT sizes, different fracturing, different register counts. Compare within a family only.

Q24. After placement and routing, the delay on a long net is dominated by:
(a) the LUT (b) the flip-flop **(c) the number of switch hops** (d) the clock manager

(c) Each hop is a transistor plus the next segment's capacitance.

Q25. A clock arrives on a pin that is not clock-capable. The most likely result is:
(a) it works normally **(b) the tools object, or the clock is routed poorly with large skew** (c) higher fmax (d) a CRC error

(b) Only certain pins have a dedicated path to the clock resources.

// §58 DESIGN-ANALYSIS QUESTIONS

The kind asked out loud, where the reasoning matters more than the answer.

A design uses only 30% of the LUTs but fails timing badly. Why?

A Utilisation says nothing about speed. Look for: deep combinational logic between registers (many LUTs in series); a net with enormous fanout; logic scattered so far apart that routing dominates; an unconstrained or badly constrained clock; or a path through I/O that was never analysed. Check the critical path's **logic vs routing** split first — it tells you which of those it is.

Why does adding pipeline registers increase maximum frequency?

A Fmax is set by the longest register-to-register path. Inserting a register splits that path into two shorter ones, so the longest remaining path — and therefore the minimum clock period — is smaller. It works only if the tool can also **place** the new register somewhere sensible; otherwise the routing that dominated the path is simply divided badly.

How can adding a register increase latency but improve throughput?

A Latency is cycles from input to output for one item, and you added a cycle. Throughput is items per second: after the pipeline fills, one result emerges per cycle, and the cycles are now shorter. So each item takes longer, but more items finish per second. For a stream this is a clear win; for a single-shot latency-critical path it is a loss.

Why does high fanout hurt timing?

A Every load adds capacitance to the driver and usually another routing hop to reach it. The edge slows and the net's delay becomes hard to control, because the loads are scattered across the die. Fixes: replicate the driver, pipeline the signal, or reduce the number of loads — which usually means not resetting flops that do not need resetting.

Why do clocks need dedicated networks?

A A clock is the highest-fanout, highest-activity net in the design and must reach every flip-flop with bounded skew. General routing gives unequal path lengths, insufficient drive and poorly characterised delay. A dedicated tree is length-matched, heavily buffered, and tightly characterised so timing analysis can model it accurately rather than pessimistically.

Two logically identical RTL implementations have different timing. How?

A Timing is a property of the physical implementation, not the logic. Different synthesis structuring, different technology mapping, different placement and therefore different routing all change the delays. Even the **same** RTL with a different tool seed can differ. This is why timing must be measured after routing, not estimated from the source.

Why can synthesis remove logic you wrote?

A Because it preserves **behaviour at the boundaries**, not your structure. Logic whose output reaches no register and no pin is unobservable and is deleted. Constants propagate, duplicate logic merges, unreachable branches disappear. If you need a signal to survive for debug, give it an observable destination or a keep attribute.

A design passes simulation but fails on hardware. Name five causes.

A (1) Timing violations — RTL simulation has no delays. (2) Metastability on an asynchronous input, which cannot occur in an RTL simulator. (3) A reset that is never released properly, or released asynchronously. (4) A clock manager that has not locked before the design starts. (5) Constraints that do not describe reality — wrong I/O delays, a false path that should not be there, or unconstrained paths never analysed at all. A sixth, very common: the testbench models the external device incorrectly.

// §59 FIFTY THINGS TO MEMORISE

1. An FPGA is a configurable hardware fabric — it does not execute your HDL, or the bitstream.
2. The transistors are fixed at manufacture; only **stored configuration bits** change.
3. An N-input LUT holds exactly `2^N` configuration bits, one per truth-table row.
4. LUT delay is **independent of the function**. Logic **depth** costs time; complexity does not.
5. Configuration bits are the mux **data**; your logic signals are the mux **select**.
6. Most configuration bits are **routing** mux selects, not LUT contents.
7. Mainstream fabrics use 4- to 6-input LUTs; larger ones are fractured to avoid waste.
8. A logic block = LUTs + flip-flops + muxes + carry logic + local routing, whatever it is called.
9. `AMD` 7-series: a slice is 4x LUT6 + 8 storage elements; a CLB is two slices.
10. `INTEL` ALM: one 8-input fracturable LUT, two adders, four registers; ten per LAB.
11. Vendor logic-block units do **not** translate between vendors. Compare within a family only.
12. Flip-flops are plentiful and nearly free; LUTs and routing are what you run out of.
13. A clock enable uses a **dedicated flip-flop pin** and costs nothing. Use it instead of a gated clock.
14. Dedicated carry chains make adders, counters and comparators cheap and fast.
15. A carry chain must be placed contiguously in **one column** — it constrains placement.
16. Routing is switch boxes (channel to channel) and connection boxes (block pin to channel).
17. Every routing hop is a real transistor. Delay grows with **hops**, not just distance.
18. Routing usually contributes **more** delay than logic on a long path.
19. Identical RTL can have very different timing — timing is a property of the implementation.
20. Utilisation percentage does not predict performance. Depth, fanout and locality do.
21. SRAM FPGAs are volatile and need an external boot device; flash and antifuse parts do not.
22. A bitstream is device-specific configuration data — not compiled Verilog, and not executed.
23. Configuration order: power up, clear, sample mode pins, sync, check ID, load frames, CRC, startup.
24. Startup releases DONE, then enables I/O (GTS), then enables register and RAM writes (GWE).
25. Block RAM reads are **synchronous** — one cycle, or two with the output register.
26. Read-during-write behaviour is a **configured mode**: new data, old data, or hold.
27. Distributed RAM lives in LUTs, is often asynchronous-read, and competes with your logic.
28. An asynchronous FIFO is the standard, safe way to move a stream between clock domains.
29. Gray-coded pointers change one bit per step, so a mid-change sample is still a valid value.
30. FIFO flags built from synchronised pointers are pessimistic — which is the safe direction.
31. A DSP block is a pre-adder, a multiplier, an accumulator and pipeline registers.
32. DSP widths are family-specific. `AMD` DSP48E1 is 25x18 with a 48-bit accumulator.
33. A DSP only runs at its rated speed with its **pipeline registers enabled**.
34. Clocks belong on dedicated, length-matched networks — never on general routing.
35. Global clock buffers are a countable, scarce resource; running out silently ruins timing.
36. A clock manager cannot improve on the reference's frequency accuracy; it must LOCK first.
37. Skew is spatial and static; jitter is temporal and random. Skew can help setup; jitter never helps.
38. Setup fails when the path is too slow, and slowing the clock fixes it.
39. Hold fails when the path is too fast, and slowing the clock does **nothing**.
40. `slack = required - arrival`. WNS is the worst path; TNS counts how many are bad.
41. The critical path alone sets fmax. Optimising any other path changes nothing.
42. STA is vector-free: it checks every path, including ones no testbench exercises.
43. Unconstrained paths are not analysed at all — always check the unconstrained-endpoint count.
44. A false path stops the tool checking; it does not make anything faster.

45. Pipelining raises throughput and fmax, and raises latency. It cannot help a feedback loop.
46. High fanout is fixed by replication, pipelining, or having fewer loads.
47. Metastability cannot be eliminated — only made improbable, one bit at a time.
48. Assert reset asynchronously; release it **synchronously**, one synchroniser per domain.
49. Hard IP is fixed in number and position; soft IP costs fabric and timing budget.
50. Passing simulation proves the logic against your stimulus, and nothing about timing.

// §60 RAPID REVISION — TWENTY MINUTES

```
FPGA configurable hardware fabric; configuration bits, not instructions
BLOCKS LUT + FF + MUX + carry | routing | BRAM + DSP | clocking + I/O | config
LUT 2^N config bits, mux tree, ANY N-input function, delay independent of function
FF D, Q, clk, CE (dedicated pin), set/reset; plentiful, nearly free
CARRY dedicated bit-to-bit path; adders/counters/comparators; must stay in one column
ROUTING connection box = pin to channel; switch box = channel to channel
every hop is a transistor; usually the LARGEST delay term
BRAM kilobits, hard block, SYNCHRONOUS read, write-mode matters, dual port
DIST RAM LUTs as memory, often async read, competes with logic
DSP pre-adder + multiplier + accumulator + PIPELINE REGISTERS + cascade
CLOCK clock-capable pin → manager (must LOCK) → global buffer → dedicated tree
never a fabric-generated clock; use a clock ENABLE
I/O buffer + optional FF; standard, drive, slew; BANKS share a supply voltage
CONFIG SRAM (volatile) | flash (live at power-up) | antifuse (one-time)
BITSTREAM every config bit for THIS device; loaded once; not code, not portable
STARTUP DONE released → GTS negated (I/O on) → GWE asserted (FFs may change) → EOS
SYNTHESIS RTL → Boolean network; folds constants, shares, retimes, DELETES dead logic
MAPPING Boolean → THIS device's LUTs/FFs/DSPs/BRAMs. First device-specific step.
PLACEMENT which physical site. Decides how far every signal must travel.
ROUTING which tracks and switches. Cannot rescue a bad placement.
STA vector-free; every constrained path; Tclk ≥ tcq+tlogic+troute+tsu+unc-skew
SLACK required - arrival. WNS = worst path. TNS = how many are bad.
CRIT PATH worst slack; sets fmax alone; check its LOGIC vs ROUTING split
SETUP too SLOW; depends on Tclk; slow the clock, pipeline, reduce depth/fanout
HOLD too FAST; Tclk NOT in the equation; add delay, or reduce skew
PIPELINE throughput UP, fmax UP, latency UP; useless in a feedback loop
FANOUT replicate, pipeline, or reduce the number of loads
CDC 1 bit → 2-FF | pulse → toggle | word → handshake/Gray | stream → async FIFO
METASTAB never eliminated; reduced to an MTBF; ONE BIT at a time
RESET assert async, RELEASE SYNC; one synchroniser per domain; do not over-reset
VS ASIC ASIC faster/smaller/lower power, huge NRE; FPGA reconfigurable, no NRE
VS CPU CPU = one datapath over TIME; FPGA = many datapaths in SPACE
```

// §61 LEARNING ROADMAP

Level	What you master	You can then answer
1 — Basics	digital logic, and what an FPGA is and is not (§1–3)	"Why is this not a CPU?"
2 — The fabric	LUT, flip-flop, mux, carry chain, routing (§5–9)	"What does my Boolean expression become?"
3 — Hard blocks	BRAM, DSP, clocking, I/O (§13–21)	"Why is my memory a cycle late, and my multiplier slow?"
4 — Configuration	configuration memory, bitstream, startup (§10–12)	"What actually happens at power-up?"
5 — RTL to hardware	synthesis, mapping, placement, routing (§26–30)	"Why did my design get bigger when I changed one line?"
6 — Timing	STA, setup, hold, slack, critical path, pipelining (§31–34)	"Why does it fail at 100 MHz and pass at 50?"
7 — Advanced	CDC, metastability, reset, optimisation, SoC, high-speed I/O (§38–44, §22–25)	"Why does it fail once a week, and only on that board?"

The one thing to carry away. Never memorise an FPGA term on its own. For every term, be able to say the whole chain:

TERM → PURPOSE → INTERNAL STRUCTURE → SIGNAL FLOW → THE RTL THAT PRODUCES IT → PHYSICAL EFFECT → TIMING CONSEQUENCE.

"A LUT is a lookup table" is a definition you can forget. "A LUT holds `2^N` configuration bits read out by a mux tree that my logic inputs address, which is why any 6-input function costs the same delay, which is why I should reduce logic *depth* rather than Boolean complexity" is understanding you can use — and it is the difference between reciting an FPGA glossary and being able to design with one.

// §62 WHERE THE FAMILY-SPECIFIC NUMBERS CAME FROM

Every vendor-chipped claim in this sheet was taken from the manufacturer's own architecture documentation rather than from secondary summaries. If you need a number for a device you are actually using, go to the equivalent document for **that** family — the values differ, and that is the point.

Claim in this sheet	Primary source
Slice = 4x LUT6 + 8 storage elements; CLB = 2 slices; 256 b distributed RAM and 128 b shift register per CLB; ~2/3 SLICEL; F7AMUX / F7BMUX / F8MUX; LUT delay independent of the function implemented	`AMD` **UG474**, 7 Series FPGAs Configurable Logic Block User Guide
Block RAM 36 Kb, splittable into two 18 Kb; true and simple dual port; WRITE_FIRST / READ_FIRST / NO_CHANGE; optional output register; ECC and FIFO modes	`AMD` **UG473**, 7 Series FPGAs Memory Resources User Guide
8 to 24 clock regions; a region spans 50 CLBs and one 50-pin I/O bank; 32 global clock lines, up to 12 per region; CMT = one MMCM + one PLL; the PLL is a subset of the MMCM; local routing is not recommended for clocks	`AMD` **UG472**, 7 Series FPGAs Clocking Resources User Guide
DSP48E1: 25 x 18 two's-complement multiplier, 48-bit accumulator, pre-adder, SIMD dual-24 / quad-12, ten-function logic unit, pattern detector, dedicated cascade buses	`AMD` **UG479**, 7 Series DSP48E1 Slice User Guide
The eight configuration steps, and the startup phases that release DONE, negate GTS and assert GWE	`AMD` **UG470**, 7 Series FPGAs Configuration User Guide
UltraScale CLB: 8 LUT6 + 16 flip-flops in a single slice; 512 b distributed RAM; SRL32 to a 256-bit shift register; LUT as a 4:1 mux	`AMD` **UG574**, UltraScale Architecture CLB User Guide
ALM: 8-input fracturable LUT, two dedicated adders, four registers; all 6-input functions, two independent 4-input functions, selected 7-input functions; ten ALMs per LAB; M20K and MLAB memory; variable-precision DSP	`INTEL` / Altera device overviews and Adaptive Logic Module chapters (Arria, Stratix, Agilex families)
Registers distributed through the routing to enable aggressive retiming	`INTEL` HyperFlex architecture documentation (Stratix 10, Agilex)
Logic element = fully permutable 4-input LUT + separate DFF + carry-look-ahead carry chain; 18 x 18 MACC math block; LSRAM 20 kbit; uSRAM 64 x 12; configuration cells SEU-immune; non-volatile, live at power-up, no external boot device	`MICROCHIP` **PolarFire FPGA Product Overview** (DS60001657)
Metal-to-metal antifuse, one-time programmable, radiation-tolerant	`MICROCHIP` RTAX family documentation
LUT4-based fabrics: iCE40 logic cell, ECP5 slice of two LUT4 + two FFs within a PFU, Nexus-generation LUT4	`LATTICE` family architecture documentation

How to read a vendor architecture guide quickly. Go straight to the block diagram of the logic block and count what is in it. Then find the memory chapter and note the block size and the read behaviour. Then find the clocking chapter and note how many global buffers there are and how large a clock region is. Those three answers tell you most of what distinguishes one family from another — and they are exactly the three that do not transfer between vendors.