

Unit I : Design with HDL

Assignment No.1

Write any five

- | | | |
|----|---|-----|
| Q1 | Explain any two modelling styles in VHDL. | CO1 |
| Q2 | Write VHDL code and test bench for D flip flop | CO1 |
| Q3 | Explain signal and variable differences in VHDL. | CO1 |
| Q4 | Explain function and procedure with example. | CO1 |
| Q5 | Explain any two concurrent statements in VHDL with syntax. | CO1 |
| Q6 | Design bit shift Register using VHDL programming. | CO1 |
| | Also write Test bench for the same. | |
| Q7 | Explain different types of wait statements used in VHDL, with suitable example. | CO1 |

Unit II : Digital design and Issues

Assignment No.2

Write any five

- | | | |
|----|---|-----|
| Q1 | What is supply and ground bounce? What are remedies to it? | CO2 |
| Q2 | What is the need of clock distribution? Explain techniques of clock distribution. | CO2 |
| Q3 | Draw FSM diagram and write VHDL code for 100 Moore sequence detector. | CO2 |
| Q4 | Draw CPLD architecture in detail. Explain in brief. | CO2 |
| Q5 | Give typical features and specifications of FGPA. | CO2 |
| Q6 | What is clock skew? What are techniques to minimize? | CO2 |
| Q7 | Why should supply and ground bounce be taken care? How are these minimized? | CO2 |
| Q8 | Explain signal integrity issues involved in VLSI design. | CO2 |

Unit III: PLD Architecture and Applications

Assignment No.3

Write any five

- | | | |
|----|---|-----|
| Q1 | Draw CPLD XC9500 series architecture and explain in details. | CO3 |
| Q2 | Describe the PLD design flow. What are the limitations of PLD architecture. | CO3 |
| Q3 | Draw and explain the detail architecture of FPGA. | CO3 |
| Q4 | Compare CPLD and FPGA. What is need of CPLD? | CO3 |
| Q5 | Draw and explain the following for FPGA | CO3 |
| | i. Logic cell | |
| | ii. CLB | |
| | iii. Programmable switch matrix | |
| | iv. I/O block | |
| Q6 | With the help of suitable schematics, compare PROM, PLA and PAL architectures. | CO3 |
| Q7 | Explain with suitable diagram SRAM and anti-fuse programming techniques used in FPGA. | CO3 |
| Q8 | What is a role of CPLD and FPGA in embedded system developments? | CO3 |

Unit IV: Digital CMOS Circuits

Assignment No.4

Write any five

- | | | |
|----|--|-----|
| Q1 | Explain the following: | CO4 |
| | i) Body Effect | |
| | ii) Hot electron Effect | |
| Q2 | Derive the static and dynamic power dissipation in CMOS. | CO4 |
| Q3 | Explain the working of transmission gate and implement a circuit of 2:1 multiplexer using transmission gate. | CO4 |
| Q4 | Draw NAND, NOR, AND, OR, EX-OR gates using CMOS. | |
| Q5 | Draw and Explain RC Delay Model for NMOS and PMOS transistor.
Give example. | CO4 |
| Q6 | Design CMOS logic for $Y = ABC + D$.
Calculate W/L ratio for NMOS and PMOS are needed on chip. | CO4 |
| Q7 | Define scaling and explain any one type of scaling. | CO4 |

Unit V: Application Specific Integrated Circuit

Assignment No.5

Write any five

- | | | |
|----|---|-----|
| Q1 | Which lambda rules are used for CMOS layout? Give its significance | CO5 |
| Q2 | Explain Design issues like antenna effect and electro migration effect. | CO5 |
| Q3 | Draw stick diagram for CMOS inverter, NAND, NOR gate. | CO5 |
| Q4 | Explain in detail micron-based design rules. | CO5 |
| Q5 | Write in lambda rules with diagram. | CO5 |
| Q6 | Explain Parasitic extraction. Explain Crosstalk in detail. | CO5 |
| Q7 | Explain Static Timing Analysis. | CO5 |

Unit VI: Digital CMOS Circuits

Assignment No.6

Write any five

- | | | |
|----|---|-----|
| Q1 | Compare data flow, behavioral and structural modeling. | CO6 |
| Q2 | Write code in VHDL for 4:1 multiplexer in three different modeling types. | CO6 |
| Q3 | Design CMOS logic for $Y = A + BC + DE$. Calculate the area in terms of W. | CO6 |
| Q4 | What are the effects of technology scaling on chip level design? | CO6 |
| Q5 | Describe the following: | CO6 |
| | a. BSDF | |
| | b. JTAG | |
| Q6 | Explain Stuck-at-0 and stuck-at-1 faults. | CO6 |
| Q7 | Write short notes on : | CO6 |
| | a. ASIC | |
| | b. ASIC design flow. | |

VLSI DESIGN

Unit-wise Assignment Question Bank with Model Answers

Branch: Electronics & Telecommunication (E&TC) / Embedded Systems

Year / Semester: B.E. — Semester VII

Subject Code: VLSI Design

Contents

Assignment	Unit / Topic	Coverage
1	Design with HDL	7 questions answered
2	Digital Design and Issues	8 questions answered
3	PLD Architecture and Applications	8 questions answered
4	Digital CMOS Circuits	7 questions answered
5	Application Specific Integrated Circuit	7 questions answered
6	Digital CMOS Circuits	7 questions answered

Assignment 1

Unit 1: Design with HDL | VHDL Modelling, Coding and Testbenches

Q1. Explain any two modelling styles in VHDL.

VHDL (VHSIC Hardware Description Language) is a hardware description language used to model the behaviour and structure of digital electronic systems at multiple levels of abstraction. Unlike a software programming language, statements written outside a process in VHDL execute concurrently (in parallel), which reflects the parallel nature of real hardware. VHDL supports three broad modelling styles - Dataflow, Behavioral and Structural - and a designer may even mix all three within the same architecture (mixed-style modelling). Below, two of these styles are explained in detail.

1. Dataflow Modelling: This style describes a circuit in terms of how data flows between input and output signals, using concurrent signal assignment statements. It expresses the circuit as a set of Boolean/RTL equations rather than as an algorithm, and is therefore closest to a logic/RTL schematic. Every concurrent statement outside a process executes simultaneously and independently, reacting automatically whenever any signal on its right-hand side changes value (this is called being 'sensitive' to that signal).

- Key constructs used: simple concurrent signal assignment ($\leq$), conditional signal assignment (when/else), and selected signal assignment (with/select/when).
- Because there is no explicit process/sensitivity list to write, dataflow code is short, and closely mirrors combinational logic equations such as $Y = A.B + C$.
- Best suited for describing multiplexers, decoders, adders, and other purely combinational blocks where the function can be expressed as a Boolean equation.

```
-- Example: 2-input AND gate, Dataflow style
architecture Dataflow of AND_GATE is
begin
  Y <= A and B; -- concurrent signal assignment,
                -- executes whenever A or B changes
end Dataflow;
```

2. Behavioral Modelling: This style describes only the functional behaviour of the circuit, using sequential statements (if-else, case, for/while loop) written inside a process block, very similar to writing a software algorithm. The internal hardware structure is not specified at all - only 'what' the circuit should do, not 'how' it is built - and it is left to the synthesis tool to infer the actual gates/flip-flops.

- A process is itself a concurrent statement (it runs in parallel with other processes/statements outside it), but the statements written inside the process body execute sequentially, one after another, exactly like software.
- A process is triggered either by a sensitivity list (a list of signals that 'wake up' the process on any change) or by wait statements inside its body.
- Behavioral modelling is powerful and flexible - it is the natural choice for describing Finite State Machines, complex sequential circuits, and algorithms that are difficult to express as pure equations.

```
-- Example: 2-input AND gate, Behavioral style
architecture Behavioral of AND_GATE is
begin
  process(A, B) -- sensitivity list
  begin
    if (A = '1' and B = '1') then
      Y <= '1';
    else
      Y <= '0';
    end if;
  end process;
end Behavioral;
```

(For completeness: the third style, Structural Modelling, describes the circuit purely as an interconnection of previously-designed components/sub-modules using component declarations and port maps - similar to drawing/describing a schematic netlist. It is used for hierarchical, reusable designs where larger systems are built from smaller, already verified blocks.)

Comparative note: Dataflow code is compact and hardware-oriented but becomes unwieldy for complex sequential logic; Behavioral code is easy to write and debug for complex algorithms/FSMs but requires care (proper sensitivity list, avoiding unintended latches) to synthesize into efficient hardware. In practice, real designs use a mixed style: dataflow/structural for datapaths and behavioral for control logic (FSMs).

Q2. Write VHDL code and test bench for D flip flop.

A D flip-flop (Data / Delay flip-flop) is the most fundamental sequential storage element used in synchronous digital design. On every active (rising, in this example) edge of the clock, it copies (samples) the value present at its D input to its Q output, and holds that value until the next active clock edge. It typically also has an asynchronous reset (RST) input that forces Q to '0' immediately, independent of the clock, which is essential for bringing a circuit to a known state at power-up.

Truth / operation table:

RST	CLK edge	D	Q (next state)
1	X (don't care)	X	0 (asynchronous reset)
0	rising edge	0	0
0	rising edge	1	1
0	no edge	X	Q (previous value held)

Design entity (Behavioral style):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_FF is
  Port ( D, CLK, RST : in STD_LOGIC;
        Q           : out STD_LOGIC);
end D_FF;

architecture Behavioral of D_FF is
begin
  process(CLK, RST) -- sensitivity list has both
  begin -- CLK and RST (async reset)
    if (RST = '1') then
      Q <= '0'; -- asynchronous, has priority
    elsif (rising_edge(CLK)) then
      Q <= D; -- synchronous data capture
    end if;
  end process;
end Behavioral;
```

Note: rising_edge(CLK) is equivalent to writing 'CLK'event and CLK = 1'. Placing RST as a separate 'if' branch (evaluated first) before the elsif for CLK makes it an asynchronous reset - it can override Q at any time, not only on a clock edge.

Test bench (self-checking stimulus generator):

```
entity D_FF_TB is end D_FF_TB;
architecture sim of D_FF_TB is
  component D_FF
    Port(D,CLK,RST: in STD_LOGIC; Q: out STD_LOGIC);
  end component;
  signal D,CLK,RST,Q : STD_LOGIC := '0';
```

```

begin
  UUT: D_FF port map(D=>D, CLK=>CLK, RST=>RST, Q=>Q);

  -- Clock generation process: 20 ns period (10 ns high/low)
  CLK <= not CLK after 10 ns;

  -- Stimulus process
  RST <= '1', '0' after 15 ns;
  D  <= '1' after 25 ns, '0' after 65 ns, '1' after 95 ns;
end sim;

```

Explanation of the test bench: A test bench is itself a VHDL module with no ports - it instantiates the Unit Under Test (UUT) as a component and drives its inputs with a planned sequence of values (stimulus) while the clock toggles continuously in the background. Here, RST is first asserted (1) and then released (0) at 15 ns to initialize the flip-flop; D is then changed at various times, and by observing Q in the simulation waveform we verify that Q always follows D at the rising edge and instantly goes to 0 whenever RST=1, confirming correct functional behaviour before proceeding to synthesis and hardware implementation.

Q3. Explain signal and variable differences in VHDL.

VHDL provides two different kinds of objects for holding and communicating values inside a design: signals and variables. Although at first glance they resemble each other, they behave very differently and are used for different purposes. Understanding this difference is essential, since using the wrong one is one of the most common sources of simulation-versus-synthesis mismatch for VHDL beginners.

Parameter	Signal	Variable
Declaration	Declared in the architecture body (outside any process), or in entity/package	Declared inside a process, function or procedure only
Scope	Global - visible throughout the whole architecture and to all processes in it	Local - visible only within the process/subprogram where declared
Assignment operator	<= (scheduled / non-blocking assignment)	:= (immediate / blocking assignment)
Update timing	New value is scheduled and takes effect only after a delta-cycle delay (or after the process suspends) - not visible to the very next statement	New value is updated immediately; visible to the next statement in the same process
What it represents in hardware	Models an actual physical wire/net or a register connecting different parts of the design	Models a temporary variable used purely for internal computation; not directly a hardware wire
Where usable	Can be used in both concurrent statements (outside a process) and sequential statements (inside a process)	Can be used only inside sequential code - i.e. only inside a process, function, or procedure
Retains value between calls	Yes, as long as it is not re-driven	Local variables inside a process retain value between activations of that same process (since it is not re-declared); shared variables (rare) persist globally

Illustrative example showing the difference: Consider the following process, executed with initial values A='0', B='0':

```

process(clk)
  variable temp : STD_LOGIC := '0';
begin
  if rising_edge(clk) then
    temp := A;  -- variable: new value used immediately below
    B  <= temp; -- B gets OLD value of A from before this edge
  end if;
end process;

```

Because ':' updates 'temp' immediately, the statement 'B <= temp' uses the just-updated value of temp in the same execution of the process. If a signal had been used in place of 'temp', its updated value would only become visible after the process suspends (i.e. in the next simulation delta cycle), which can introduce a one-cycle behavioural difference - this is a classic pitfall when converting behavioral pseudo-code into synthesizable VHDL, and is why signals should be used to model actual registers, while variables should be used only for intermediate scratch calculations within a single clock cycle.

Q4. Explain function and procedure with example.

VHDL allows a designer to package a commonly-used sequence of computations into a reusable block called a subprogram. VHDL provides two kinds of subprograms: functions and procedures. Both help make code more modular, readable and reusable, and both can be declared either locally (inside an architecture/process) or globally (inside a package, so they can be shared across multiple design units).

Function: A subprogram that always computes and returns exactly one value, and is called directly from within an expression (wherever a value of that type is expected), similar to a mathematical function. Every parameter of a function is, by default, an 'in' (input-only) parameter - a function cannot modify its arguments, and it cannot contain a wait statement (this restriction exists because functions must execute instantaneously/in zero simulation time when called).

```
function max_val(a, b : integer) return integer is
begin
  if a > b then
    return a;
  else
    return b;
  end if;
end function;
```

```
-- Usage - called directly inside an expression:
Y <= max_val(A, B);
```

Procedure: A subprogram that may take parameters of mode 'in', 'out' or 'inout', and can therefore return zero, one, or several values through its out/inout parameters. Unlike a function, a procedure is called as a complete, independent statement (not embedded inside an expression), and - if declared inside a process - it may even contain wait statements, since it can consume simulation time.

```
procedure add_val(a, b : in integer; sum : out integer) is
begin
  sum := a + b;
end procedure;
```

```
-- Usage - called as a standalone statement:
add_val(A, B, RESULT);
```

Key differences summarized:

Aspect	Function	Procedure
Returns	Exactly one value (via 'return')	Zero, one, or many values (via out/inout params)
Parameter modes	Only 'in' (default)	in, out, inout
Called as	Part of an expression	A standalone statement
Can contain wait?	No	Yes (if declared in a process)

Both functions and procedures greatly improve code readability and maintainability - a computation written once as a subprogram can be reused at multiple places in the design or test bench without duplicating code, reducing the chance of coding errors.

Q5. Explain any two concurrent statements in VHDL with syntax.

Concurrent statements are statements written directly inside the architecture body, outside any process. All concurrent statements in a VHDL architecture execute in parallel with one another, and each one automatically re-evaluates/re-executes whenever any signal appearing on its right-hand side (its 'sensitivity') changes value. This concurrency is what allows VHDL to correctly model the inherently parallel nature of real digital hardware, as opposed to the sequential execution of a normal software program.

1. Conditional Signal Assignment (when / else): Assigns one of several possible values to a target signal, based on a series of Boolean conditions that are evaluated strictly in the order written (top to bottom) - i.e. it has a built-in priority, exactly like a chain of if-elsif-else statements, but written as a single concurrent statement.

```
Y <= A when Sel = "00" else
    B when Sel = "01" else
    C when Sel = "10" else
    D;           -- default / final else
```

- Because the conditions are checked in priority order, this construct naturally synthesizes to a priority-encoded multiplexer structure.
- Very useful for describing multiplexers, priority encoders, and any logic that depends on multiple independent conditions rather than one selector value.

2. Selected Signal Assignment (with / select / when): Assigns a value to the target signal by comparing a single selector expression against a list of possible values - similar in spirit to a 'case' statement, but written as a concurrent statement. Unlike the conditional assignment, all choices in a selected assignment must be mutually exclusive and, together, must cover every possible value of the selector (the 'others' choice is used to guarantee this).

```
with Sel select
    Y <= A when "00",
        B when "01",
        C when "10",
        D when others; -- covers remaining/undefined codes
```

- Because all branches are checked against one selector with no priority ordering, this construct maps very efficiently to a balanced multiplexer, and is the natural choice when a single select line clearly picks among several data paths.

Other concurrent statements (for completeness): Simple (unconditional) concurrent signal assignment, the Process statement (itself a single concurrent statement even though it contains sequential code inside), Component Instantiation (structural connections), Concurrent Assertion statements (used for simulation-time checks), Generate statements (for repetitive/parameterized structures), and Block statements (for grouping related concurrent statements).

Q6. Design bit shift Register using VHDL programming. Also write Test bench for the same.

A shift register is a sequential circuit built from a chain of D flip-flops connected so that data moves one position ('shifts') to the left or right on every active clock edge. Shift registers are widely used for serial-to-parallel and parallel-to-serial data conversion, in UART/communication interfaces, in delay lines, and inside arithmetic units (e.g. for multiplication/division by shifting).

Types of shift registers: Serial-In Serial-Out (SISO), Serial-In Parallel-Out (SIPO), Parallel-In Serial-Out (PISO), and Parallel-In Parallel-Out (PIPO). The example below implements a 4-bit SISO (serial-in, serial-out) left-shift register, where the internal 4-bit content is also available (SIPO-style) for illustration.

4-bit Serial-In Serial-Out (SISO) Shift Register

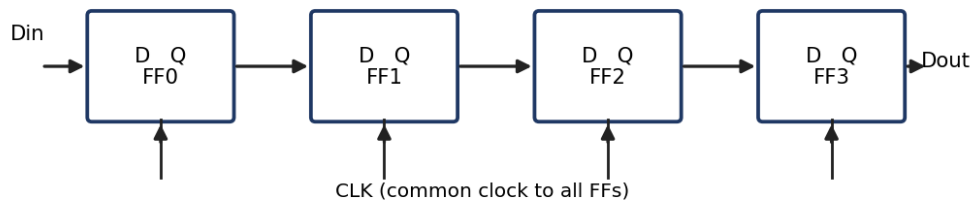


Fig: 4-bit shift register - block diagram (chain of D flip-flops, common clock)

Working: at every rising edge of CLK, the flip-flop chain shifts its stored content one position to the left; the new serial input bit (Din) enters at FF0, while the bit that was in FF3 is shifted out as Dout and lost. An asynchronous RST clears the whole register to 0000, which is essential to bring the register to a known state before shifting begins.

VHDL code (Behavioral style, using a 4-bit signal vector):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SHIFT_REG is
  Port ( CLK, RST, Din : in STD_LOGIC;
        Dout      : out STD_LOGIC);
end SHIFT_REG;

architecture Behavioral of SHIFT_REG is
  signal temp : STD_LOGIC_VECTOR(3 downto 0);
begin
  process(CLK, RST)
  begin
    if RST = '1' then
      temp <= "0000"; -- clear register
    elsif rising_edge(CLK) then
      temp <= temp(2 downto 0) & Din; -- shift left, load Din
    end if;
  end process;
  Dout <= temp(3); -- MSB is the serial output
end Behavioral;
```

The expression 'temp(2 downto 0) & Din' concatenates the lower 3 bits of the current register with the new input bit, and re-assigns this 4-bit value back to temp - this single line, executed on every clock edge, is exactly what performs the 'shift and load' operation without needing four separate flip-flop instantiations.

Test bench:

```
entity SHIFT_REG_TB is end SHIFT_REG_TB;
architecture sim of SHIFT_REG_TB is
  signal CLK, RST, Din, Dout : STD_LOGIC := '0';
begin
  UUT: entity work.SHIFT_REG port map(CLK,RST,Din,Dout);
  CLK <= not CLK after 10 ns; -- 20 ns period clock
  RST <= '1','0' after 12 ns; -- clear, then release
  Din <= '1','0' after 30 ns,
        '1' after 50 ns,
        '0' after 70 ns; -- serial bit pattern
end sim;
```

Simulation verification: after RST is released, the sequence of bits applied at Din (1,0,1,0,...) should appear, delayed by 4 clock cycles, at Dout - confirming the register correctly shifts data through all four flip-flop stages.

Q7. Explain different types of wait statements used in VHDL, with suitable example.

The wait statement is a sequential statement, used only inside a process (or a subprogram called from a process), that suspends the execution of that process until a specified condition becomes true. A very important rule in VHDL is that a process may contain either a sensitivity list or wait statement(s) inside its body, but never both together - if a process has an explicit sensitivity list, it is treated as if it implicitly contains a single 'wait on <sensitivity-list-signals>,' at the very end of the process body. VHDL provides four forms of the wait statement, each suited to a different situation.

1. wait; (unconditional wait): Suspends the process indefinitely - forever, or until the simulation ends. It is typically used in a process that should execute only once, for example, to perform one-time initialization of signals at the very start of simulation.

```
process
begin
  RST <= '1';
  wait for 20 ns;
  RST <= '0';
  wait;          -- process never resumes again
end process;
```

2. wait on (signal list); (sensitivity wait): Suspends the process until any one of the listed signals changes value - this is functionally identical to writing those same signals in the process's sensitivity list.

```
wait on A, B;    -- resumes when A or B changes
```

3. wait until (condition); (condition wait): Suspends the process until the given Boolean condition evaluates to true. This form is extremely common for describing synchronous (clocked) logic, since it lets a process wait specifically for a rising or falling clock edge.

```
wait until CLK'event and CLK = '1'; -- wait for rising edge
-- equivalent to: wait until rising_edge(CLK);
```

4. wait for (time expression); (timeout wait): Suspends the process for a fixed, specified amount of simulation time. This form does not depend on any signal changing - it is a pure time delay, and is used almost exclusively in test benches to generate stimulus waveforms and clocks, since real synthesizable hardware cannot have a fixed physical time delay of this kind.

```
process
begin
  CLK <= '0';
  wait for 10 ns; -- clock low for 10 ns
  CLK <= '1';
  wait for 10 ns; -- clock high for 10 ns
end process;    -- process loops back automatically
```

Important synthesis note: Only 'wait until' (and an equivalent sensitivity list with a clock edge check) is generally synthesizable into real hardware, since it directly maps to a clocked flip-flop; 'wait for' (time delay) is not synthesizable and is used purely for simulation/test-bench purposes, since real silicon has no concept of an absolute, technology-independent time delay.

Assignment 2

Unit II: Digital Design and Issues | Clocking, Signal Integrity and FSM Design

Q1. What is supply and ground bounce? What are remedies to it?

In an ideal circuit, the VDD (supply) and GND (ground) rails are treated as perfectly constant, zero-impedance voltage references. In a real chip, however, every bond wire, package lead and PCB trace connecting the die to the external supply has a small parasitic (self and mutual) inductance L . When many output drivers or internal gates switch simultaneously, a large, rapidly changing current $i(t)$ flows through this parasitic inductance. By the fundamental inductor voltage relation $V = L(di/dt)$, even a small inductance combined with a fast di/dt produces a significant voltage spike.

Ground bounce: When many outputs switch from '1' to '0' simultaneously, a large discharge current flows to ground through the ground-lead inductance, momentarily pulling the on-chip ground reference above the true external ground level. Because internal gates measure logic levels relative to this on-chip ground, the effective ground 'bounces' upward.

Supply (VDD) bounce: Similarly, when many outputs switch from '0' to '1' simultaneously, a large charging current is drawn from VDD through the supply-lead inductance, momentarily pulling the on-chip VDD level below its nominal value (sometimes also called 'VDD droop' or 'IR-drop' when combined with resistive effects).

This phenomenon is also called Simultaneous Switching Noise (SSN) or Delta-I noise, since it is proportional to the number of outputs switching together (ΔI) and the rate of current change (di/dt).

Effects if not controlled:

- False switching/glitches on quiet (non-switching) signal lines that share the same power/ground rail.
- Reduced noise margin - a logic '0' may appear as an elevated voltage, or a logic '1' as a reduced voltage, risking mis-interpretation by the receiving gate.
- False clocking of flip-flops if the bounce couples onto the clock network, or unwanted double-triggering.
- Increased jitter and reduced maximum operating frequency of the whole chip.

Remedies:

- Use multiple, well-distributed VDD and GND pins/pads rather than a single pin each - connecting several parallel bond wires effectively divides and reduces the total effective inductance seen by the switching current.
- Add sufficient on-chip and on-board decoupling (bypass) capacitors placed as physically close as possible to the switching circuitry - these capacitors act as a local charge reservoir, supplying/absorbing the fast transient current so it does not have to travel through the long, inductive package/PCB path.
- Choose a low-inductance package (e.g. flip-chip/BGA packages with short, wide interconnects) instead of long wire-bond packages.
- Stagger or slew-rate-control simultaneously switching outputs (SSO) so that not all outputs transition at exactly the same instant, spreading the current demand over time.
- Use separate, isolated power/ground planes (or domains) for noisy digital I/O drivers versus sensitive internal core/analog circuitry, so bounce generated by I/O switching does not directly couple into the core supply.
- Careful PCB layout - wide, short power/ground traces and dedicated ground planes minimize the parasitic inductance in the external path as well.

Q2. What is the need of clock distribution? Explain techniques of clock distribution.

Need of clock distribution: A synchronous digital system contains thousands to millions of flip-flops/latches, and correct operation depends on every one of them sampling data at the correct, common clock edge. The clock network is the circuitry that carries the clock signal from a single source (an oscillator/PLL) to every sequential element on the chip.

- If the clock reaches different flip-flops at different times, this difference (called clock skew) can eat into the available timing margin for both setup and hold checks, and in the worst case cause a downstream flip-flop to sample data before it is valid (setup violation) or capture the wrong, newly-changing value (hold violation).
- The clock net typically has the largest fan-out and switches every single cycle, so it also dominates dynamic power consumption (often 30-40% of total chip dynamic power) - so clock distribution must be designed not just for correctness but also for power efficiency.
- Good clock distribution design directly determines the maximum achievable operating frequency of the chip, since skew and clock insertion delay both reduce the effective time budget available for logic between two flip-flops.

Techniques of clock distribution:

(a) Clock Tree (buffered tree): The clock is distributed through a hierarchical tree of buffers/inverters, starting from the root (clock source) and branching out to leaf buffers that finally drive individual flip-flops. Buffer sizes and wire lengths at each level are chosen (balanced) so that every leaf-to-root path has nearly equal total delay, minimizing skew. This is the most common technique used by modern EDA tools during Clock Tree Synthesis (CTS).

(b) H-Tree distribution: A special, highly symmetric case of a buffered tree, shaped like a recursively repeated letter 'H'. Because the wire length (and hence RC delay) from the root to every leaf is made exactly equal by construction/geometry, an H-tree achieves extremely low clock skew with a relatively simple, regular layout.

Clock Distribution: Symmetric H-Tree Network

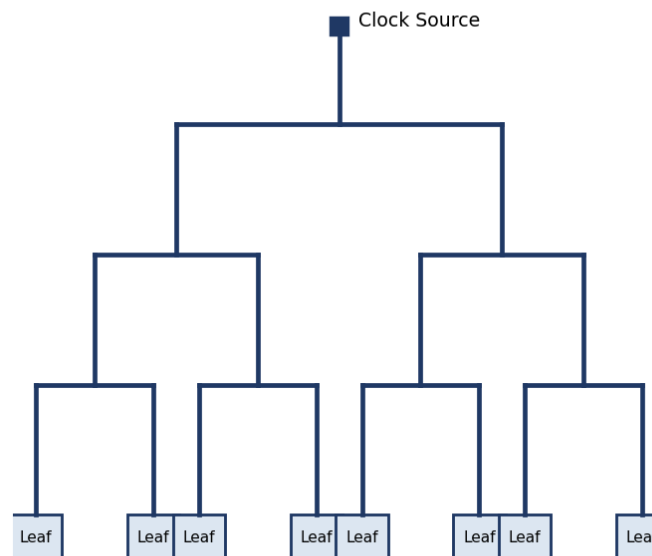


Fig: Symmetric H-tree clock distribution network

(c) Clock Mesh / Grid: Clock wires running in both horizontal and vertical directions are deliberately shorted together at many intersection points, forming a mesh. Because many redundant paths connect the source to every point on the mesh, any local skew or resistance variation is averaged out, making the mesh very robust to process variation and load imbalance - at the cost of significantly higher dynamic power (many extra wires are being driven) and larger routing area.

(d) Clock Spine: A simpler trunk-and-branch structure (a single main 'spine' with shorter branches tapping off it), used for small-to-medium sized blocks where a full H-tree/mesh is not justified by the design size.

(e) Useful-skew (skew scheduling): A deliberate, controlled skew is intentionally introduced between certain flip-flop pairs (rather than forcing zero skew everywhere) to borrow timing slack from a non-critical path and give it to a critical path, improving the achievable clock frequency.

In practice, large modern SoCs commonly combine a global H-tree/balanced tree at the chip level with local clock meshes within critical high-performance blocks (such as the CPU core), to get both low skew and controlled power.

Q3. Draw FSM diagram and write VHDL code for 100 Moore sequence detector.

A Finite State Machine (FSM) is a sequential circuit whose output depends on both its current input and its internal 'state' (memory of past inputs), implemented using flip-flops for state storage and combinational logic for the next-state and output functions. FSMs are broadly classified as Moore machines (output depends only on the present state) or Mealy machines (output depends on both present state and current input). A sequence detector is a classic FSM application that outputs '1' exactly when a specified bit pattern has been received on a serial input line.

Problem: Design a Moore-type sequence detector that outputs $Z=1$ for one clock cycle whenever the pattern "100" has just been received on serial input X , allowing overlapping occurrences (e.g. for the input stream ...1001001..., overlap is permitted so a new "100" can start reusing the trailing bits of the previous match).

State assignment (state = summary of the longest matched prefix of the target pattern seen so far):

State	Meaning (prefix matched)	Moore output Z
S0	No part of pattern matched (reset state)	0
S1	Last bit(s) received end in "1"	0
S2	Last bits received end in "10"	0
S3	Last bits received end in "100" - pattern found!	1

Moore FSM: "100" Sequence Detector (overlap allowed)
S0=reset, S1=seen "1", S2=seen "10", S3=seen "100" ($Z=1$)

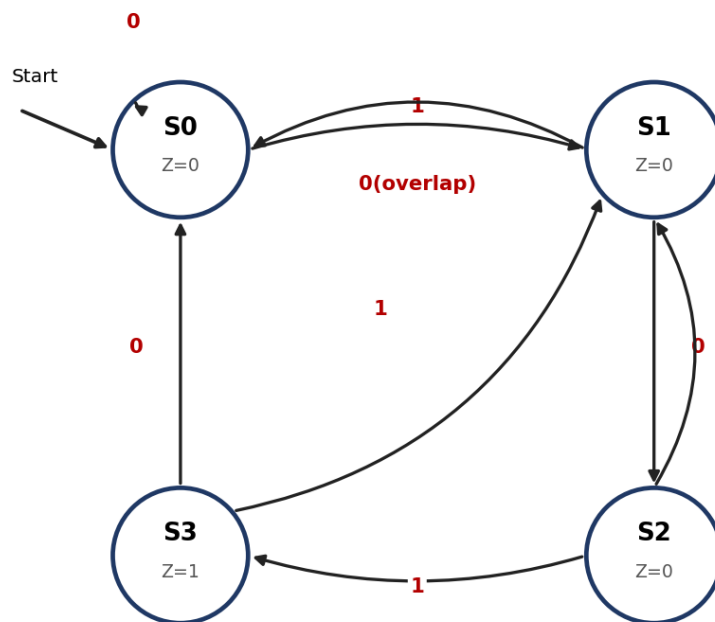


Fig: Moore FSM state diagram for "100" sequence detector

State transition logic (derivation): From S0: input 0 keeps no progress (stay S0); input 1 starts a match (go to S1). From S1 (last bit '1'): input 1 stays at S1 (still ends in '1'); input 0 progresses the match (go to S2). From S2 (ends in '10'): input 0 completes the pattern (go to S3, Z becomes 1); input 1 breaks the '0,0' requirement but the new '1' itself restarts a match, so go back to S1 (this is what allows overlap). From S3 (pattern just found): input 0 means the tail is now '00', which matches no prefix of "100", so go back to S0; input 1 restarts a match, so go to S1 (again enabling overlap).

```

entity SEQ_DET is
  Port(CLK,RST,X : in STD_LOGIC; Z : out STD_LOGIC);
end SEQ_DET;
architecture Moore of SEQ_DET is
  type state_t is (S0,S1,S2,S3);
  signal state : state_t := S0;
begin
  -- Next-state process (synchronous)
  process(CLK,RST)
  begin
    if RST='1' then state <= S0;
    elsif rising_edge(CLK) then
      case state is
        when S0 => if X='1' then state<=S1; else state<=S0; end if;
        when S1 => if X='0' then state<=S2; else state<=S1; end if;
        when S2 => if X='0' then state<=S3; else state<=S1; end if;
        when S3 => if X='0' then state<=S0; else state<=S1; end if;
      end case;
    end if;
  end process;
  -- Moore output: depends only on present state
  Z <= '1' when state = S3 else '0';
end Moore;

```

Because Z is assigned purely as a function of 'state' (and not of X), this is a true Moore machine - the output changes only on a clock edge, synchronized with the state register, giving a glitch-free output compared to a Mealy implementation of the same detector.

Q4. Draw CPLD architecture in detail. Explain in brief.

A Complex Programmable Logic Device (CPLD) is a programmable logic device built by combining several smaller PAL/PLA-like blocks (called Function Blocks or Logic Array Blocks) on a single chip, connected together through a central, fully programmable interconnect matrix, with I/O blocks around the periphery for external connections. The XC9500 family from Xilinx is a well-known industrial example of this architecture.

XC9500 CPLD Architecture (4 Function Blocks + FastCONNECT)

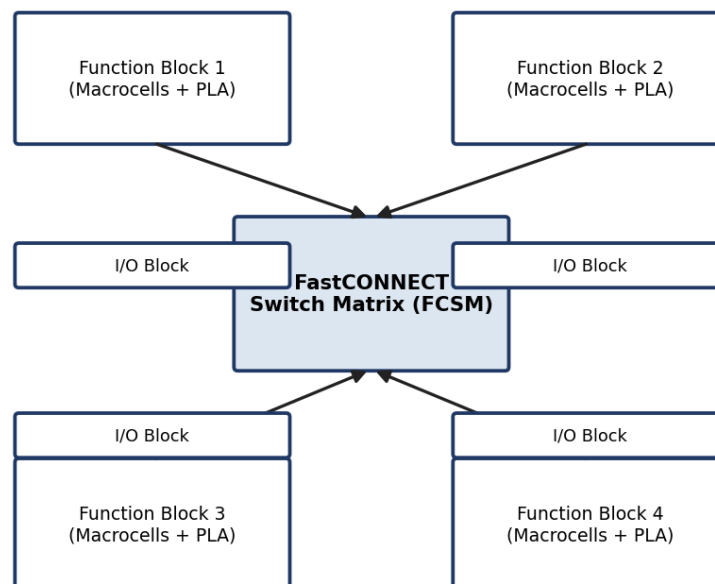


Fig: XC9500-style CPLD architecture

Major building blocks:

- Function Block (FB): the core logic unit of a CPLD. Internally it contains a large programmable AND array feeding a set of fixed OR gates (i.e. a PAL-like sum-of-products structure), followed by a bank of macrocells (typically 16-18 per FB).
- Macrocell: each macrocell contains the OR-plane logic plus a configurable flip-flop (which can be set up as D-type, T-type, or bypassed for pure combinational output), and local feedback/control logic - giving each output the choice of combinational or registered operation.
- FastCONNECT / Switch Matrix: a fully programmable central interconnect that connects the output of any Function Block to the input of any other Function Block with a fixed, predictable delay, regardless of which specific FBs are communicating - this is what gives CPLDs their characteristic fixed, easy-to-predict pin-to-pin timing.
- I/O Block (IOB): interfaces macrocell signals to the external package pins; configurable as input, output or bidirectional/tri-state, with programmable slew-rate and drive strength, and often supports multiple I/O voltage standards.

Key architectural properties:

- Non-volatile configuration memory (EEPROM or Flash based) - the device retains its programmed logic even with power removed, and is immediately functional the instant power is applied ('instant-on'), unlike SRAM-based FPGAs.
- In-System Programmable (ISP) via a standard JTAG interface, allowing the device to be reprogrammed on the circuit board without removing it from the socket.
- Fixed/predictable routing delay through the central switch matrix makes timing analysis and design closure very straightforward compared to the highly variable, congestion-dependent routing delays of an FPGA.
- Logic capacity is comparatively small (typically a few hundred to a few thousand equivalent gates), so CPLDs are best suited to simple control logic, address decoding, and small state machines rather than large, complex systems.

Q5. Give typical features and specifications of FGPA.

An FPGA (Field Programmable Gate Array) is a reconfigurable semiconductor device containing a large, uncommitted array of programmable logic blocks and routing resources that the user can configure, after manufacture ('in the field'), to implement virtually any digital circuit by loading a configuration bitstream.

Typical features and specifications:

- Logic capacity: ranges from a few thousand Logic Elements/CLBs in small devices, up to several million in the largest high-end devices, usually specified in terms of 'equivalent logic cells' or 'system gates'.
- Look-Up Tables (LUTs): the basic combinational logic element, usually a 4-input or 6-input SRAM-based LUT, capable of implementing any Boolean function of its inputs by simply storing the function's truth table in the SRAM cells.
- Configurable Logic Blocks (CLBs) / Slices: clusters of several LUTs plus flip-flops and local routing/carry logic, forming the basic repeated tile of the fabric.
- Embedded Block RAM (BRAM): dedicated, hard on-chip memory blocks (commonly a few Kbits each, totaling from tens of Kbits to many Mbits per device) used for FIFOs, buffers, and register files without consuming general logic resources.
- DSP slices / hardened multipliers: dedicated hardware multiplier-accumulator blocks, essential for efficient, high-speed signal-processing and arithmetic-heavy designs.
- Programmable I/O: hundreds to over a thousand user I/O pins, each individually configurable for different voltage standards (LVCMOS, LVTTTL, LVDS, etc.), drive strength and slew rate.
- Clock management resources: on-chip PLLs/DLLs (Phase/Delay Locked Loops) for clock generation, frequency synthesis, phase-shifting and de-skewing.

- Configuration: predominantly volatile SRAM-based (loses configuration when power is removed, and must be reloaded from an external configuration PROM, flash or processor at every power-up); some FPGA families instead use Flash or Antifuse-based non-volatile configuration.
- Speed grade / performance: characterized by maximum clock frequency, I/O timing (setup/hold, propagation delay), and typically offered in multiple 'speed grades' for the same device to trade off cost against performance.
- Packaging: available in a wide range of packages (QFP, BGA, CSP, etc.) with different pin counts to match different application sizes.

Q6. What is clock skew? What are techniques to minimize?

Clock skew is defined as the difference in arrival time of the same active clock edge at two different sequential elements (flip-flops) in a synchronous circuit. Ideally, the clock edge should reach every flip-flop at exactly the same instant; in a real chip, differences in wire length, buffer sizing, load capacitance, and process/voltage/temperature (PVT) variation across the die cause the clock edge to arrive at slightly different times at different points.

Skew = $t(\text{CLK at FF2}) - t(\text{CLK at FF1})$, and it can be positive or negative depending on which flip-flop's clock arrives later. Skew directly eats into the timing margin available for setup and hold checks between two communicating flip-flops - positive skew (destination clock arrives later than source clock) helps setup timing but hurts hold timing, and vice-versa for negative skew.

Techniques to minimize clock skew:

- Use a balanced/symmetric clock distribution network (buffered tree or H-tree) so that every leaf-to-source path has an equal number of buffer stages and an equal total wire length/load, ensuring nearly equal delay to every flip-flop.
- Insert appropriately-sized clock buffers/repeaters at each branching level, sized according to the actual fan-out and wire length of that branch, to equalize delay across branches carrying different loads.
- Use a clock mesh/grid structure for the most critical, high-performance blocks, since the many redundant paths in a mesh average out local variations in resistance/capacitance.
- Perform dedicated Clock Tree Synthesis (CTS) during physical design, with explicit skew and latency targets/constraints given to the EDA tool, which automatically inserts and sizes buffers to meet them.
- Route clock nets on dedicated, wider/thicker metal layers with lower resistance, and shield them from coupling noise (crosstalk) that could otherwise add jitter/skew.
- Deliberately use 'useful skew' (skew scheduling) - rather than forcing zero skew everywhere (which may be difficult and unnecessary), intentionally introduce controlled skew to borrow slack from a non-critical path and improve a critical path's timing margin.
- Account for On-Chip Variation (OCV)/derating in timing analysis so that the skew budget realistically reflects manufacturing and environmental variation, not just nominal (typical-case) conditions.

Q7. Why should supply and ground bounce be taken care? How are these minimized?

Supply and ground bounce arise from the same fundamental cause explained earlier - fast, large simultaneous switching currents flowing through the parasitic inductance of package leads, bond wires and PCB traces ($V = L \, di/dt$) - but this question focuses on why controlling them is critical to correct chip operation, and reinforces the minimization techniques.

Why it must be taken care of:

- Functional failure risk: if bounce is severe enough, a quiet ('0') signal line sharing the same ground reference can appear to rise above the logic threshold of a receiving gate, causing a false '1' to be read - or vice versa for VDD bounce affecting a '1' line - directly corrupting logic values.
- Clock corruption: if bounce couples onto the clock distribution network, flip-flops may be falsely triggered (extra, unwanted clock edges) or may miss/mis-time a legitimate edge, both of which break synchronous operation.

- Reduced noise margin: even sub-threshold bounce reduces the effective voltage margin between a valid logic level and the switching threshold, making the circuit far more susceptible to any additional noise source (crosstalk, IR drop, etc.), compounding reliability problems.
- Reliability and long-term degradation: repeated large voltage transients can accelerate device aging effects (e.g. hot-carrier and oxide-stress related degradation) over the operating lifetime of the chip.
- Reduced maximum operating frequency: designers must add extra timing margin to guard against bounce-induced errors, which lowers the achievable clock speed of the whole chip.

How they are minimized (summary of remedies):

- Multiple, well-distributed VDD/GND pins to reduce effective package inductance per switching current path.
- Adequate on-chip and on-board decoupling capacitance placed physically close to switching circuits, acting as a local charge reservoir.
- Low-inductance packaging (e.g. flip-chip/BGA) instead of long wire-bond connections.
- Staggering/slew-limiting simultaneously switching outputs (SSO control) to spread current demand over time.
- Separate, isolated power domains for noisy I/O drivers versus sensitive core/analog circuitry.
- Careful PCB design - short, wide power/ground traces and solid ground planes to minimize external parasitic inductance.

Q8. Explain signal integrity issues involved in VLSI design.

Signal Integrity (SI) refers to the ability of an electrical signal to carry information correctly, at the required speed, without unacceptable distortion, from its source to its destination. As feature sizes shrink and clock frequencies rise in modern VLSI technology, interconnects (wires) increasingly dominate over gate delay, and a range of signal-integrity issues must be carefully analyzed and controlled during design.

Major signal integrity issues:

1. Crosstalk: Unwanted electrical coupling, through parasitic coupling capacitance and mutual inductance, between two adjacent (aggressor and victim) interconnect wires that run close and parallel to each other. Crosstalk causes two distinct problems: a crosstalk glitch, where a switching aggressor net induces a false, temporary voltage bump on a static (non-switching) victim net; and crosstalk-induced delay variation, where a switching victim net's own propagation delay is either sped up or slowed down depending on whether a nearby aggressor switches in the same or opposite direction at nearly the same time.

2. IR drop (resistive voltage drop): As current flows through the finite resistance of the power/ground distribution network, a voltage drop ($I \times R$) occurs, so the actual supply voltage seen by gates far from the power pad is lower than the nominal VDD - this reduces gate drive strength, increases delay and can eventually cause functional failure if excessive.

3. Ground/Supply bounce (Ldi/dt noise): As explained above, transient voltage fluctuation on the power/ground rails due to rapidly changing current through parasitic inductance.

4. Electromigration (EM): At the very high current densities present in modern narrow interconnects, momentum transfer from the flowing electrons gradually displaces metal atoms, eventually causing open-circuit voids or short-circuit hillocks and degrading long-term reliability.

5. Reflection / Ringing: On high-speed interconnects that behave as transmission lines, any impedance mismatch (e.g. at a via, a branch, or the receiving gate) causes part of the signal to reflect back towards the source, producing ringing/overshoot/undershoot on the waveform.

6. Jitter: Cycle-to-cycle timing uncertainty in a clock or data edge, caused by coupled noise (from crosstalk, power supply noise, etc.) onto timing-critical nets, directly eating into the available timing margin.

Because these effects interact and worsen together as technology scales down (smaller wire spacing increases crosstalk, higher current density worsens EM and IR drop, and higher frequencies make transmission-line effects more significant),

modern VLSI design flows include dedicated Signal Integrity analysis steps (parasitic extraction, crosstalk/noise analysis, IR-drop analysis) as a mandatory part of physical design sign-off, alongside timing and power analysis.

Assignment 3

Unit III: PLD Architecture and Applications | CPLD / FPGA Architecture and Programming Technologies

Q1. Draw CPLD XC9500 series architecture and explain in details.

The Xilinx XC9500 family is a classic, industrially significant CPLD architecture, widely studied as the representative example of CPLD design. It is built from multiple identical Function Blocks (FBs) - for example the XC9536 has 2 FBs, the XC95108 has 8 FBs - interconnected through a central FastCONNECT switch matrix, with I/O Blocks (IOBs) around the periphery connecting to package pins.

XC9500 CPLD Architecture (4 Function Blocks + FastCONNECT)

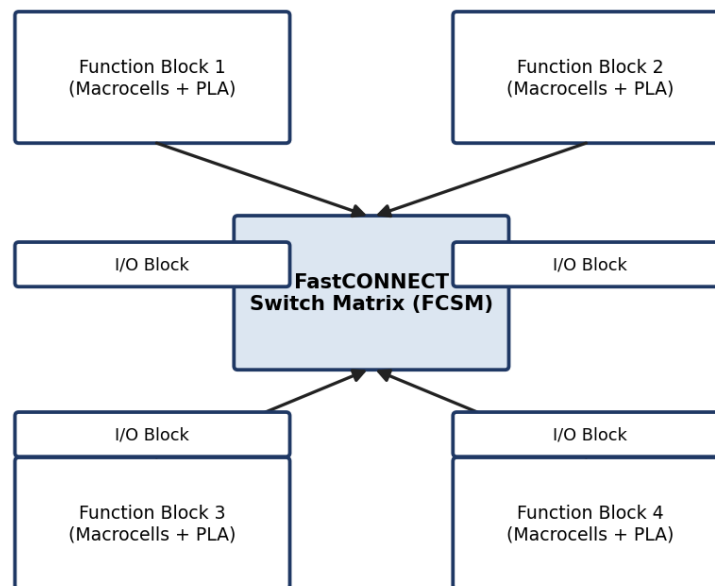


Fig: XC9500 CPLD architecture

(a) Function Block (FB): Each Function Block receives 36 input signals (from the FastCONNECT switch matrix and from feedback of its own macrocell outputs) and feeds them into a large programmable AND array that generates up to 90 product terms. These product terms feed a product-term allocator which flexibly distributes them among 18 macrocells, allowing some macrocells to 'borrow' extra product terms from neighbouring macrocells when a function needs more terms than the default allocation - this is a key XC9500 innovation for efficiently supporting wide logic functions.

(b) Macrocell: Each of the 18 macrocells per FB contains OR-plane logic (summing the allocated product terms), a configurable flip-flop that can be set up as D-type or T-type (or bypassed entirely for pure combinational output), and local control for clock, clock-enable, set and reset - giving the designer full flexibility between combinational-only or registered outputs on a per-macrocell basis.

(c) FastCONNECT Switch Matrix: A fully-populated, non-blocking programmable interconnect that can route the output of any Function Block (or any I/O pin) to the input of any other Function Block, with a fixed, constant delay regardless of the specific source-destination pair. This gives the XC9500 its hallmark predictable, pin-to-pin timing behaviour, which greatly simplifies static timing analysis compared to FPGA routing.

(d) I/O Block (IOB): Provides the electrical interface between the internal macrocell logic and the external package pin - supporting input, output, or bidirectional/tri-state operation, with a programmable slew-rate control (fast/slow edges) and configurable pull-up resistor.

(e) Configuration and programming: The XC9500 uses non-volatile, in-system-programmable (ISP) EEPROM cells to store its configuration, accessed via the standard 4-wire IEEE 1149.1 JTAG interface - the device can therefore be reprogrammed directly on the target circuit board without removal, and it retains its configuration and is instantly functional at power-up, unlike volatile SRAM-based FPGAs.

- Typical applications: address decoding, bus interface glue logic, small state machines, and other 'complex glue logic' functions that are too large for a single simple PAL but do not require the huge capacity of an FPGA.

Q2. Describe the PLD design flow. What are the limitations of PLD architecture.

PLD Design Flow (step-by-step):

- 1. Design entry: The circuit is described either via schematic capture (drawing gates/components) or, more commonly today, using an HDL such as VHDL or Verilog to describe the desired functionality.
- 2. Functional (behavioural) simulation: The HDL/schematic design is simulated with test vectors, purely to verify logical correctness, before committing to a specific device.
- 3. Synthesis: A logic synthesis tool translates the HDL description or schematic into an optimized, technology-mapped netlist expressed in terms of the target PLD's actual macrocells/product terms.
- 4. Fitting (Place & Route for PLDs): A 'fitter' tool maps (assigns) the synthesized logic equations onto the actual physical Function Blocks/macrocells of the chosen device, and configures the switch matrix/interconnect to realize the required connections - analogous to place-and-route in ASIC/FPGA flows, but generally simpler due to the PLD's fixed, regular structure.
- 5. Timing simulation / Static timing verification: After fitting, the actual (now known) propagation delays are back-annotated, and the design is re-verified to ensure it meets the required timing specification.
- 6. Device programming: The final fit result (typically a JEDEC file) is downloaded into the actual PLD device using an in-system programmer or the JTAG/ISP interface, permanently (until reprogrammed) configuring its non-volatile memory cells.
- 7. On-board verification: The programmed device is tested in the actual target system/board to confirm correct operation under real operating conditions.

Limitations of PLD architecture:

- Limited logic capacity: PLDs (SPLDs and CPLDs) have a comparatively small, fixed number of macrocells and product terms per macrocell, making them unsuitable for large, complex digital systems.
- Fixed/rigid interconnect: while this gives predictable timing, it also limits routing flexibility - very wide or complex functions may not fit efficiently, or may require 'borrowing' product terms from neighbouring macrocells, adding extra delay.
- Very limited or no dedicated memory: PLDs generally lack the large embedded block-RAM resources common in FPGAs, restricting their use for memory-intensive applications.
- No dedicated DSP/multiplier hardware, making PLDs inefficient for signal-processing or arithmetic-heavy designs.
- Higher static power consumption per logic function compared to modern SRAM-based FPGA fabrics, in many cases.
- As the design size grows, the wide, brute-force AND-OR (sum-of-products) structure of a PLD becomes area- and power-inefficient compared to the LUT-based structure of an FPGA.

Q3. Draw and explain the detail architecture of FPGA.

A Field Programmable Gate Array (FPGA) is organized in an 'island-style' architecture: a two-dimensional array of Configurable Logic Blocks (CLBs), separated by horizontal and vertical programmable routing channels, with programmable switch matrices at channel intersections, and I/O Blocks surrounding the entire array at the periphery for external connectivity.

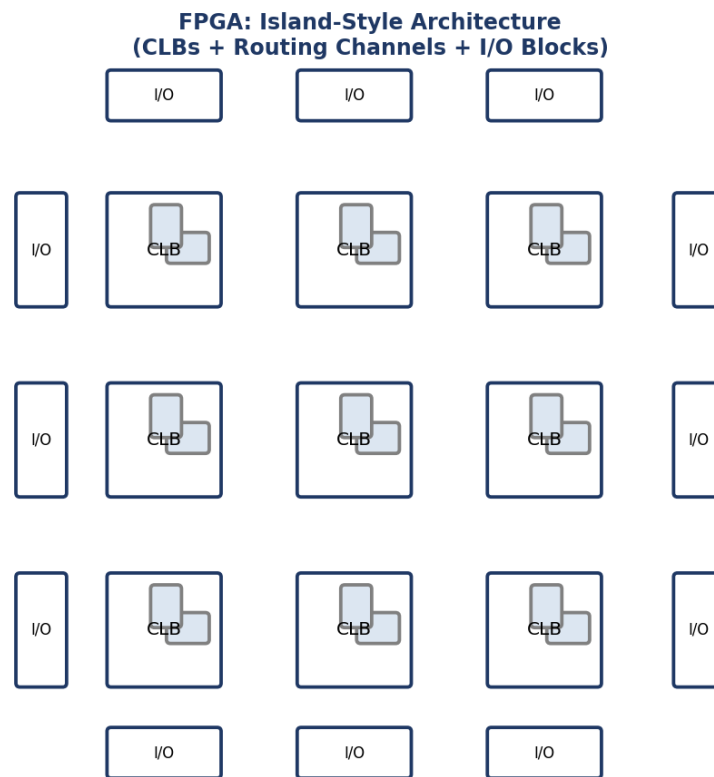


Fig: Island-style FPGA architecture

(a) Configurable Logic Block (CLB): The basic logic-implementing unit of the FPGA, built internally from one or more 'logic cells' or 'slices', each containing a Look-Up Table (LUT), a D flip-flop, and an output multiplexer. The LUT is implemented as a small SRAM memory that stores the truth table of any desired Boolean function of its (typically 4 or 6) inputs, so any combinational function can be realized simply by loading the correct truth-table bits.

(b) Programmable Routing / Interconnect: A rich network of horizontal and vertical wire segments of varying lengths, connected through Programmable Interconnect Points (PIPs), which carry signals between CLBs across the chip. The flexibility (and congestion) of this routing network is what makes FPGA place-and-route more complex, and timing less predictable, compared to a CPLD's fixed switch matrix.

(c) Programmable Switch Matrix: Located at the intersection of horizontal and vertical routing channels; a grid of programmable connection points that allows a signal travelling on one track to switch direction (turn) onto a perpendicular track, enabling routing across the full two-dimensional fabric.

(d) I/O Block (IOB): Positioned around the periphery of the array, connecting the internal fabric to the external package pins, with configurable direction (input/output/bidirectional), voltage standard, drive strength and slew rate.

(e) Additional embedded resources in modern FPGAs: Block RAM (BRAM) modules for on-chip memory, DSP slices with hardware multipliers/accumulators for high-speed arithmetic, and Clock Management Tiles (PLL/DLL) for generating and distributing high-quality clock signals - these hardened blocks are embedded at intervals within the CLB array to avoid consuming general-purpose logic resources for common, performance-critical functions.

Configuration of the entire fabric (LUT contents, routing switch settings, I/O configuration) is typically stored in volatile SRAM cells, loaded from an external non-volatile configuration memory (flash/PROM) at every power-up - this SRAM-based approach is what gives FPGAs their full re-programmability.

Q4. Compare CPLD and FPGA. What is need of CPLD?

Parameter	CPLD	FPGA
Basic architecture	PAL/PLA-based Function Blocks with a central switch matrix	LUT-based CLBs in an island-style array with distributed routing

Parameter	CPLD	FPGA
Logic capacity	Small - typically a few hundred to a few thousand equivalent gates	Large - from thousands up to several million equivalent gates
Configuration memory	Non-volatile (EEPROM/Flash) - retains configuration without power	Predominantly volatile SRAM - needs external boot PROM/reload at power-up
Timing behaviour	Fixed, predictable pin-to-pin delay (switch-matrix routing)	Variable, congestion/routing-dependent delay
Power-up behaviour	Instant-on - functional the moment power is applied	Requires a configuration/boot sequence before becoming functional
On-chip memory / DSP	Little to none	Dedicated Block RAM and DSP slices available
Static power	Generally higher per logic function	Generally lower per logic function in modern processes
Cost & tool complexity for small designs	Lower cost, simpler design flow	Higher cost, more complex place-and-route tools
Typical application	Glue logic, address decoding, small FSMs	Complex SoC, DSP pipelines, large digital systems

Need of CPLD: Even though FPGAs offer far greater logic capacity, CPLDs remain essential for applications that specifically require instant-on operation (functional immediately at power-up, with no configuration delay), non-volatile, secure configuration storage, and simple, guaranteed-predictable pin-to-pin timing - properties that make CPLDs ideal for small, timing-critical control/glue-logic tasks such as power-sequencing logic, address decoding, boot-configuration logic, and simple state machines, often working alongside a larger FPGA or processor in the same system, where using a full FPGA would be unnecessarily costly, power-hungry and would introduce an undesirable configuration delay.

Q5. Draw and explain the following for FPGA: (i) Logic cell (ii) CLB (iii) Programmable switch matrix (iv) I/O block

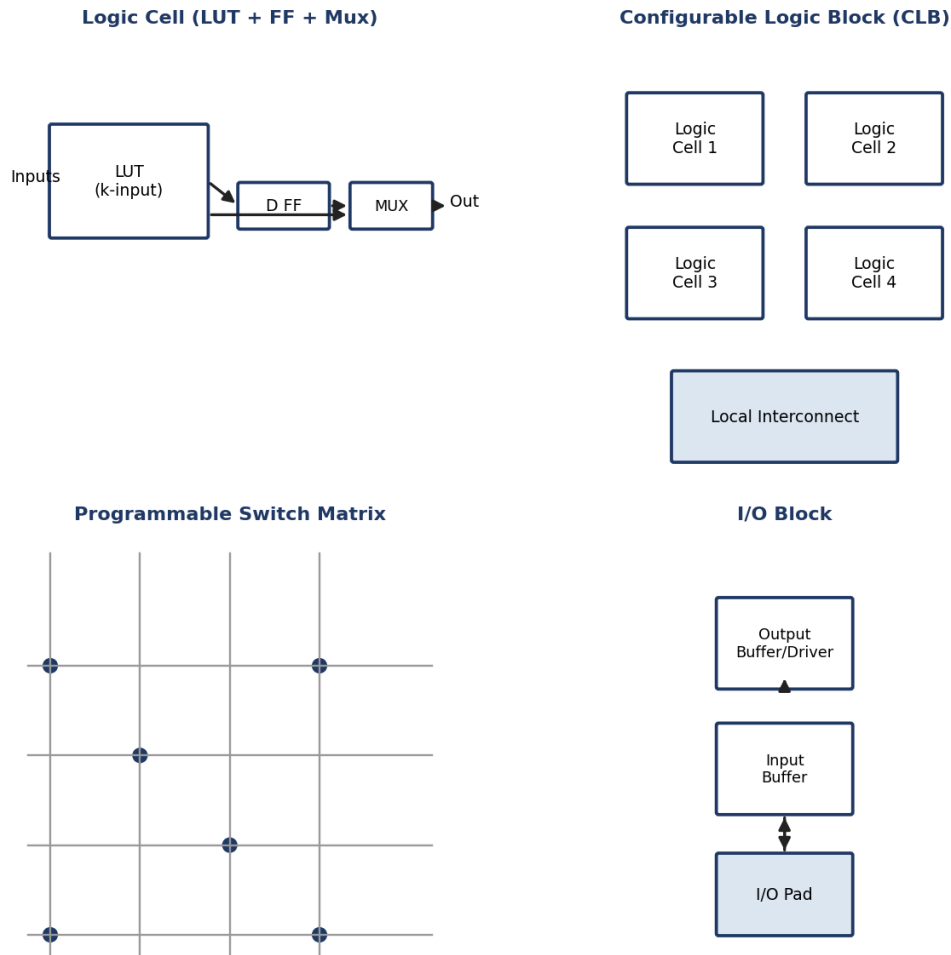


Fig: (a) Logic cell (b) CLB (c) Programmable switch matrix (d) I/O block

(i) Logic Cell: The smallest programmable computing unit in the FPGA fabric. It consists of a Look-Up Table (LUT) - a small SRAM-based memory addressed by the cell's input signals, storing the truth table of the desired k-input Boolean function - followed by a D flip-flop that can optionally register the LUT's combinational output, and an output multiplexer that selects between the registered and unregistered (combinational) result. Because any k-input Boolean function's truth table can be loaded into the LUT, a single logic cell can implement any function of up to k inputs (commonly $k = 4$ or 6 in modern devices).

(ii) CLB (Configurable Logic Block): A cluster grouping several logic cells (commonly 2, 4 or 8) together with dedicated fast local interconnect (and often dedicated carry-chain logic for efficient arithmetic), allowing more complex functions to be built with lower delay than if the logic cells had to communicate purely through the general-purpose routing fabric. The CLB is the basic 'tile' that is repeated to form the whole FPGA array.

(iii) Programmable Switch Matrix: Located at the intersections of horizontal and vertical routing channels between CLBs; internally it consists of a grid of programmable pass-transistor or multiplexer-based connection points. By selectively enabling certain connection points (through configuration bits), a signal travelling along one wire segment can be routed onto a different, perpendicular (or continuing) wire segment, which is what allows any CLB output to eventually be routed to any other CLB input, anywhere on the chip.

(iv) I/O Block: Positioned at the chip periphery, it contains an input buffer (to bring external signals into the fabric, often with configurable Schmitt-trigger thresholds), an output buffer/driver (with programmable drive strength and slew rate to control switching speed and EMI), and control logic to select the pin's direction (input, output, or bidirectional/tri-state) and the specific I/O voltage/signalling standard used (e.g. LVCMOS, LVDS).

Together, these four elements form the complete programmable fabric: logic cells provide the computation, CLBs group them efficiently, the switch matrix provides global connectivity, and I/O blocks provide the interface to the outside world - all configured by loading the appropriate bitstream into the device's configuration memory.

Q6. With the help of suitable schematics, compare PROM, PLA and PAL architectures.

PROM, PLA and PAL are all classical two-level (AND-OR) programmable logic device architectures used to implement combinational Boolean functions in sum-of-products form. They differ specifically in which of the two planes - the AND (product-term generation) plane or the OR (sum) plane - is fixed and which is user-programmable.

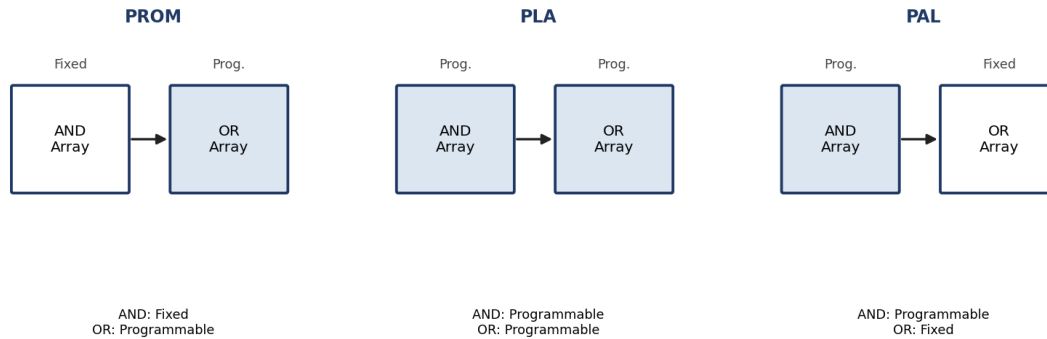


Fig: PROM, PLA and PAL - fixed vs. programmable AND/OR planes

Device	AND array	OR array	Remark
PROM (Programmable Read-Only Memory)	Fixed (implemented as a full address decoder - generates all possible minterms)	Programmable	Efficient only when a function genuinely needs many/all minterms; the AND array size grows exponentially (2^n decoder lines for n inputs) making it impractical for functions with many inputs
PLA (Programmable Logic Array)	Programmable	Programmable	Most flexible of the three - since both planes are programmable, multiple output functions can share common product terms, minimizing the required number of AND gates; however, the double programmable structure makes it more complex, larger, and generally slower than a PAL
PAL (Programmable Array Logic)	Programmable	Fixed (each output has a dedicated, fixed set of OR-plane connections)	Simpler, faster and cheaper to manufacture and program than a PLA since only one plane is programmable; the trade-off is that product terms cannot be shared between different outputs, so functions needing many shared terms are less area-efficient

Overall comparison: PROM is best suited to implement lookup-table-like functions where nearly all input combinations must be individually addressable (e.g. code converters); PLA gives the greatest logic-sharing flexibility at the cost of speed and complexity; PAL offers the best balance of speed, simplicity and cost for typical sum-of-products logic and became the historical basis for later SPLD/CPLD macrocell architectures (including the XC9500 family discussed earlier), since its fixed OR-plane with a small number of dedicated product terms per output closely matches the structure of a CPLD macrocell.

Q7. Explain with suitable diagram SRAM and anti-fuse programming techniques used in FPGA.

The 'programming technology' of an FPGA refers to the underlying physical mechanism used to store the configuration that determines the function of each LUT and the state of every routing switch. The two major technologies are SRAM-based and Anti-fuse-based programming, each with very different trade-offs.

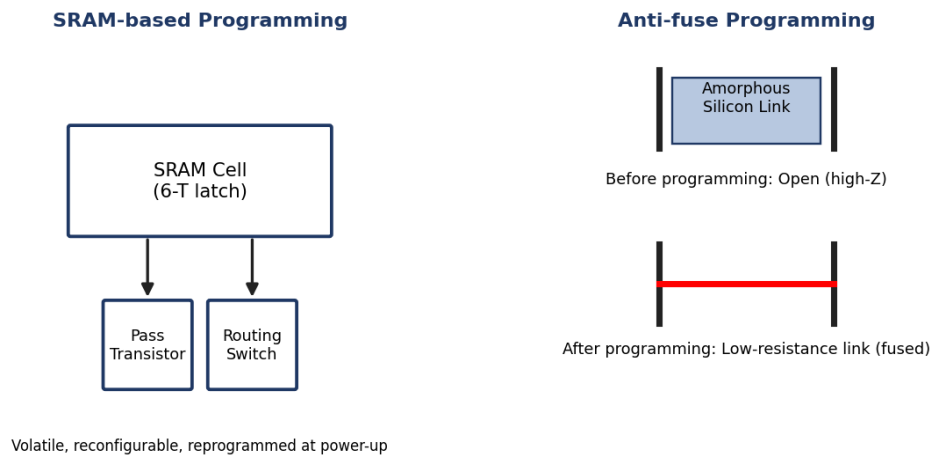


Fig: SRAM-based vs Anti-fuse programming

SRAM-based programming: Every configurable point in the fabric (LUT content bit, routing switch, I/O configuration) is controlled by a small SRAM latch cell. The stored bit directly turns an associated pass-transistor or multiplexer select line on or off, thereby determining the logic function or connection at that point.

- Volatile: the SRAM cells lose their content the instant power is removed, so the entire configuration must be reloaded from an external non-volatile source (configuration PROM, Flash memory, or an external processor) every time the device is powered up.
- Fully re-programmable - an essentially unlimited number of times, and can even be partially or fully reconfigured in-system during operation in advanced devices (dynamic partial reconfiguration).
- Manufactured using a completely standard CMOS logic process, allowing SRAM-based FPGAs to track the latest, smallest process nodes and hence offer the highest logic density and lowest cost per transistor.
- This is by far the dominant technology used in almost all modern general-purpose FPGAs (Xilinx/AMD, Intel/Altera, etc.).

Anti-fuse programming: Uses a special two-terminal structure (e.g. a thin layer of amorphous silicon, or a dielectric layer) sandwiched between two conductor layers, which is normally in a high-resistance (open) state. Applying a relatively high programming voltage/current across the anti-fuse causes localized, permanent dielectric breakdown, creating a low-resistance conductive link - essentially the opposite behaviour of a conventional fuse (which starts closed and is 'blown' open).

- One-Time Programmable (OTP): once an anti-fuse link is programmed (blown), it cannot be reset or reprogrammed - the configuration is permanent.
- Non-volatile by nature: since the programmed state is a physical, permanent change in the material, no external configuration memory or reload process is needed at power-up - the device is instantly functional.
- Lower parasitic capacitance and smaller physical size of the switch (compared to an SRAM cell plus pass transistor) generally gives anti-fuse FPGAs faster routing and better resistance to radiation-induced upset, making them attractive for aerospace, defense and other high-reliability, security-sensitive applications where re-programmability is undesirable and permanence/security of the design is valued.

Summary comparison: SRAM technology offers unlimited reprogrammability, uses a standard CMOS process (lowest cost/highest density), but is volatile and requires external configuration memory and a boot delay at power-up; anti-fuse technology offers instant-on, non-volatile, radiation-tolerant, tamper-resistant, permanent configuration, at the cost of being one-time programmable only, generally on a more specialized (costlier) fabrication process.

Q8. What is a role of CPLD and FPGA in embedded system developments?

CPLDs and FPGAs play several important, complementary roles throughout the development lifecycle and final deployment of modern embedded systems, alongside microcontrollers/microprocessors.

- 1. Rapid prototyping and design verification:** Before committing to a full custom ASIC (which involves very high non-recurring engineering cost and long fabrication turnaround), a digital design can be quickly implemented and tested on an FPGA, allowing designers to verify functional correctness, debug the logic in real hardware, and even demonstrate the product to stakeholders, all with much faster iteration than an ASIC flow.
- 2. Glue logic and system interfacing:** CPLDs are frequently used in embedded boards to implement simple 'glue logic' such as address decoding for memory-mapped peripherals, bus-protocol translation between chips using different voltage/logic standards, chip-select generation, and power-sequencing/reset control logic - functions that are simple enough not to justify a full FPGA, but too specific for a fixed off-the-shelf logic chip.
- 3. Custom peripherals and hardware acceleration:** FPGAs allow embedded designers to implement application-specific hardware co-processors, custom communication protocol controllers, and DSP/signal-processing pipelines that run in parallel with (and much faster than) a general-purpose embedded CPU executing the equivalent function in software - this hardware/software co-design approach is central to many high-performance embedded systems (e.g. motor control, image processing, software-defined radio).
- 4. Field reconfigurability:** Because FPGA (and, via JTAG/ISP, CPLD) configuration can be updated after the product has shipped, embedded systems built around these devices can receive functional upgrades, bug fixes, or entirely new hardware features through a firmware/bitstream update in the field, without any change to the physical board - a major advantage over fixed-function ASICs.
- 5. System integration (SoC-FPGAs):** Modern SoC-FPGA devices integrate a hard, high-performance ARM processor core directly on the same die as the programmable FPGA fabric, allowing a complete embedded system - CPU, custom hardware accelerators, memory controllers and I/O - to be implemented on a single chip, combining the software flexibility of a processor with the hardware parallelism and speed of programmable logic.

Assignment 4

Unit IV: Digital CMOS Circuits | CMOS Gate Design, Delay and Power

Q1. Explain the following: (i) Body Effect (ii) Hot electron Effect

(i) Body Effect (Substrate-bias effect): In a MOSFET, the threshold voltage V_T is derived assuming the source and body (substrate/bulk) terminals are at the same potential ($V_{SB} = 0$). In real digital circuits, however, whenever transistors are stacked in series - for example, the two NMOS transistors in series inside a CMOS NAND gate - the source terminal of the upper transistor is not directly at ground; it sits at whatever intermediate voltage exists at the internal node, so a non-zero source-to-body voltage V_{SB} exists (since the body/substrate is tied to a fixed reference, usually ground for NMOS).

This non-zero V_{SB} widens the depletion region under the gate, which in turn increases the threshold voltage according to the body-effect equation:

$$V_T = V_{T0} + \gamma [\sqrt{2\phi_F + V_{SB}} - \sqrt{2\phi_F}]$$

where V_{T0} is the threshold voltage at $V_{SB} = 0$, γ is the body-effect (substrate-bias) coefficient (a process-dependent constant, typically $0.4 - 1 \text{ V}^{0.5}$), and ϕ_F is the Fermi potential of the substrate.

Consequences: As V_{SB} increases, V_T increases, which reduces the transistor's effective gate overdrive ($V_{GS} - V_T$) and hence reduces its drive current for the same applied V_{GS} . This means the upper (stacked) transistor in a series-connected structure is effectively 'weaker' than a single, unstacked transistor - an important second-order effect that must be included when calculating realistic delay or sizing series-connected transistors (such as in NAND/NOR gates or pass-transistor chains).

(ii) Hot Electron Effect (Hot Carrier Injection, HCI): As channel lengths shrink into the deep-submicron regime while supply voltages are not scaled down proportionally, the lateral electric field near the drain end of the channel becomes very high. Electrons travelling through this high-field region can gain enormous kinetic energy - far above their normal thermal energy - and are therefore called 'hot' electrons/carriers.

- Some of these hot electrons gain enough energy to overcome the Si-SiO₂ potential barrier and become injected into the gate oxide, where a fraction of them get permanently trapped as fixed charge.
- This trapped charge causes a gradual, permanent shift in the threshold voltage V_T and a corresponding degradation of transconductance (g_m) and drive current over the operating lifetime of the device - this is a key long-term reliability and 'aging' mechanism in scaled CMOS technology.
- Hot electron degradation is worst for transistors that repeatedly operate with high V_{DS} while V_{GS} is near its threshold (the saturation/pinch-off region), which is a common bias condition in fast-switching digital circuits.

Mitigation techniques: Use of Lightly-Doped Drain (LDD) structures to spread the peak electric field over a wider region and reduce its maximum value; scaling down V_{DD} along with channel length (constant-field scaling, discussed later); and careful circuit-level design margins/derating to account for expected lifetime V_T shift.

Q2. Derive the static and dynamic power dissipation in CMOS.

Total power dissipation in a CMOS digital circuit has two fundamental components: static (leakage) power, dissipated even when the circuit is not switching, and dynamic power, dissipated as a direct result of switching activity. A precise understanding of both is essential for low-power VLSI design, since dynamic power historically dominated, but static/leakage power has become increasingly significant (and in the most advanced nodes, comparable to or exceeding dynamic power) as devices scale down.

(A) Static (Leakage) Power: In an ideal CMOS logic gate, for any valid steady-state input combination, exactly one of the two complementary networks (the pull-up PMOS network or the pull-down NMOS network) is fully ON while the other is fully OFF, so ideally there is no direct DC conduction path from V_{DD} to GND, and static power should be zero.

In real devices, however, the 'OFF' transistors are not perfect open switches; several leakage mechanisms allow a small but non-zero current to flow even in the steady 'OFF' state:

- Sub-threshold leakage ($I_{\text{subthreshold}}$): current that flows between drain and source even when V_{GS} is below V_T , due to the exponential tail of the MOSFET's conduction characteristic - this is the dominant leakage mechanism in most modern processes, and increases exponentially as V_T is reduced (a direct, unwanted side-effect of scaling V_T down to maintain drive current at lower V_{DD}).
- Gate leakage (I_{gate}): tunneling current through the very thin gate oxide, becoming significant as oxide thickness scales into the nanometer range.
- Junction (reverse-bias diode) leakage (I_{junction}): leakage current across the reverse-biased source/drain-to-body p-n junctions.

Total static power is then given by:

$$P_{\text{static}} = I_{\text{leakage}} \times V_{DD}, \text{ where } I_{\text{leakage}} = I_{\text{subthreshold}} + I_{\text{gate}} + I_{\text{junction}}$$

(B) Dynamic Power: Has two distinct components - switching (capacitive) power and short-circuit power.

(i) Switching power: Every time a gate's output transitions, its output (load) capacitance CL must be charged ($0 \rightarrow 1$ transition, drawing charge from V_{DD}) or discharged ($1 \rightarrow 0$ transition, dumping charge to ground). Considering a full $0 \rightarrow 1 \rightarrow 0$ charge/discharge cycle, the energy drawn from the supply per full cycle is $CL \cdot V_{DD}^2$, and if the gate switches with an average activity factor α (the probability/fraction of clock cycles in which the output actually transitions) at a clock frequency f , the average switching power is:

$$P_{\text{switching}} = \alpha \times CL \times V_{DD}^2 \times f$$

(Derivation sketch: energy delivered by the supply when charging CL from 0 to V_{DD} through the PMOS network is $E_{\text{charge}} = CL \cdot V_{DD}^2$ total, of which exactly half, $\frac{1}{2}CL \cdot V_{DD}^2$, is stored on the capacitor and the other half is dissipated as I^2R loss in the PMOS switch resistance; on the subsequent discharge, the stored $\frac{1}{2}CL \cdot V_{DD}^2$ is dissipated in the NMOS switch resistance. So each full charge-discharge cycle dissipates a total of $CL \cdot V_{DD}^2$ from the supply, which when multiplied by the switching activity α and frequency f gives the average power formula above.)

(ii) Short-circuit power: Because real input signals have a finite (non-zero) rise/fall time rather than an ideal instantaneous transition, there is a brief interval during each transition when both the PMOS and NMOS networks are simultaneously partially ON, creating a direct, momentary conduction path from V_{DD} straight to GND. The resulting current spike, integrated over time and multiplied by the switching frequency, gives the short-circuit power:

$$P_{\text{short-circuit}} = I_{\text{sc}}(\text{avg}) \times V_{DD} \times f, \text{ where } I_{\text{sc}}(\text{avg}) \text{ is the average short-circuit current per transition}$$

Total power dissipation:

$$P_{\text{total}} = P_{\text{static}} + P_{\text{switching}} + P_{\text{short-circuit}}$$

In most digital VLSI circuits at moderate-to-high frequency, $P_{\text{switching}}$ dominates the total; however, at very low operating frequencies or in standby/sleep modes, P_{static} (leakage) can dominate, which is why low-power design techniques (power gating, multi- V_T transistors, clock gating, voltage scaling) specifically target reducing whichever component is dominant for the given operating mode.

Q3. Explain the working of transmission gate and implement a circuit of 2:1 multiplexer using transmission gate.

A pass transistor (a single NMOS or PMOS used as a simple switch) has a significant drawback: an NMOS pass transistor transmits a strong logic '0' but a 'weak' logic '1' (degraded by one threshold-voltage drop, $V_{GS} = V_T$ loss near $V_{DD} - V_T$), while a PMOS pass transistor transmits a strong '1' but a weak '0'. Neither device alone can pass both logic levels cleanly, which is problematic for cascaded logic and for driving the gates of following transistors reliably.

Transmission Gate (TG): A Transmission Gate solves this problem by connecting an NMOS and a PMOS transistor in parallel between the same input and output nodes, with their gates driven by complementary control signals - the NMOS gate is driven by control signal S , and the PMOS gate is driven by its complement S' ($S' = \text{NOT } S$).

- When $S = 1$ ($S' = 0$): both the NMOS (VGS sufficiently positive) and PMOS (VSG sufficiently positive, since its gate is at 0 while source can be near VDD) are turned ON simultaneously, so the gate is 'closed' - it conducts, and the strong '0' capability of the NMOS combined with the strong '1' capability of the PMOS together pass both logic levels with low, roughly constant, resistance across the full input voltage range.
- When $S = 0$ ($S' = 1$): both transistors are OFF, so the gate is 'open' - presenting a very high impedance between input and output, effectively isolating them.
- Because the ON-resistance of a transmission gate stays relatively constant across the input voltage swing (the NMOS resistance rises as input approaches VDD, while the PMOS resistance falls, and vice versa near GND, so the parallel combination stays roughly constant), TGs are widely used not only as simple switches but also as the core building block of pass-transistor-logic multiplexers, XOR/XNOR gates, and D-latches.

2:1 Multiplexer using Transmission Gates:

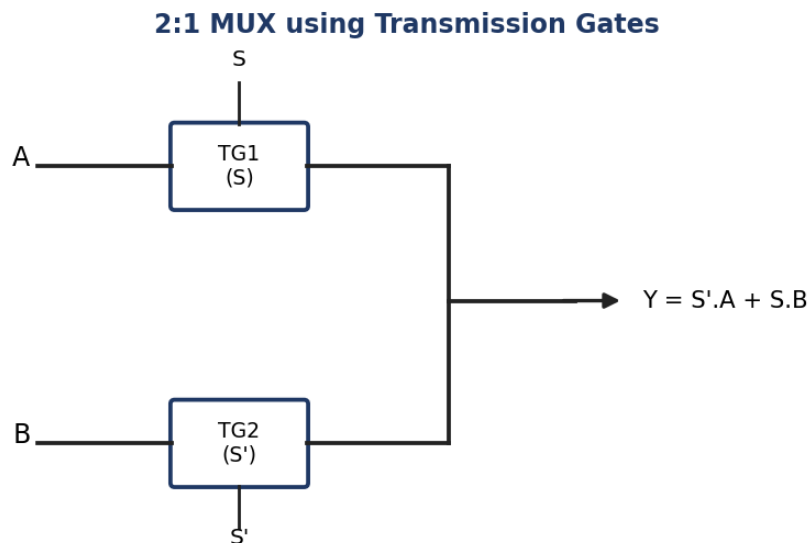


Fig: 2:1 MUX using transmission gates

The circuit uses two transmission gates, TG1 (controlled by select signal S) connected to input A , and TG2 (controlled by the complementary select S') connected to input B , with both TG outputs tied together at the common output node Y .

- When $S = 1$ ($S' = 0$): TG1 is ON (passes A through to Y) and TG2 is OFF (B is isolated) $\rightarrow Y = A$.
- When $S = 0$ ($S' = 1$): TG2 is ON (passes B through to Y) and TG1 is OFF (A is isolated) $\rightarrow Y = B$.

This behaviour is captured by the Boolean expression: $Y = S'.A + S.B$

Advantages of this implementation: Only 4 transistors are needed (2 per transmission gate) compared to 6 transistors for an equivalent static-CMOS AOI-based multiplexer, giving a smaller area and lower parasitic capacitance; however, since a TG-based mux has no dedicated buffering/restoring stage, its output can be somewhat weaker (higher output impedance) than a fully-buffered static gate, and long chains of pass-transistor logic can accumulate delay - so buffers are often inserted periodically in larger pass-transistor-logic designs.

Q4. Draw NAND, NOR, AND, OR, EX-OR gates using CMOS.

Every static CMOS logic gate is built from two complementary networks: a Pull-Up Network (PUN), made of PMOS transistors, connecting the output to VDD, and a Pull-Down Network (PDN), made of NMOS transistors, connecting the output to GND. The PUN and PDN are always the logical dual (complement) of each other - series connections in one network correspond to parallel connections in the other - which guarantees that for every valid input combination, exactly one of the two networks conducts, giving a well-defined, non-conflicting output at all times and (ideally) zero static power.

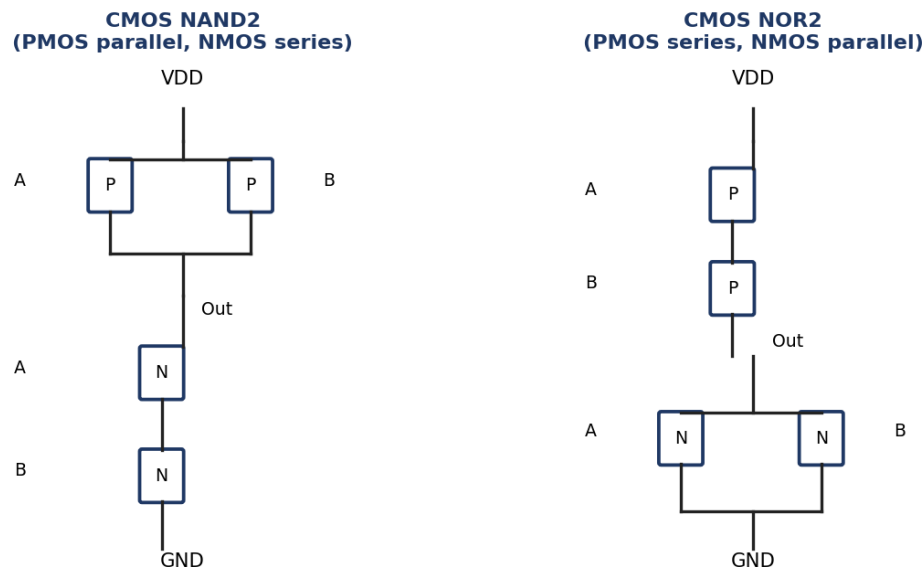


Fig: CMOS NAND2 (PMOS parallel/NMOS series) and NOR2 (PMOS series/NMOS parallel)

NAND2 gate ($Y = (A \cdot B)'$): The pull-down NMOS network must conduct (pull output low) only when BOTH A AND B are '1' - this AND condition is realized by placing the two NMOS transistors in SERIES. By the duality rule, the pull-up PMOS network must be the dual - the two PMOS transistors are placed in PARALLEL, so that the output is pulled high whenever EITHER A OR B is '0'.

NOR2 gate ($Y = (A + B)'$): The pull-down network must conduct whenever EITHER A OR B is '1' (an OR condition), so the two NMOS transistors are placed in PARALLEL. The dual pull-up PMOS network must then have the two PMOS transistors in SERIES, conducting (pulling output high) only when BOTH A AND B are '0'.

AND gate: Since a single static CMOS stage is naturally inverting (it can only directly implement NAND/NOR/AOI/OAI functions, never a non-inverted AND/OR directly), a true AND gate is built as a NAND gate followed by a CMOS inverter stage: $AND = NAND + INV$. This adds 2 extra transistors (1 PMOS + 1 NMOS for the inverter) beyond the NAND gate's own 4 transistors, for a total of 6 transistors.

OR gate: Similarly built as a NOR gate followed by an inverter: $OR = NOR + INV$, again totaling 6 transistors.

EX-OR gate ($Y = A \oplus B = A \cdot B' + A' \cdot B$): Can be implemented in several ways: (a) as a standard static-CMOS AOI (AND-OR-INVERT) structure realizing the complementary function $A' \cdot B' + A \cdot B$ directly requiring 8 transistors for XNOR then an extra inverter for XOR, or more commonly built from 4 two-input NAND gates in the classic '4-NAND EXOR' structure; or (b) much more compactly using pass-transistor/transmission-gate logic, which can realize XOR/XNOR with as few as 6-8 transistors by directly routing A or A' to the output depending on control signal B, exploiting the fact that XOR is fundamentally a data-selection/routing function rather than a pure AND-OR function - this pass-transistor approach is the one commonly preferred in practice for area- and power-efficient EXOR gates.

Q5. Draw and Explain RC Delay Model for NMOS and PMOS transistor. Give example.

Exact MOSFET I-V behaviour is non-linear and depends on the operating region (linear/triode or saturation), which makes hand-calculation of gate delay difficult. The RC delay model is a widely used simplification that approximates a switching MOSFET as an ideal switch in series with a fixed, effective resistance R (representing its 'ON' resistance), driving a lumped equivalent load capacitance CL (representing the total capacitance of the wire plus the gate capacitances of all fan-out gates being driven).

RC Equivalent Delay Model of a Gate

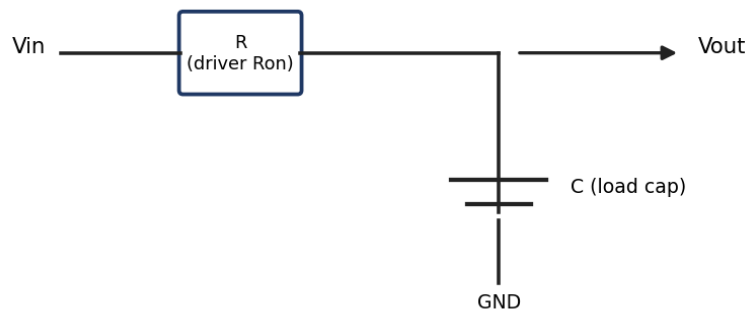


Fig: RC equivalent delay model of a CMOS gate

Physical basis of R: The equivalent resistance of a MOSFET switch is inversely proportional to its (W/L) ratio and to carrier mobility: $R \propto L / (\mu \cdot C_{ox} \cdot W \cdot (V_{GS} - V_T))$ — approximately, R is treated as a constant for a first-order estimate, with $R_{NMOS} \propto 1/\mu_n$ and $R_{PMOS} \propto 1/\mu_p$.

NMOS vs PMOS resistance asymmetry: Because hole mobility (μ_p) in silicon is roughly 2 to 2.5 times lower than electron mobility (μ_n), a PMOS transistor of the same (W/L) as an NMOS transistor has roughly 2-2.5 times higher ON resistance. This means a CMOS inverter built with equal-sized NMOS and PMOS would pull down (discharge) much faster than it pulls up (charges), giving asymmetric rise/fall delay.

Standard sizing rule: To balance rise and fall delay (equal drive strength in both directions), the PMOS transistor width is made roughly 2 to 2.5 times the NMOS width for the same channel length - i.e. $(W/L)_{PMOS} \approx 2 \text{ to } 2.5 \times (W/L)_{NMOS}$ - so that $R_{PMOS} \approx R_{NMOS}$ despite the mobility difference.

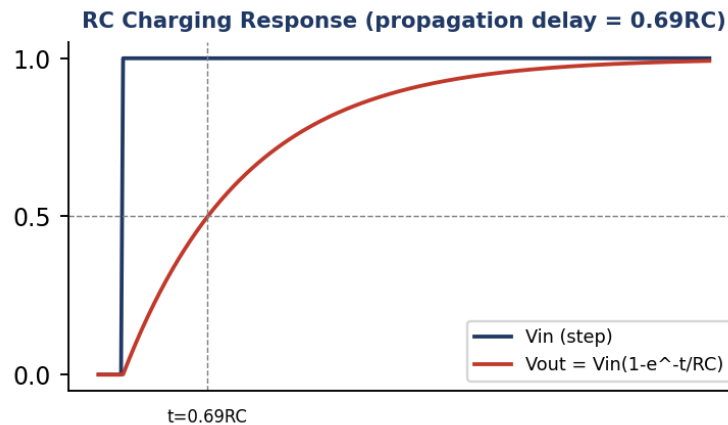


Fig: Output response - exponential RC charging

Delay estimate from the RC model: For a simple RC circuit responding to a step input, the output voltage rises/falls exponentially, $V_{out}(t) = V_{DD}(1 - e^{-(t/RC)})$ for charging. The time taken to reach the 50% point ($V_{DD}/2$), which is the standard reference point for defining propagation delay, works out to:

$$t_p \approx 0.69 \times R \times CL$$

Example calculation: If the effective driver resistance $R = 5 \text{ k}\Omega$ and the total load capacitance $CL = 20 \text{ fF}$, then:

$$t_p \approx 0.69 \times 5 \times 10^3 \Omega \times 20 \times 10^{-15} \text{ F} = 0.69 \times 100 \times 10^{-12} \text{ s} = 69 \times 10^{-12} \text{ s} = 69 \text{ ps}$$

This simple, first-order RC model - though it ignores non-linear MOSFET behaviour, velocity saturation, and the exact shape of the input waveform - is extremely useful for quick, by-hand delay estimation during early-stage transistor sizing and for building intuition about how gate delay scales with fan-out capacitance and transistor sizing, before resorting to full SPICE-level simulation for final sign-off accuracy.

Q6. Design CMOS logic for $Y = ABC + D$. Calculate W/L ratio for NMOS and PMOS are needed on chip.

To design a single static CMOS complex gate for a given Boolean function, the standard procedure is: (1) express the function to be realized at the output node - since a single CMOS stage is inherently inverting, the NMOS pull-down network is designed to conduct (pull the output LOW) exactly when the function's complement condition, i.e. when the un-negated expression itself is true if we treat the gate as producing Y' at that node, or equivalently the PDN is built to implement the AND/OR structure of the given expression directly, and the PUN is then built as its logical dual.

For $Y = ABC + D$: the pull-down NMOS network must conduct whenever (A AND B AND C) is true, OR whenever D is true. This is realized as: NMOS(A), NMOS(B), NMOS(C) connected in SERIES (implementing the AND/ABC term), and this whole series branch connected in PARALLEL with a single NMOS(D) (implementing the OR with D).

CMOS Complex Gate: $Y = (ABC + D)'$
PDN: (A.B.C) parallel with D | PUN: dual (series/parallel swap)

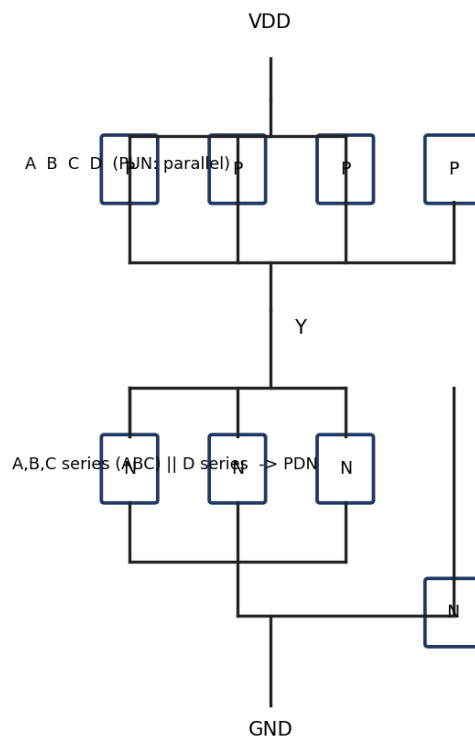


Fig: CMOS complex gate realizing $Y' = (ABC + D)'$

Pull-down network (NMOS): NMOS(A) - NMOS(B) - NMOS(C) in series, and this series combination placed in parallel with NMOS(D).

Pull-up network (PMOS) - the dual network: By the duality rule (series $\leftrightarrow$ parallel), the PMOS network has PMOS(A), PMOS(B), PMOS(C) in PARALLEL with each other, and this parallel combination connected in SERIES with PMOS(D).

W/L sizing procedure: The goal of sizing is to give every possible conduction path through the network the same effective drive strength (same equivalent resistance) as a single 'reference' unit inverter, whose NMOS has $(W/L)_n = 1$ and PMOS has $(W/L)_p = 2$ (the standard mobility-balancing ratio, as explained in Q5). The rule used is: transistors in SERIES need their width multiplied (their individual W/L increased) to compensate for the added series resistance, while transistors in PARALLEL do not need extra width, since parallel paths already reduce the combined resistance on their own (only the worst-case single-path condition needs to meet the unit reference resistance).

NMOS sizing:

- A, B, C are in series (3 transistors in series is equivalent, in resistance, to a single transistor of length $3L$) - to bring this series branch's resistance back down to the unit reference value, each of A, B, C individually needs $(W/L)_n = 3$ (i.e. width = $3 \times$ the unit NMOS width).
- D stands alone in its own parallel branch (only a single transistor in that path, no series stacking), so it only needs the unit reference size: $(W/L)_n = 1$.

PMOS sizing:

- In the pull-up network, A, B and C are in PARALLEL with each other, so each individually only needs the unit PMOS reference size: $(W/L)_p = 2$ (matching the standard 2:1 PMOS:NMOS mobility-compensation ratio, since no series-stacking penalty applies to a parallel branch).
- D is alone, in series with the $(A||B||C)$ parallel group; since it is a single transistor on that series path (not stacked with any other PMOS), it also only needs the unit reference size: $(W/L)_p = 2$.

Final summary of required sizing:

Transistor	Network	Required W/L
NMOS - A, B, C (each)	Pull-down, series branch	3
NMOS - D	Pull-down, parallel branch	1
PMOS - A, B, C (each)	Pull-up, parallel branch	2
PMOS - D	Pull-up, series branch	2

This sizing ensures that in the worst case (only one series path conducting, e.g. only A-B-C all high, or only D high), the gate delivers drive current equivalent to the unit reference inverter in both the pull-up and pull-down directions, keeping propagation delay balanced regardless of which input combination causes the transition.

Q7. Define scaling and explain any one type of scaling.

Scaling in VLSI refers to the systematic, proportional reduction of MOSFET physical dimensions - channel length L , channel width W , and gate-oxide thickness t_{ox} - together with a corresponding reduction of operating voltages, across successive process technology generations, with the primary goals of increasing transistor density (more transistors per unit area, lowering cost per function), increasing switching speed, and reducing power consumption per logic operation.

Types of scaling: Two classical scaling models are defined, differing in how the supply voltage is treated relative to the dimensional scaling factor S (all dimensions reduced by $1/S$): Constant-Field (Full) Scaling, and Constant-Voltage Scaling.

Constant-Field (Full) Scaling - explained in detail: In this model, all dimensions AND all voltages (V_{DD} , V_T , and all internal node voltages) are scaled down by the same factor $1/S$, while the substrate/channel doping concentration is scaled UP by the factor S . The specific intent of this proportional scaling is to keep the internal electric field within the device ($E = V/L$) exactly constant from one generation to the next, which helps avoid worsening short-channel effects such as hot-carrier degradation, velocity saturation and oxide breakdown that would otherwise result if voltage were kept constant while dimensions shrank.

Parameter	Scaling factor (constant-field)
Physical dimensions: L , W , t_{ox}	$1/S$
Voltages: V_{DD} , V_T	$1/S$
Doping concentration N_A , N_D	S
Gate capacitance per unit area (C_{ox})	S
Total gate capacitance $C (=C_{ox} \cdot W \cdot L)$	$1/S$
Drain current I_D (saturation)	$1/S$
Device area (per transistor)	$1/S^2$

Parameter	Scaling factor (constant-field)
Gate delay ($\tau = C \cdot V/I$)	$1/S$
Power dissipation per gate ($P = C \cdot V^2 \cdot f$)	$1/S^2$
Power density (P per unit area)	constant (unchanged)

Interpretation of the results: Constant-field scaling predicts that as dimensions shrink by $1/S$, transistor delay improves by the same factor $1/S$ (i.e. speed improves by S), while power per gate drops even faster, by $1/S^2$, and - crucially - power density (power per unit chip area) remains constant even though far more transistors are packed into the same area, because both the power-per-gate reduction and the area-per-gate reduction scale together. This constancy of power density is what historically allowed chip complexity and clock speed to increase generation after generation without the chip 'melting' from excessive heat - this was the central assumption behind Dennard Scaling.

Practical limitation: In real industrial practice, VDD has NOT always been scaled down exactly in proportion to dimensions (partly for compatibility with existing I/O standards, and partly because reducing V_T to match a lower VDD would cause sub-threshold leakage to increase exponentially) - this deviation from ideal constant-field scaling (sometimes called constant-voltage or generalized/quasi-constant-voltage scaling) is a major reason why power density has, in fact, been rising in recent process generations, directly causing the modern 'power wall' and the associated design challenges (leakage power, thermal management, dynamic voltage-frequency scaling) faced in current deep-submicron VLSI design.

Assignment 5

Unit V: Application Specific Integrated Circuit | Layout Design Rules and Physical Design Issues

Q1. Which lambda rules are used for CMOS layout? Give its significance.

When drawing the physical layout (mask geometry) of a CMOS circuit, every layer (diffusion, polysilicon, metal, contacts/vias) must obey a set of minimum-dimension design rules - minimum width, minimum spacing between features, and minimum overlap/enclosure between layers - imposed by the limitations of the photolithographic fabrication process. There are two common ways of expressing these rules: micron-based (absolute) rules, and lambda-based (scalable) rules.

Lambda (λ) based design rules: Introduced by Carver Mead and Lynn Conway (the 'Mead-Conway rules'), this approach expresses every single layout dimension as a multiple of one fundamental unit, λ (lambda), which is defined as half of the minimum drawable transistor channel length for a given process ($\lambda = L_{min}/2$). Every design rule in the entire rule set is then written purely in terms of λ , completely independent of the actual absolute size (in microns) of that specific fabrication process.

Lambda (λ) Based Design Rules

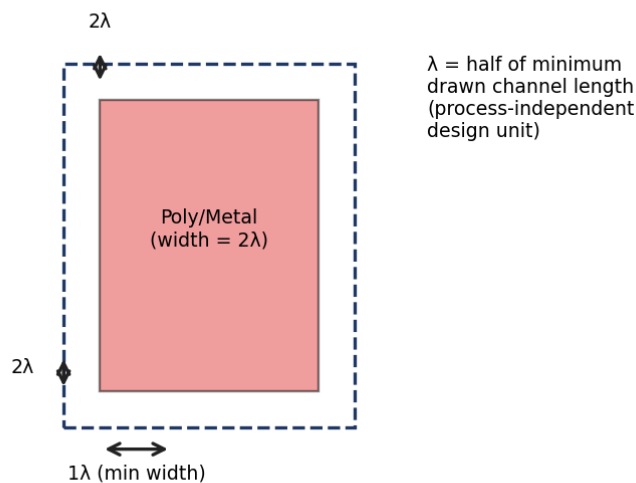


Fig: Lambda-based layout dimensioning

Representative lambda rules:

- Minimum width of a feature (e.g. poly or metal line): 2λ
- Minimum spacing between two features on the same layer: 2λ to 3λ
- Minimum overlap of poly over diffusion (forming the transistor gate) / of diffusion over poly (source-drain contact area): 1λ to 2λ
- Minimum contact/via size and its required overlap by the layers it connects: typically $2\lambda \times 2\lambda$, with 1λ overlap

Significance of lambda rules:

- Process (technology) independence and portability: because every dimension is expressed as a multiple of λ rather than an absolute micron value, exactly the same layout drawing can be 'migrated' to a completely different (usually smaller/newer) fabrication process simply by re-defining the numerical value of λ for that new process - no geometry needs to be redrawn.

- Design reuse: a library of standard cells or a complete chip layout designed once in λ units can be reused across multiple technology generations, dramatically reducing design/verification effort and time-to-market for each new process node.
- Simplified, unified design-rule checking (DRC): since all rules reduce to simple multiples of one number, the rule set - and the DRC deck used to verify it - is much simpler to write, understand, and check than a full set of independent absolute-micron rules for every layer and layer-pair.
- Educational/pedagogical value: lambda rules make it easy for students and new designers to learn and apply layout design rules without needing process-specific data, which is why they remain a standard teaching tool even though modern industrial design uses micron-based rules almost exclusively.

Trade-off / limitation: Because a single λ value is applied uniformly to many different layers and layer combinations, lambda rules are inherently more conservative (larger, less area-efficient) than a full, layer-specific micron-based rule set, which can exploit the exact minimum capability of each individual layer in a specific process - this is why real industrial ASIC design today uses micron-based (absolute) design rules, provided by the foundry as part of a Process Design Kit (PDK), rather than lambda rules.

Q2. Explain Design issues like antenna effect and electro migration effect.

As VLSI technology scales to smaller feature sizes and higher current densities, several physical/process-related design issues must be explicitly checked and mitigated during the physical design and layout verification stages, beyond simple geometric design rules. Two important such issues are the Antenna Effect and Electromigration.

Antenna Effect (Plasma-Induced Gate Oxide Damage): During the plasma-etching steps used to pattern metal and polysilicon interconnects, the exposed conductor being etched acts electrically like an antenna - it collects stray charge from the ionized plasma environment. If this particular piece of metal/poly is, at that stage of fabrication, connected only to a MOSFET gate (and not yet connected to any diffusion/junction that could safely discharge the accumulated charge, since diffusion layers are typically formed and connected at a different process step), the collected charge has nowhere to go except through the very thin gate oxide beneath it.

- If the accumulated charge (and hence the resulting voltage across the thin gate oxide) becomes large enough, it can exceed the oxide's breakdown field strength and cause immediate or latent (delayed, reliability-affecting) damage to the gate oxide - permanently degrading or destroying that transistor's characteristics.
- The severity of the antenna effect for a given net is generally proportional to the ratio of the total exposed conductor area (or perimeter) connected to a gate, divided by the gate area itself - this ratio is called the 'antenna ratio', and foundries specify a maximum allowed antenna ratio per metal layer as part of the design rules.

Remedies for the antenna effect:

- Antenna diodes: small reverse-biased diodes are deliberately added, connected to the sensitive net close to the gate, providing a safe path to bleed off any accumulated charge to the substrate before it can damage the oxide.
- Jumper/bridging technique (layer splitting): the offending long metal segment is broken and part of the net is routed on a higher metal layer that is deposited/etched at a later process step - since that later step occurs after the transistor's source/drain diffusion (and hence a safe discharge path) already exists, the accumulated-charge risk is eliminated for that portion of the route.
- Design rule limits on the maximum length/area of any single metal segment that connects directly to a gate on a given layer, enforced automatically by the router and checked by an 'Antenna Rule Check' during physical verification.

Electromigration (EM): At the very high current densities typical of narrow, deep-submicron interconnects, the sustained flow of electrons transfers momentum to the metal ions forming the lattice of the wire. Over time, this repeated momentum transfer causes gradual physical migration/displacement of metal atoms in the direction of electron flow.

- At the location where atoms are being depleted (typically near the cathode end, where the electron flow originates), voids form in the metal, which can grow over time until they cause a complete open-circuit failure of the interconnect.

- At the location where the displaced atoms accumulate (typically near the anode end), hillocks or whiskers of extruded metal can form, which may grow large enough to create an unintended short-circuit to a neighbouring interconnect.
- EM is a slow, cumulative, statistical (time-dependent) failure mechanism - it does not usually cause immediate failure but progressively degrades reliability, and is characterized by a 'Mean Time To Failure' (MTTF) that decreases roughly as the inverse square (or higher power) of current density, per Black's equation.

Remedies for electromigration:

- Enforce a maximum allowed current density (current per unit cross-sectional wire area) for every metal layer, as a mandatory design rule, especially for high-current power/ground and clock nets.
- Use wider metal traces, or multiple metal layers/multiple parallel vias in parallel, for nets that must carry large currents (such as power distribution and clock trunks), directly reducing the current density for the same total current.
- Use more EM-resistant interconnect materials - modern processes have largely switched from aluminum to copper interconnect specifically because copper has significantly better electromigration resistance (higher MTTF for the same current density) than aluminum.

Q3. Draw stick diagram for CMOS inverter, NAND, NOR gate.

A stick diagram is a simplified, topological (not-to-scale) representation of a physical VLSI layout, drawn using colour-coded lines/bars to represent each mask layer - it captures the relative placement and connectivity of transistors and interconnect, without showing exact widths, spacings or absolute dimensions. Stick diagrams are an essential intermediate design step between a transistor-level schematic and the final, fully dimensioned mask layout, since they let a designer quickly plan an efficient transistor arrangement (minimizing area, avoiding unnecessary jogs/vias) before investing time in exact geometric layout.

Standard colour/line conventions used:

- Red line = Polysilicon (forms the transistor gate wherever it crosses a diffusion region)
- Green (or yellow) line = Metal (interconnect wiring)
- Orange/brown region = P-diffusion (source/drain regions of PMOS transistors, normally placed in the N-well/upper part near VDD)
- Blue region = N-diffusion (source/drain regions of NMOS transistors, normally placed near GND)
- A black filled square/dot = Contact or via (connection between two different layers, e.g. metal-to-diffusion or metal-to-poly)

CMOS Inverter stick diagram: A single PMOS transistor is placed in the P-diffusion strip (upper row, connected to VDD), and a single NMOS transistor is placed in the N-diffusion strip (lower row, connected to GND); a single vertical polysilicon line crosses both diffusion strips simultaneously, forming both the PMOS and NMOS gates together - this shared poly line is the common input, and the drains of both transistors are joined by a metal line to form the common output.

Stick Diagram: CMOS Inverter

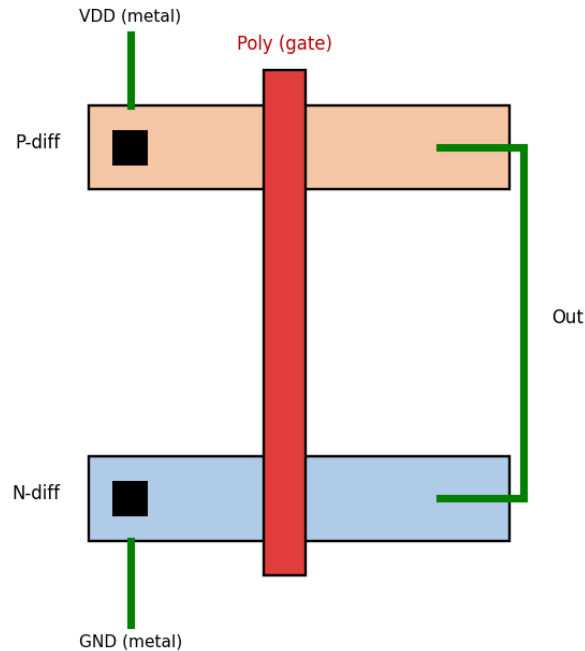


Fig: Stick diagram - CMOS Inverter

CMOS NAND2 stick diagram: The two NMOS transistors (for the series pull-down network) share a single, continuous N-diffusion strip, crossed by two separate poly gate lines (one for input A, one for input B) - placing series-connected transistors on one continuous diffusion strip, without breaking it, is what naturally realizes the series connection in the stick diagram. The two PMOS transistors (for the parallel pull-up network) are instead placed on two separate P-diffusion strips, whose drains are both tied together by a common metal line, realizing the parallel connection.

CMOS NOR2 stick diagram: The mirror-image arrangement of the NAND2: the two PMOS transistors share a single continuous P-diffusion strip (series pull-up), while the two NMOS transistors are placed on separate N-diffusion strips joined by metal at their drains (parallel pull-down).

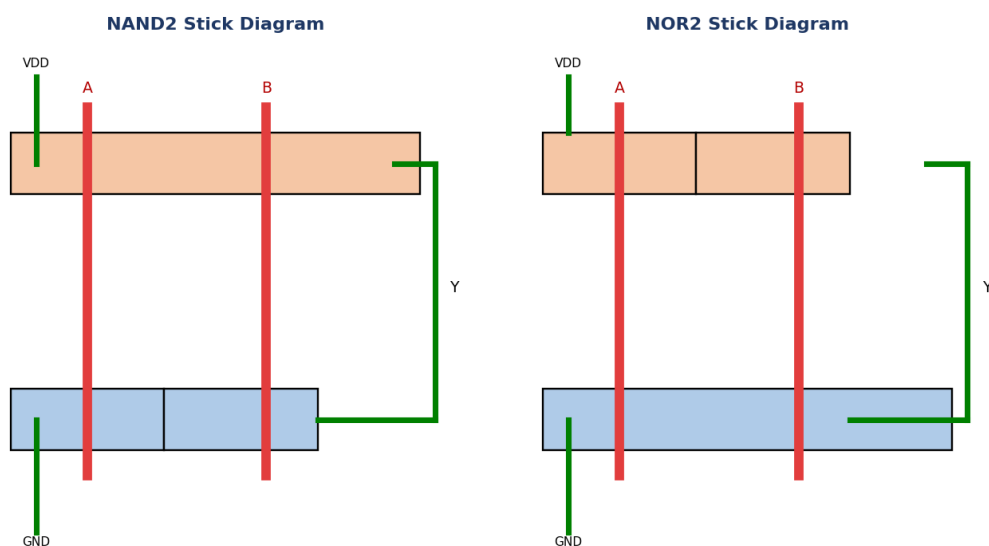


Fig: Stick diagrams - CMOS NAND2 (left) and NOR2 (right)

This general principle - series-connected transistors of the same type share one continuous diffusion strip crossed by multiple poly gates, while parallel-connected transistors of the same type sit on separate diffusion strips joined by metal -

is the key layout insight that stick diagrams are specifically meant to make visually obvious, before the design proceeds to full, dimensioned mask-layout.

Q4. Explain in detail micron-based design rules.

Micron-based (also called absolute) design rules specify every minimum layout dimension - minimum width, minimum spacing, minimum overlap/enclosure between different layers, minimum extension - as an exact numerical value in micrometers (μm) or, in modern deep-submicron processes, in nanometers (nm), rather than as a multiple of a single scalable unit λ as in lambda-based rules.

Key characteristics of micron-based rules:

- Process-specific: the numerical values are tied directly and precisely to the actual physical capability of a particular fabrication process/technology node (e.g. 0.18 μm , 90 nm, 45 nm) offered by a specific semiconductor foundry, and are published by that foundry as part of its Process Design Kit (PDK).
- Independent rules per layer: unlike lambda rules, where a single λ multiple (e.g. 2λ , 3λ) is commonly reused across several different layers, micron-based rules give each individual layer (metal-1, metal-2, poly, diffusion, via, etc.) its own independently-optimized minimum width and spacing value, reflecting the true, different lithographic/etch capability of each specific layer and process step.
- Layer-pair-specific overlap/enclosure rules: rules such as 'minimum contact enclosure by metal-1' or 'minimum poly extension beyond active area' are each given their own specific micron value, rather than a single generic overlap rule applied uniformly.

Advantages over lambda-based rules:

- Higher area efficiency: because each layer's rule exploits the exact, true minimum capability of that specific process step (rather than being generalized to one conservative λ value used everywhere), a layout drawn with micron-based rules can typically be made smaller (denser) than an equivalent layout constrained by lambda rules.
- Accuracy and up-to-date fidelity to actual fabrication capability: as a process is refined/improved over its production life, the foundry can update individual micron-based rules for specific layers to reflect real, measured process capability improvements, without needing to redefine a single global scaling unit.

Disadvantages compared to lambda-based rules:

- Not portable across different processes/technology nodes: a layout designed to one foundry's micron-based rule set generally cannot be directly reused on a different process or a different foundry without significant redesign and re-verification, since the absolute dimensions and rule relationships differ.
- More complex rule decks: because rules are specified independently per layer and per layer-pair (rather than as multiples of one number), the design-rule-check (DRC) rule deck is considerably larger and more complex to develop, verify and maintain.

Industrial usage: Virtually all modern, real-world ASIC and full-custom IC design is carried out using micron-based (absolute) design rules, supplied by the foundry as part of a comprehensive Process Design Kit (PDK) that also includes device models (SPICE parameters), standard-cell libraries, and DRC/LVS rule decks - lambda-based rules today are used mainly as a simplified teaching tool for introducing the underlying concept of layout design rules, rather than for actual chip fabrication.

Q5. Write in lambda rules with diagram.

(This question builds directly on the lambda-rule concept explained in Q1 above - refer to that answer for the full definition, diagram, and significance of lambda rules; a representative, more complete numerical rule table is given here.)

Lambda (λ) Based Design Rules

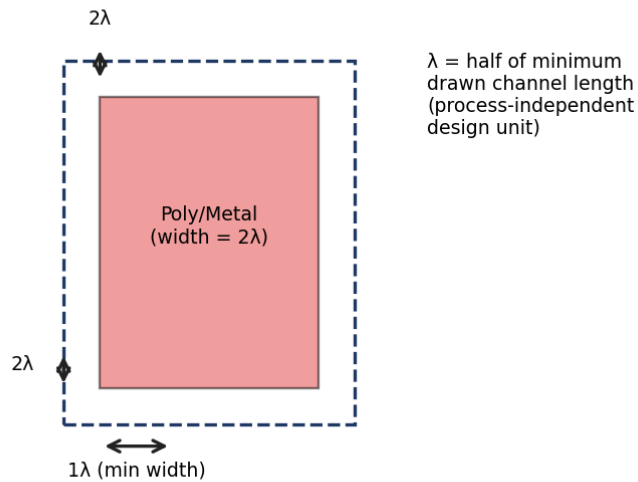


Fig: Lambda-based layout dimensioning (λ = half of minimum drawn channel length)

Representative lambda design rule table (values are typical/illustrative, expressed strictly as multiples of λ):

Layer / Feature	Rule (in λ)
Minimum poly (gate) width	2λ
Minimum poly-to-poly spacing	2λ
Minimum diffusion (active area) width	3λ
Minimum diffusion-to-diffusion spacing	3λ
Poly overlap of diffusion (forms transistor gate correctly)	2λ
Diffusion overlap of poly / gate-to-source-drain edge extension	2λ
Minimum metal-1 width	3λ
Minimum metal-1 to metal-1 spacing	3λ
Contact/via size	$2\lambda \times 2\lambda$
Contact overlap required by metal (enclosure)	1λ
Contact overlap required by diffusion/poly (enclosure)	1λ
Minimum spacing, contact to poly gate edge	2λ
N-well to P-diffusion (of an NMOS) spacing	6λ (typical, process dependent)

How these are applied (worked description): When laying out a transistor, the poly gate line must be drawn with a minimum width of 2λ (this directly sets the minimum realizable channel length $L = 2\lambda$, which is exactly why λ itself is defined as $L_{\min}/2$). The diffusion region must extend at least 2λ beyond each edge of the crossing poly gate, to provide adequate source/drain contact area and to guarantee correct transistor formation despite mask misalignment tolerances. Any two adjacent, unconnected features on the same layer (e.g. two separate poly lines, or two separate metal lines) must be separated by at least the specified minimum spacing (commonly 2λ - 3λ) to prevent unintended shorting due to the finite resolution of the photolithography process. Contacts/vias require a minimum enclosure (overlap) by the layers they connect, to guarantee reliable electrical connection even under worst-case mask alignment error.

Q6. Explain Parasitic extraction. Explain Crosstalk in detail.

Parasitic Extraction: After the full physical layout of a chip has been completed (placement and routing finished), the actual physical geometry of every wire - its exact width, thickness, length, spacing to neighbouring wires, and the specific metal layer it is routed on - is now precisely known. Parasitic extraction is the automated process (performed by a dedicated extraction tool, e.g. an RC extractor or a full 3-D field-solver-based extractor) of analyzing this real, physical geometry and computing the resulting parasitic resistances (R), capacitances (C), and, for high-speed nets, parasitic inductances (L), that were not present in the original, ideal pre-layout schematic/netlist.

- The extracted parasitics are typically written out into a standard parasitic-netlist format such as SPEF (Standard Parasitic Exchange Format) or DSPF (Detailed Standard Parasitic Format).
- This parasitic netlist is then back-annotated onto the original logical netlist, and used to re-run Static Timing Analysis (STA), power analysis, and signal-integrity (crosstalk/noise) analysis with real, physical, 'post-layout' accuracy - since in deep-submicron technology, interconnect parasitics frequently dominate over intrinsic gate delay, so a design that looked perfectly correct and fast using only ideal, pre-layout (schematic-level) timing can fail to meet timing once real parasitics are included, making parasitic extraction and post-layout re-verification (STA/power/SI sign-off) an absolutely mandatory step in any modern physical design flow.
- Parasitic extraction accuracy versus runtime is a key trade-off: simplified 2-D/lumped RC extraction is fast but less accurate, while full 3-D field-solver extraction is highly accurate but computationally expensive - modern flows often use fast extraction for early, iterative optimization and full extraction for final sign-off.

Crosstalk (Detailed explanation):

Crosstalk is unwanted electrical coupling between two physically adjacent interconnect wires that run close together and roughly parallel for some distance, caused by the parasitic coupling capacitance (and, at high frequency, mutual inductance) that necessarily exists between any two nearby conductors. The wire whose switching activity causes the disturbance is called the 'aggressor', and the wire that is disturbed is called the 'victim'.

Two distinct crosstalk problems:

- (a) Crosstalk glitch (noise injection): if the victim net is in a steady, non-switching (static) state while a nearby aggressor switches, the coupling capacitance injects a temporary, unwanted voltage bump (glitch) onto the victim net. If this glitch is large enough and occurs at a sensitive moment (e.g. right at a clock edge, disturbing a flip-flop's data or clock input), it can be mistakenly interpreted as a real logic transition, causing a functional error.
- (b) Crosstalk-induced delay variation (speed-up / slow-down): if the victim net itself is also switching at roughly the same time as a nearby aggressor, the coupling capacitor either helps or hinders the victim's own charge/discharge current depending on the relative switching direction. If the aggressor switches in the SAME direction as the victim, the coupling capacitor effectively 'helps' (charge is pushed in the same direction), speeding up the victim's transition (reducing its delay); if the aggressor switches in the OPPOSITE direction, the coupling capacitor 'fights' the victim's transition, effectively increasing the load capacitance the victim must charge/discharge against, slowing it down (increasing its delay). This delay uncertainty must be included as a 'crosstalk delta delay' in timing analysis, since a nominally fast path could become the actual critical path under worst-case crosstalk conditions.

Mitigation techniques for crosstalk:

- Increase the physical spacing between adjacent, parallel-routed wires, directly reducing the coupling capacitance between them.
- Insert dedicated shielding wires (tied to a constant VDD or GND) between two sensitive, closely-routed signal nets, which absorbs/blocks the coupling field.
- Assign particularly sensitive nets (e.g. clock nets, analog signals) to different, non-adjacent metal layers, or route them with extra spacing/shielding rules specifically applied to those nets.
- Insert buffers along a long net to restore signal strength and reduce the relative impact of coupled noise.

- Use crosstalk-aware routing tools during physical design, which specifically try to avoid long, parallel-routed segments between nets identified as timing-critical or noise-sensitive.

Q7. Explain Static Timing Analysis.

Static Timing Analysis (STA) is a method for verifying that a digital circuit meets all its required timing constraints, by exhaustively analyzing the delay along every possible signal path in the design - from every primary input or flip-flop output, through combinational logic, to every primary output or flip-flop input - and comparing each path's actual delay against the timing requirement imposed by the clock. It is called 'static' because it examines the circuit's structure and timing directly, without applying any actual input test vectors or performing time-domain (dynamic) simulation, which makes it exhaustive (covers every path, unlike simulation which only covers the specific vectors applied) and computationally very fast even for extremely large designs.

Two fundamental timing checks performed at every flip-flop (register-to-register path):

(a) Setup check: Ensures that data arriving at the input (D) of the destination (capturing) flip-flop becomes stable sufficiently EARLY before the next active clock edge, so the flip-flop can reliably and correctly capture it. The relevant condition is: $\text{Data Arrival Time} \leq \text{Data Required Time}$, where the required time is set by $(\text{clock period} + \text{destination clock arrival time} - \text{setup time of the flip-flop})$. The margin between them, $\text{Slack} = \text{Required Time} - \text{Arrival Time}$, must be ≥ 0 for the path to meet timing; a negative slack indicates a setup violation, meaning the design's clock frequency must be reduced, or the path must be sped up (e.g. through gate resizing, buffer insertion, or logic restructuring), for the design to function correctly at the intended clock speed.

(b) Hold check: Ensures that data at the destination flip-flop's input does NOT change too soon after the relevant (capturing) clock edge, so that the flip-flop, which requires the data to remain stable for a brief hold-time window after the clock edge, correctly captures the intended value rather than a new, incoming value that arrived too quickly (a 'race-through' condition). Hold violations, unlike setup violations, cannot be fixed simply by slowing down the clock (since hold timing is independent of clock period) - they must be fixed structurally, typically by inserting delay buffers into the offending (too-fast) path.

Other key STA concepts:

- Clock latency/insertion delay: the time taken for the clock signal to travel from its source to a given flip-flop's clock pin, through the clock distribution network.
- Clock skew: the difference in clock arrival time between the source and destination flip-flops of a given timing path (discussed in detail in Assignment 2, Q6), which directly adds to or subtracts from the setup/hold timing margin.
- Critical path: the specific signal path in the entire design with the least (most negative, or smallest positive) slack - this single path effectively determines the maximum achievable clock frequency of the whole chip, since the clock cannot be sped up beyond what this path allows without a violation.
- False paths: paths that exist structurally in the netlist but can never actually be sensitized/activated together during real functional operation (e.g. two mutually-exclusive control signals) - these must be explicitly excluded (constrained) from timing analysis, otherwise STA would report a violation on a path that could never actually cause a real functional failure, leading to unnecessary, wasted design effort.
- Multi-cycle paths: paths that are deliberately allowed, by design intent, more than one clock cycle to complete (e.g. certain slow control paths) - these too must be explicitly constrained with the correct number of allowed cycles, otherwise STA would incorrectly and overly-pessimistically flag them as single-cycle violations.

Limitation of STA: Static Timing Analysis verifies ONLY the timing behaviour of a circuit - it does not, and cannot, verify functional/logical correctness (that job is done by separate functional/logic simulation and formal verification tools). Furthermore, STA's usefulness and accuracy strongly depend on correctly and completely specifying all timing constraints (clock definitions, false paths, multi-cycle paths, input/output delays); an incompletely or incorrectly constrained design can cause STA to either miss real timing violations or report a large number of spurious, non-real violations, both of which undermine the reliability of the timing sign-off process.

Assignment 6

Unit VI: Digital CMOS Circuits | Modelling Styles, Testability and ASIC Flow

Q1. Compare data flow, behavioral and structural modeling.

VHDL (and similarly Verilog) supports three fundamentally different styles of describing a digital circuit, each representing the design at a different level of abstraction. A designer's choice of modelling style affects code readability, ease of debugging, and how directly the code maps onto physical hardware. In practice, complex, real-world designs almost always use a MIXED style, combining all three within different parts of the same overall design - for example, dataflow/structural for datapath logic, and behavioral for control/FSM logic.

Parameter	Dataflow	Behavioral	Structural
Abstraction level	RTL - close to Boolean/logic equations	Algorithmic - close to a software program	Gate/component level - close to a schematic netlist
Key constructs	Concurrent signal assignment ($\leftarrow$), conditional (when/else), selected (with/select/when)	Sequential statements inside a process: if-elsif-else, case, for/while loop	component declaration, port map, generic map
Execution model	All statements execute concurrently; each re-triggers on any RHS signal change	Statements inside a process execute sequentially, like software; process itself runs concurrently with others	Purely describes interconnection; the actual behaviour comes from the instantiated sub-components
Readability for combinational logic	Very direct and compact for simple Boolean equations	Can be verbose for simple combinational logic, but very natural for complex conditional logic	Not typically used to describe primitive combinational logic directly
Readability for sequential/FSM logic	Awkward and unnatural - FSMs are difficult to express purely as concurrent equations	Most natural and widely used style for FSMs and complex sequential algorithms	Only describes interconnection of already-designed FSM/sequential sub-blocks
Hierarchy / reuse	Limited - typically single-level, not inherently hierarchical	Limited - typically single-level, focuses on one block's behaviour	Naturally hierarchical - designed specifically for building larger systems from smaller, pre-verified sub-modules
Best suited for	Multiplexers, decoders, simple adders/combinational logic expressed as equations	FSMs, algorithms, complex sequential/control logic, memory models	Top-level system integration, connecting multiple sub-blocks/IP cores together

Concluding remark: None of the three styles is universally 'best' - each is optimized for describing a particular kind of logic naturally and efficiently; skilled VHDL designers deliberately choose (or mix) the style that most naturally and clearly expresses the specific piece of hardware being described, since forcing an unnatural style (e.g. writing an FSM purely in dataflow style) typically produces code that is harder to read, harder to debug, and sometimes harder for the synthesis tool to optimize effectively.

Q2. Write code in VHDL for 4:1 multiplexer in three different modeling types.

A 4:1 multiplexer selects exactly one of four data inputs (I0, I1, I2, I3) to route to a single output Y, based on the value of a 2-bit select signal Sel. Below, the same 4:1 MUX functionality is implemented in all three VHDL modelling styles, to directly illustrate how the same logical function can be described very differently depending on the chosen abstraction level.

(a) Dataflow Modelling - using a Selected Signal Assignment:

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity MUX4x1 is
```

```

Port ( I0,I1,I2,I3 : in STD_LOGIC;
      Sel      : in STD_LOGIC_VECTOR(1 downto 0);
      Y        : out STD_LOGIC);
end MUX4x1;

architecture Dataflow of MUX4x1 is
begin
  with Sel select
    Y <= I0 when "00",
         I1 when "01",
         I2 when "10",
         I3 when others;
end Dataflow;

```

Here the selected signal assignment directly and compactly expresses the multiplexing function as a single concurrent statement, closely resembling the equation $Y = I0.Sel'.Sel' + I1.Sel'.Sel + I2.Sel.Sel' + I3.Sel.Sel$.

(b) Behavioral Modelling - using a process with a case statement:

```

architecture Behavioral of MUX4x1 is
begin
  process(I0,I1,I2,I3,Sel)  -- sensitive to all inputs
  begin
    case Sel is
      when "00" => Y <= I0;
      when "01" => Y <= I1;
      when "10" => Y <= I2;
      when others => Y <= I3; -- covers "11" and any undefined value
    end case;
  end process;
end Behavioral;

```

This style describes the same functionality as an algorithm ('check Sel, then pick the corresponding input') rather than as a direct Boolean equation; the sensitivity list must include every signal that can affect Y (all four inputs plus Sel), otherwise the simulation could produce an incorrect result (a common beginner mistake).

(c) Structural Modelling - built hierarchically from three 2:1 MUX components:

```

-- Assume a previously-designed and verified 2:1 MUX component:
-- entity MUX2x1 is Port(A,B,S: in STD_LOGIC; Y: out STD_LOGIC); end;

architecture Structural of MUX4x1 is
  component MUX2x1
    port(A,B,S: in STD_LOGIC; Y: out STD_LOGIC);
  end component;
  signal w0,w1 : STD_LOGIC; -- internal wires
begin
  M0: MUX2x1 port map(I0,I1,Sel(0),w0); -- 1st stage
  M1: MUX2x1 port map(I2,I3,Sel(0),w1); -- 1st stage
  M2: MUX2x1 port map(w0,w1,Sel(1),Y); -- 2nd stage
end Structural;

```

This style builds the 4:1 MUX as a small hierarchy: two 2:1 MUXes (each choosing between a pair of inputs based on the LSB, Sel(0)) feed a third 2:1 MUX (which chooses between their outputs based on the MSB, Sel(1)) - directly mirroring how such a circuit would actually be drawn as a schematic/gate-level interconnection, and demonstrating design reuse of a previously verified sub-component.

Q3. Design CMOS logic for $Y = A + BC + DE$. Calculate the area in terms of W.

Following the same systematic complex-gate design procedure explained in Assignment 4 (Q6): the pull-down NMOS network is designed to conduct (pull output LOW) exactly according to the structure of the given expression, and the pull-up PMOS network is then built as its logical dual (series $\leftrightarrow$ parallel exchanged).

Pull-down network (NMOS) for $Y = A + BC + DE$: NMOS(A) alone in one branch (realizing the 'A' term); NMOS(B) in series with NMOS(C) in a second branch (realizing the 'BC' AND term); NMOS(D) in series with NMOS(E) in a third branch (realizing the 'DE' AND term); all three branches are connected in PARALLEL with each other (realizing the overall OR between the three terms A, BC, and DE).

Pull-up network (PMOS) - the dual network: PMOS(A) alone, in series with, {PMOS(B) parallel PMOS(C)} in series with {PMOS(D) parallel PMOS(E)} - i.e. all three groups (the single PMOS(A), the parallel pair B||C, and the parallel pair D||E) are connected in SERIES with each other, exactly mirroring the parallel-to-series swap required by the duality rule.

Transistor count: This function has 5 distinct literals (A, B, C, D, E), and a static CMOS complex gate always requires exactly 2 transistors per literal (one NMOS + one PMOS) - so the total transistor count is 5 NMOS + 5 PMOS = 10 transistors.

W/L sizing and area estimate (in terms of unit width W, with unit NMOS width = W, taking a fixed reference channel length L for all transistors):

- NMOS(A): stands alone in its own parallel branch (no series stacking on that branch), so it only needs the unit reference width: width = W.
- NMOS(B), NMOS(C): in series with each other (2 transistors in series is equivalent, in resistance, to a single transistor of length 2L), so each individually needs double the unit width to compensate: width = 2W each.
- NMOS(D), NMOS(E): similarly in series with each other, so each also needs width = 2W each.
- Total NMOS width = W (A) + 2W (B) + 2W (C) + 2W (D) + 2W (E) = 9W.
- PMOS(A): alone, but now in SERIES with the other two parallel groups in the pull-up network - being a single transistor on that series path (not itself stacked with another PMOS), it needs only the standard unit PMOS reference width (2W, applying the usual 2:1 PMOS:N MOS mobility-compensation ratio, i.e. treating 'unit PMOS width' = 2W where unit NMOS width = W): width = 2W.
- PMOS(B), PMOS(C): in PARALLEL with each other in the pull-up network, so each individually needs only the unit PMOS reference width (no series-stacking penalty): width = 2W each.
- PMOS(D), PMOS(E): similarly in parallel with each other, so each also needs width = 2W each.
- Total PMOS width = 2W (A) + 2W (B) + 2W (C) + 2W (D) + 2W (E) = 10W.

Total layout area estimate: Approximate area $\propto$ (Total NMOS width + Total PMOS width) $\times$ L = (9W + 10W) $\times$ L = 19WL — this is a first-order estimate proportional to the sum of all transistor widths (for the given, fixed channel length L), representing the total 'silicon width' the gate occupies; it does not separately include additional routing, spacing and well/tap overhead that a real, fully-dimensioned mask layout would also require.

Q4. What are the effects of technology scaling on chip level design?

As VLSI fabrication technology scales to progressively smaller feature sizes (following trends historically described by Moore's Law and Dennard scaling), chip-level design is affected in numerous, often conflicting ways - some clearly beneficial, others introducing new design challenges that must be actively managed.

Positive (beneficial) effects:

- Higher transistor density: smaller transistors allow far more logic (more functionality, more cores, more cache/memory) to be integrated onto the same physical die area, directly enabling the continued growth in chip complexity and functionality generation after generation.
- Reduced intrinsic gate delay: shorter channel lengths and smaller capacitances generally allow individual transistors/gates to switch faster, supporting higher maximum clock frequencies.
- Lower dynamic power per transition: as predicted by constant-field scaling theory, dynamic switching power per individual gate decreases significantly with scaling ($P \propto C \cdot V^2$, and both C and V decrease with scaling).

- Lower cost per logic function: more transistors manufactured per wafer, at a similar or only moderately higher cost per wafer, generally reduces the manufacturing cost per individual transistor/gate over successive technology generations.

Negative / challenging effects that must be actively managed:

- Exponentially increasing sub-threshold leakage power: as threshold voltage V_T is scaled down (necessary to maintain adequate drive current at lower, scaled supply voltages), sub-threshold leakage current increases exponentially, causing static/leakage power to become an increasingly significant, and in the most advanced nodes even dominant, fraction of total chip power.
- Interconnect-dominated delay: while transistor (gate) delay continues to improve with scaling, wire resistance actually increases as interconnect cross-sectional area shrinks, while wire capacitance does not shrink proportionally - so overall chip delay becomes increasingly dominated by interconnect (wire) delay rather than intrinsic gate delay, requiring extensive use of repeater/buffer insertion along long wires.
- Worsened electromigration: the same current now flows through a proportionally smaller interconnect cross-section, increasing current density and directly worsening electromigration-related long-term reliability degradation (as discussed in Assignment 5, Q2).
- Increased process variation: as absolute feature dimensions shrink toward the scale of individual dopant atoms and lithographic wavelength limits, random dopant fluctuation and line-edge roughness cause proportionally larger variation in transistor characteristics (V_T , drive current) between nominally identical devices, which directly affects circuit matching, yield, and design margin requirements.
- Increased power density and thermal management challenges: even though power PER GATE decreases with scaling, the much higher density of gates packed into the same area means total power DENSITY (power per unit area) has, in practice (departing from ideal constant-field scaling assumptions), tended to increase in recent process generations, creating serious on-chip heat dissipation and thermal-management (hotspot) challenges.
- Increased significance of reliability/aging effects: hot-carrier injection, gate-oxide breakdown (time-dependent dielectric breakdown, TDDDB), and Negative Bias Temperature Instability (NBTI) all become proportionally more significant concerns at smaller geometries and must be explicitly accounted for in the design timing/voltage margins to guarantee reliable operation over the product's intended lifetime.

Chip-level design response to these effects: Modern chip-level design directly addresses these scaling challenges through techniques such as multi- V_T cell libraries (using higher- V_T , lower-leakage cells on non-critical paths and lower- V_T , faster cells only on critical paths), power gating and clock gating (switching off unused blocks), dynamic voltage and frequency scaling (DVFS), buffered/repeated interconnect design, and statistical/corner-based timing analysis that explicitly accounts for process variation.

Q5. Describe the following: a. BSDF b. JTAG

(a) BSDL / BSDF (Boundary Scan Description Language / Boundary Scan Description File): BSDL is a standardized subset of VHDL syntax, defined as part of the IEEE 1149.1 (JTAG) standard, used to formally and precisely describe a specific chip's boundary-scan architecture - including its complete pin list and pin types, the exact structure and bit-ordering of its Boundary Scan Register (BSR), its supported instruction set (which test instructions the device implements), and other TAP (Test Access Port) controller-related details. The resulting file (commonly given a .bsdl or .bsd extension) is referred to as a BSDF (Boundary Scan Description File).

- Purpose: automatic test generation and board-level test tools (e.g. for manufacturing/production testing) read a device's BSDL/BSDF file to automatically understand exactly how to control and observe that specific chip's pins through the boundary-scan chain, without requiring any manual, pin-by-pin programming by the test engineer.
- Because every JTAG-compliant chip's boundary-scan structure can differ (different number of pins, different register bit ordering, different supported instructions), the BSDL/BSDF file - supplied by the chip manufacturer - is what makes automated boundary-scan test generation and interconnect testing possible across a wide variety of different components on a single printed circuit board.

(b) JTAG (Joint Test Action Group) / IEEE 1149.1 Standard: JTAG is an industry-standard hardware test architecture and physical interface, originally developed to solve the problem of testing densely-packed printed circuit boards (where traditional physical 'bed-of-nails' probing became impractical as surface-mount and fine-pitch packaging became common), and now also very widely used for in-system programming and on-chip debugging of processors/FPGAs/CPLDs.

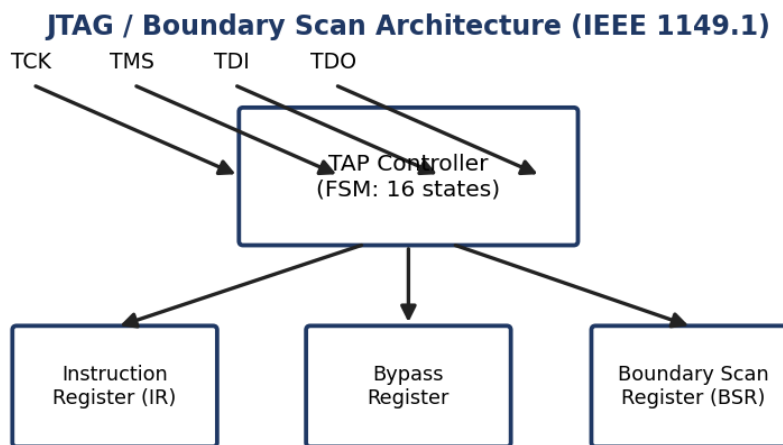


Fig: JTAG / Boundary-scan architecture

Key components of the JTAG architecture:

- **Test Access Port (TAP):** the standard 4-wire (or optionally 5-wire) physical interface consisting of TCK (Test Clock), TMS (Test Mode Select), TDI (Test Data In), TDO (Test Data Out), and an optional TRST (Test Reset).
- **TAP Controller:** a 16-state finite state machine, whose state transitions are controlled entirely by the TMS signal (sampled on each TCK edge), which sequences the device through its various test/scan operations (e.g. Capture-DR, Shift-DR, Update-DR states for manipulating the scan chain).
- **Instruction Register (IR):** a shift register that is loaded (via the TAP) with an instruction code selecting which specific test operation the device should perform next - common instructions include EXTEST (test external board interconnects using the boundary-scan cells), BYPASS (skip this device, minimizing chain length when testing a different device on the same board), and SAMPLE/PRELOAD (capture the current pin state without disrupting normal operation).
- **Boundary Scan Register (BSR):** a dedicated scan flip-flop (a 'boundary-scan cell') is placed directly at every single I/O pin of the chip, and all these cells are connected together to form one long shift register running around the entire periphery of the device. This allows an external tester to shift a known test pattern IN to every output pin (driving it to a controlled value) and shift the resulting captured value at every input pin OUT for observation, entirely through the single serial JTAG interface - enabling comprehensive board-level interconnect testing (detecting shorts/opens between chips) without any physical probe access to individual pins.

Other major applications of JTAG (beyond board test): In-System Programming (ISP) of CPLDs/FPGAs/Flash memory (as already discussed for the XC9500 CPLD architecture), and on-chip debug access for microprocessors/microcontrollers (allowing an external debugger to halt the CPU, single-step instructions, and inspect/modify internal registers and memory directly through the same JTAG port).

Q6. Explain Stuck-at-0 and stuck-at-1 faults.

After a chip has been manufactured, it must be tested to check whether any physical manufacturing defect has caused it to deviate from its intended, designed behaviour. Since it is practically impossible to directly model or test for every

conceivable physical defect (opens, shorts, bridging faults, parametric shifts, etc.) individually, VLSI testing relies on simplified, abstract LOGICAL fault models that approximate the observable effect of a wide range of underlying physical defects - by far the most widely used and fundamental of these is the Stuck-At Fault Model.

Stuck-At Fault Model: This model assumes that, due to some underlying physical manufacturing defect, a particular signal line in the circuit - which could be a gate's input, a gate's output, or any internal net - becomes permanently 'stuck' at a fixed logic value, and can no longer be correctly driven or changed by the circuitry that is actually supposed to control it, regardless of what the correct, fault-free logic value should have been at that point.

Stuck-at-0 (s-a-0) fault: The faulty signal line behaves as though it is permanently and unconditionally tied to logic '0', irrespective of what value the driving logic is actually trying to place on it. Physically, this typically models manufacturing defects such as an unintended short-circuit of that line to the ground (GND) rail, or a broken/open connection in a pull-up path that leaves the line permanently pulled low by a stray leakage/parasitic path.

Stuck-at-1 (s-a-1) fault: The faulty signal line behaves as though it is permanently and unconditionally tied to logic '1', regardless of the intended driving value. This typically models defects such as an unintended short-circuit of that line to the VDD (supply) rail, or an equivalent open/defect in a pull-down path.

Test generation for stuck-at faults (Automatic Test Pattern Generation - ATPG):

- To DETECT a suspected s-a-0 fault at a given internal line, a test engineer (or, in practice, an ATPG software tool) must find an input test vector that would, under normal fault-free operation, drive that specific line to a logic '1' (this step is called 'fault activation' or 'sensitization') - since only then would the presence of the s-a-0 fault (forcing it to stay at '0' instead) create an actual, observable difference from correct behaviour.
- The same test vector must also 'propagate' this difference all the way from the fault site to a primary output (or a scan flip-flop) where it can actually be observed and compared against the expected, fault-free value - this generally requires setting the other, unrelated inputs of any gates along the propagation path to values that allow the faulty signal's effect to pass through unmasked (this is called setting those gates to their 'non-controlling' input value).
- Similarly, to DETECT a suspected s-a-1 fault, a test vector must be found that would normally drive that line to a logic '0' (again activating and then propagating the resulting difference to an observable output).

Fault coverage: The overall effectiveness/quality of a given manufacturing test-vector set is quantified by its Fault Coverage, defined as:

$$\text{Fault Coverage} = (\text{Number of stuck-at faults successfully detected by the test set} / \text{Total number of possible stuck-at faults in the circuit}) \times 100\%$$

Achieving high fault coverage (typically targeting well above 95-99% in industrial practice) is essential to have confidence that a chip which passes the manufacturing test set is genuinely free of common physical defects; because achieving very high fault coverage purely with primary-input/primary-output test vectors becomes extremely difficult for large, deeply-sequential circuits, dedicated Design-for-Testability (DFT) techniques - most notably scan-chain insertion (converting flip-flops into shift-register-capable scan cells, closely related to the boundary-scan/JTAG concept discussed above) - are used to make internal nodes directly controllable and observable, dramatically improving achievable fault coverage.

Q7. Write short notes on: a. ASIC b. ASIC design flow.

(a) ASIC (Application Specific Integrated Circuit): An ASIC is an integrated circuit that is custom-designed and manufactured to perform one specific application or function for a particular customer/product, as opposed to general-purpose, off-the-shelf devices (like a standard microprocessor) or field-reprogrammable devices (like an FPGA/CPLD) that are designed to be flexible and usable across many different applications.

Key characteristics: ASICs offer the highest possible performance (speed), the lowest power consumption, and the smallest physical area for the specific target function, because every aspect of the design - down to individual transistor sizing and physical placement, in the case of full-custom design - can be optimized purely for that one application, with

no wasted, unused programmable/reconfigurable overhead (unlike an FPGA, which must include large amounts of generic, reconfigurable logic and routing resources that go unused once a specific function is implemented).

Trade-off: This high level of customization comes at the cost of very high Non-Recurring Engineering (NRE) cost (covering design effort, verification, and especially photomask/tooling costs for fabrication) and a long design-to-fabrication turnaround time; unlike an FPGA, an ASIC's function is fixed once fabricated and CANNOT be changed or corrected afterward (any bug requires an expensive and slow re-spin/re-fabrication) - so ASICs are economically justified primarily for very high production-volume products, where the high upfront NRE cost is amortized (spread) over a very large number of manufactured units, making the cost PER CHIP very low.

Broad categories of ASIC:

- Full-custom ASIC: every transistor is individually designed and laid out by hand for maximum possible performance/density, at the cost of the highest design effort and time - used only for extremely high-volume or performance-critical products.
- Standard-cell based ASIC: built by selecting and interconnecting pre-designed, pre-characterized logic cells (e.g. NAND, NOR, flip-flops) from a standard-cell library, with automated placement and routing tools - offers a good balance of design effort/time versus performance/density, and is by far the most common approach used in modern industrial ASIC design.
- Gate-array based ASIC: uses a wafer that is partially pre-fabricated with a regular, un-connected array of transistors; only the final interconnect (metal) layers are customized for the specific design - offering faster turnaround and lower NRE cost than full-custom or standard-cell design, at some cost in density/performance, though this approach is less common today compared to standard-cell design.

(b) ASIC Design Flow:

ASIC Design Flow

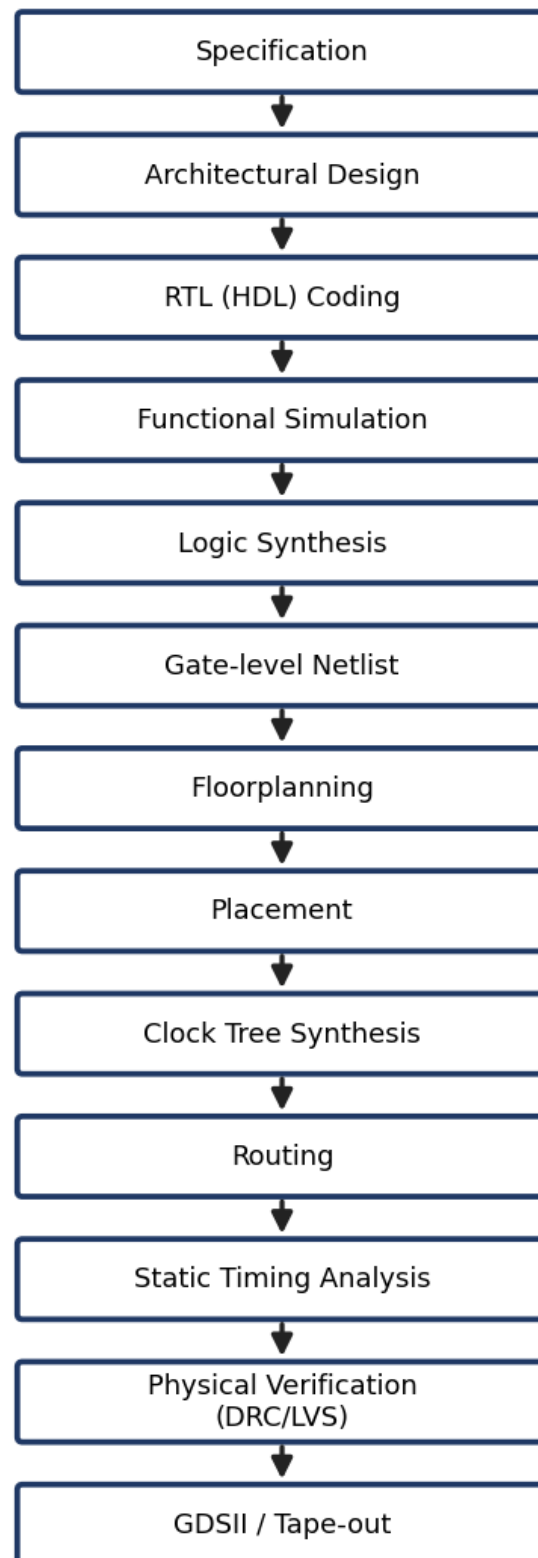


Fig: ASIC design flow

- 1. Specification and architectural design:** The overall functionality, performance targets, power budget, and interface requirements of the chip are defined, and the high-level micro-architecture is planned.
- 2. RTL coding and functional simulation:** The design is described in an HDL (VHDL/Verilog) at the Register-Transfer Level, and extensively simulated against a comprehensive test bench to verify functional correctness before any further, more expensive design steps are undertaken.
- 3. Logic synthesis:** An automated synthesis tool translates the verified RTL code into an optimized, technology-mapped gate-level netlist, built entirely from cells available in the target standard-cell library, while meeting specified timing, area and power constraints.
- 4. Floorplanning, placement, and routing (physical design):** The overall chip floorplan (arrangement of major blocks, I/O pads, power distribution) is planned; individual standard cells are placed within the floorplan (placement); Clock Tree Synthesis (CTS) builds the balanced clock distribution network (as discussed in Assignment 2, Q2); and finally the actual metal interconnect wiring between all cells is completed (routing).
- 5. Static Timing Analysis (STA) and power analysis:** After physical design (with real, extracted parasitics back-annotated, as discussed in Assignment 5, Q6), the design is thoroughly re-verified to ensure it meets all timing constraints (setup/hold, as discussed in Assignment 5, Q7) and stays within its power budget - this is often called 'timing/power closure'.
- 6. Physical verification:** Design Rule Check (DRC) confirms that the final layout obeys every geometric manufacturing rule of the target process (as discussed in Assignment 5, Q1 & Q4); Layout-Versus-Schematic (LVS) confirms that the actual, extracted transistor-level connectivity of the final layout exactly matches the intended logical netlist, with no unintended shorts, opens, or missing connections.
- 7. GDSII generation and tape-out:** The final, fully verified layout is exported in the industry-standard GDSII (Graphic Data System) file format and formally handed off ('taped out') to the semiconductor foundry, which uses it to generate the photomasks required for actual chip fabrication.