ATME College of Engineering, Mysuru
Department of Electronics and Communication

VLSI Design and Testing(BEC602)

Structured Design and Testing

1 Introduction

1.1 Introduction to VLSI Design Styles

- Integrated circuit (IC) design can be described using three main **design domains**:
  1. **Behavioral Domain** – Describes what the system *should do*.
  2. **Structural Domain** – Describes how the system is *logically constructed*.
  3. **Physical Domain** – Describes how the system is *physically implemented*.
- Each domain provides significant **design flexibility**:
  1. In the **behavioral domain**, designers can choose between sequential or parallel algorithms.
  2. In the **structural domain**, decisions include logic families, clocking strategies, and circuit styles.
  3. In the **physical domain**, implementation options range from chips to boards and cabinets.
- These domains are organized hierarchically into several **levels of abstraction**:
  1. Architectural or functional level
  2. Register Transfer Level (RTL)
  3. Logic level
  4. Circuit level
- Each level provides multiple implementation options and supports a variety of design styles.
- Here, we are mainly going to focus on **popular design styles**, and **structured hierarchical design**.

1.2 Design Flow Approaches

1. **Contemporary Design Flow(Figure a):**
   - Follows a step-by-step approach through abstraction levels.
   - Requires **verification** of equivalence between each level (e.g., from behavior to RTL, RTL to logic).

2. **Ideal Design Flow(Figure b):**
    - Fully synthesizes the system from a high-level specification.
    - Minimizes manual intervention and verification efforts.

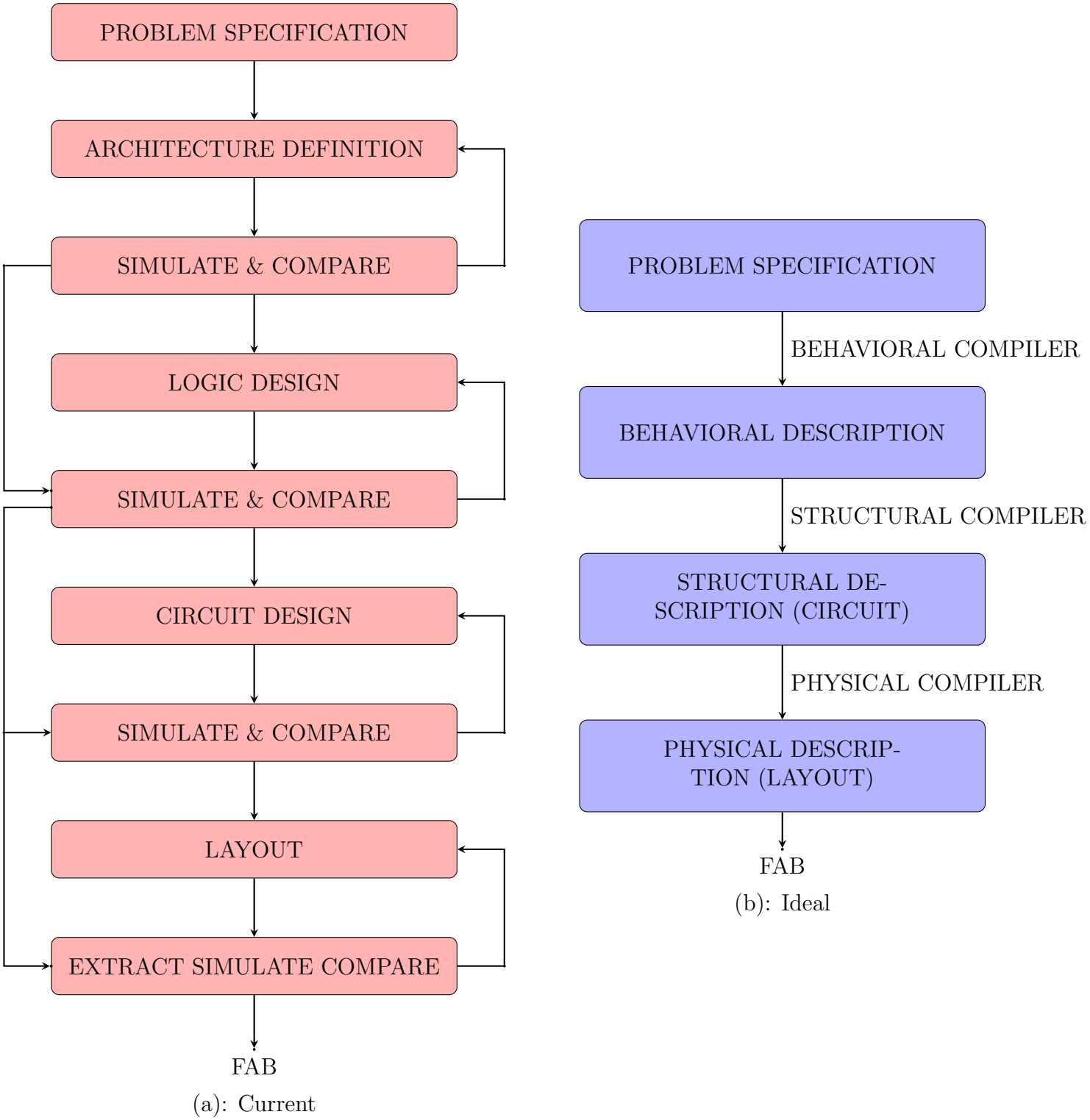


Figure 1: Current and ideal design approaches

2 Design Styles

2.1 Introduction

A good VLSI design system should provide consistent descriptions across:

- All three design domains: behavioral, structural, and physical
- All relevant levels of abstraction

The effectiveness of a design system can be measured using various **design parameters**:

- **Performance** — speed, power, functionality
- **Size of die**
- **Time to design** and **ease of use**
- **Ease of test generation** and overall **testability**

Design is a process of making trade-offs to achieve acceptable results across all these parameters. Therefore, the **tools and methodologies** selected for a particular chip depend on:

- Required performance specifications (e.g., speed or power)
- Economic factors (e.g., die size and yield)
- Designer preferences or familiarity (subjective factors)

Given the complexity of silicon-based system design, the role of VLSI design aids is to:

- Reduce design complexity
- Ensure the designer achieves a working product

2.1.1 Simplifying Design Through Constraints and Abstractions

- **Constraints** help automate the design process, reducing manual effort
- **Abstractions** help manage complexity by hiding lower-level details

Ideally, the choice of methodology is **economically driven**, provided all styles are available.

Key considerations:

- Functionality determines which methods are suitable
- Some styles may be unsuitable due to layout inefficiencies
- Die cost is estimated after choosing the style
- The fastest method to achieve that cost-effective design is preferred

2.2 Structured Design Strategies

- One of the main goals of the Mead and Conway approach to VLSI design was to empower system designers to implement high-performance systems **directly in silicon**.
- To achieve this, it is essential to **reduce the complexity** of IC and system-level design. This is especially important because:
  - Expert industrial teams require **man-years** to complete chip designs.
  - Expecting novice or academic teams to match this effort without simplification is unrealistic.

Adapting Software Design Methods

The field of software engineering has developed effective strategies for managing large, complex systems. These can be:

- **Adapted or re-adapted** to the VLSI design domain
- Used to make the IC design process more accessible to novices
- Utilized by experts to manage growing complexities — especially with circuits having **millions of devices**

2.2.1 Hierarchy

Hierarchy is a fundamental strategy used in structured VLSI design. It involves:

- Dividing a complex module into **submodules**
- Repeating this subdivision recursively until each submodule is simple enough to be easily understood and implemented

This strategy is similar to practices in software design, where:

- Large programs are decomposed into smaller components
- Eventually yielding simple **subroutines** with well-defined functions and interfaces

Parallel Hierarchy in VLSI Design Domains

Recall that VLSI design can be described in three main domains:

1. Behavioral
2. Structural
3. Physical

Hierarchy can be applied **in parallel** within each domain:

- **Behavioral domain**: An adder may be represented by a subroutine describing its operation
- **Structural domain**: A gate-level connection diagram specifies how it is built logically
- **Physical domain**: A layout section describes its actual physical structure

Consistency Across Domains

As larger structures are built from smaller ones (e.g., combining multiple adders), this hierarchical approach can be extended in all three domains. Throughout this process:

- **Domain-to-domain comparisons** are essential to ensure that each representation (behavioral, structural, and physical) is consistent with the others

2.2.2 Modularity

Modularity is a key principle in structured VLSI design. It complements hierarchy by dividing a system into a set of **submodules** that are easier to understand, manage, and implement.

Characteristics of a "Well-Formed" Module

The concept of a "well-formed" module varies, but we can borrow principles from structured software design. A well-formed module should have:

- A **well-defined interface** — In software, this is an argument list with types. In VLSI, it includes:
  - Position and name of external connections
  - Layer type and size
  - Signal type (input/output, power/ground)
- A **clearly defined function** — The module's purpose must be unambiguous.

Modularity benefits both the designer and the design system by:

- Clarifying and documenting the design strategy
- Enabling automated checking of module attributes
- Supporting **team-based design**, where multiple designers work on different parts of the chip

Programming Analogies in VLSI Design

Structured programming relies on three basic constructs:

1. **Concatenation**
2. **Iteration**
3. **Conditional selection**

These have parallels in IC design:

- **Concatenation** → *Cell abutment*: IC cells are placed adjacent to one another, with connections formed on shared boundaries.
- **Iteration** → *Arrays of cells*: One- or two-dimensional arrays of identical cells (e.g., memory arrays).
- **Conditional selection** → *Programmable Logic Arrays (PLAs)*: Logic behavior is defined by the placement of transistors in an array.

Parameterization and Modularity

When these programming constructs are combined with **parameterization** (e.g., setting sizes or bit widths via parameters), the design becomes highly modular, reusable, and scalable.

2.2.3 Regularity

Regularity is another key principle in structured VLSI design. It refers to the repeated use of uniform structures to simplify the design, verification, and manufacturing processes.

Example: Iterative Arrays

The formation of **arrays of identical cells**, such as those used in memory or arithmetic units, is a basic form of regularity. It leverages repetition to reduce complexity.

Regularity Across Design Levels

Regularity can be applied at various levels in the design hierarchy:

- **Circuit Level**:
  - Use of **uniform transistor sizes and types** instead of individually optimized devices.
- **Logic Module Level**:
  - Use of **identical gate structures** throughout the design.
- **Architecture Level**:
  - Replication of **identical processor elements** or functional units in large systems.
- **Data-path Construction**:
  - Modules may differ internally by function, but share a **common interface structure** (e.g., power, ground, clock, buses).

Benefits of Regularity

- Simplifies design and layout
- Improves manufacturability and yield
- Facilitates verification and testing
- Supports **correct-by-construction** design methodology
- Aids formal verification methods

2.2.4 Locality

Locality is a design principle that supports modularity and hierarchy by reducing interdependence between modules.

Encapsulation and Information Hiding

When a module has a **well-characterized interface**, the internal implementation details become irrelevant to external modules. This technique, known as **information hiding**, reduces complexity by:

- Isolating design changes to individual modules
- Minimizing external impact from internal changes
- Encouraging self-contained module development

This concept mirrors **software engineering** practices, where the use of global variables is discouraged to preserve locality and reduce side effects.

Implications for Physical Design

- Physical design should avoid **overlapping connections** with a module's internals to preserve its integrity.
- Module interaction should be limited to defined interfaces only.

Placement Strategy

Modern design approaches often favor:

- **"Wires first, then modules"** — global wiring paths are planned before module placement, ensuring shorter and more efficient interconnections.
- This is in contrast to older methods of **"place modules, then route"**, which can result in congested or suboptimal wiring.

Benefits of Locality

- Reduces global interconnections
- Improves layout efficiency
- Enhances reusability and maintainability
- Enables scalable and systematic VLSI design

2.3 Handcrafted Mask Layout

Handcrafted mask layout refers to custom, unconstrained layout techniques where the designer manually creates the physical layout at the mask level.

2.3.1 Characteristics

- Traditional and still widely used, especially by semiconductor vendors.
- Involves direct layout of functional subsystems at the physical level.

- Typically handled by designers with specialized knowledge in logic design, circuit behavior, and fabrication processes.

2.3.2 Evolution of Layout Techniques

- Early methods: Cutting **RUBILITH** and drawing on **MYLAR**.
- Intermediate stage: Digitization of drawn layouts.
- Modern methods: **Interactive graphical layout editors**.

2.3.3 Advantages

- Allows **optimization at the transistor level**.
- Can result in **high-performance** and **area-efficient** layouts.

2.3.4 Challenges

- Time-consuming and complex, especially for VLSI circuits with millions of transistors.
- Requires significant **manual effort and expertise**.
- Lack of constraint can lead to inconsistencies between:
  - **Behavioral description**
  - **Structural specification**
  - **Physical layout**
- Before the development of **circuit extraction tools**, validating the layout was difficult.

2.3.5 Modern Tools

- **Circuit extraction tools** can now analyze the mask-level layout and reconstruct the equivalent circuit.
- Ensures consistency and aids in verification and simulation.

2.4 Gate Array Design

Gate arrays have gained widespread popularity as an LSI/VLSI implementation medium. This is due to a combination of factors including:

- Readily available vendors,
- Availability of robust design tools,
- Compatibility with TTL designs, enabling system designers to migrate from board-level to silicon implementations with minimal effort,
- Cost-effectiveness for certain classes of ICs.

The constrained physical layout of gate arrays allows the development of bounded design tools. While this does not necessarily reduce the complexity of the tools themselves, it makes the design space more manageable.

2.4.1 Structure of Gate Arrays

Gate arrays are characterized by a regular array of identical *sites*, with each site consisting of a set of circuit elements. In CMOS technology, these sites typically include multiple nMOS and pMOS transistors.

Examples of Site Structures:

- **Six-Transistor Site**: Contains 3 nMOS and 3 pMOS transistors. Gates are connected in common, and source/drain terminals are connected as shown in the schematic.

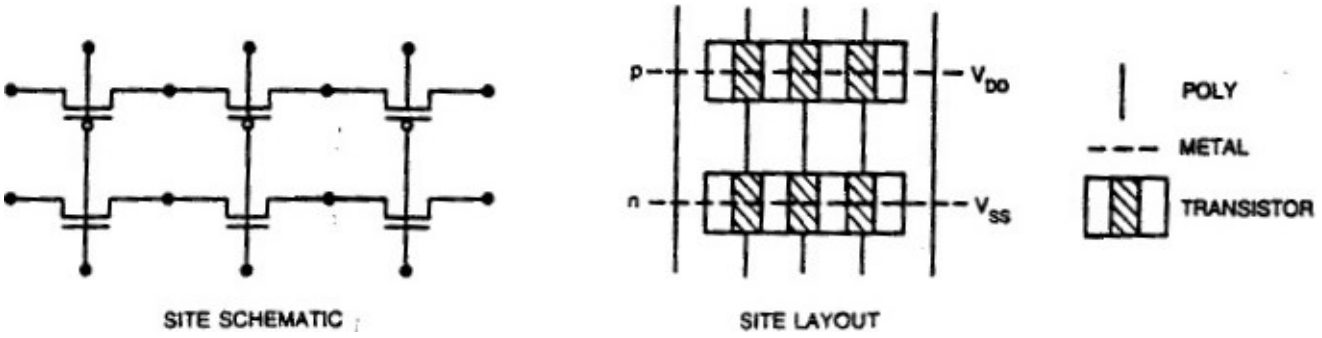


Figure 2: Six-Transistor Site

- **Layout View**: A physical layout corresponding to the six-transistor site, showing metal and polysilicon routing.

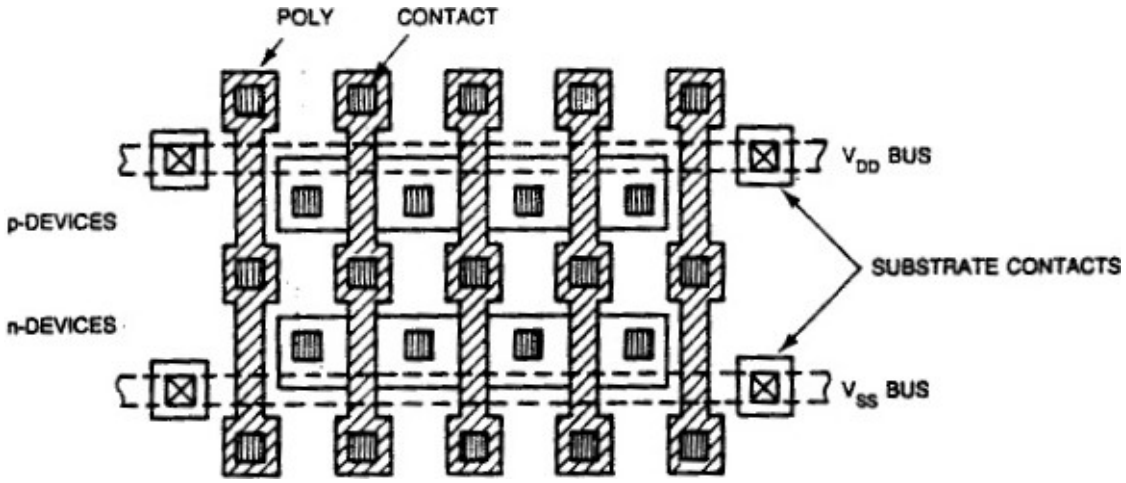


Figure 3: Six-Transistor Site:Layout View

- **Four-Transistor Site**: Includes two nMOS and two pMOS transistors. One n-p pair has a common gate connection; the other pair has separate gate signals. This configuration easily implements transmission gates and inverters, useful for latches.

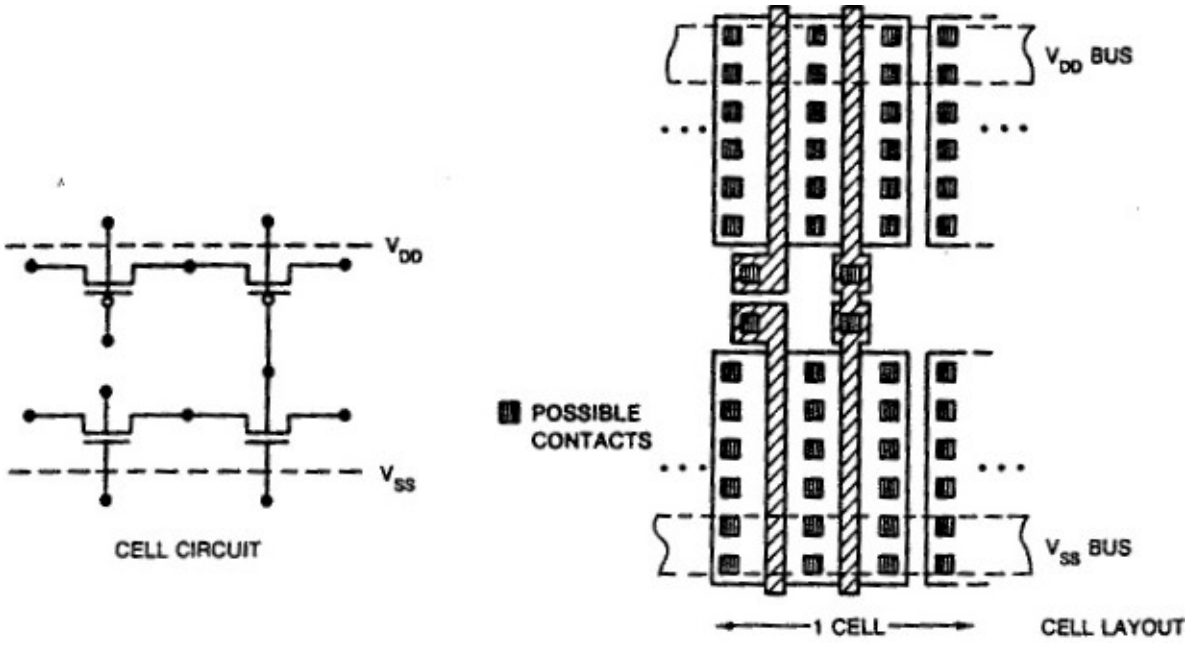


Figure 4: Four-Transistor Site

- **Static Latch Site**: Six transistors are ratioed specifically to implement static latches. This site is one of two cells used in a memory-logic gate array.

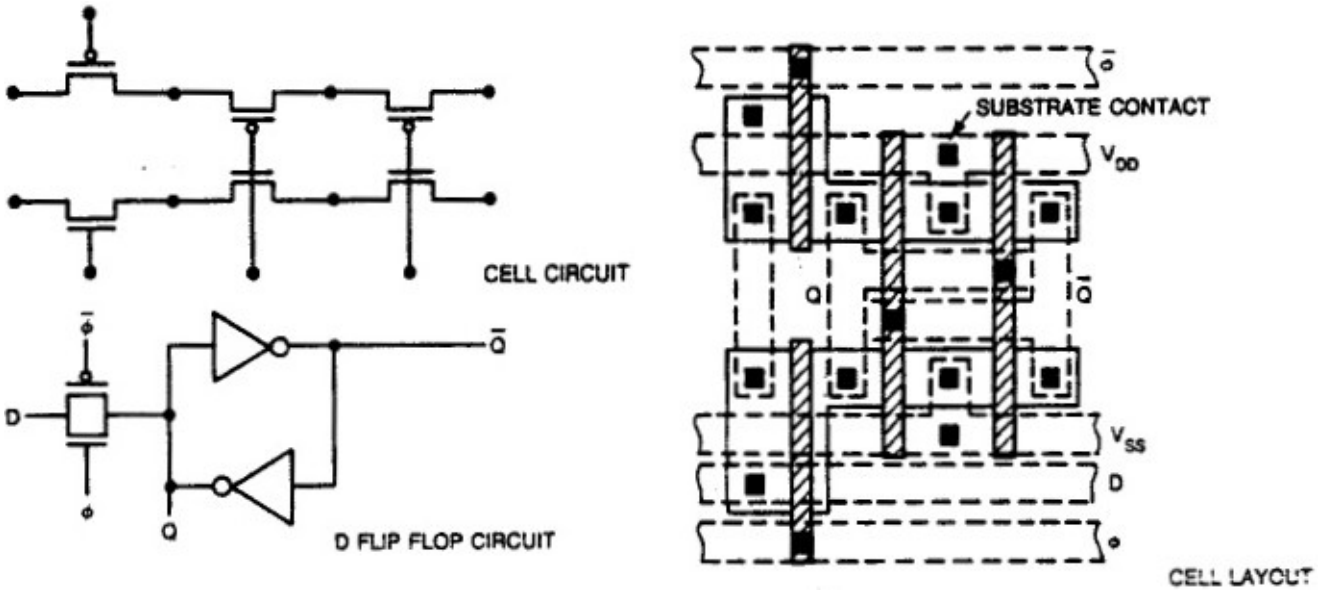


Figure 5: Static Latch Site

- **Metal-Programmable Site**: Features a continuous array of transistors with functionality defined entirely by metal interconnects, similar to the CMOS cell array method.

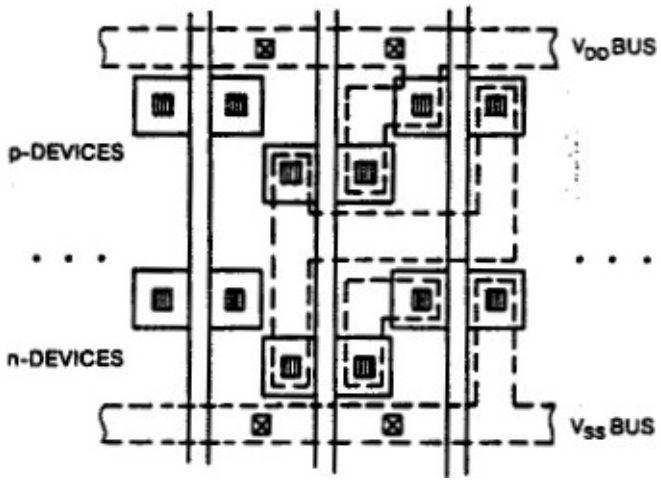


Figure 6: Metal-Programmable Site

2.4.2 Advantages of Gate Arrays

- **Predefined Topology**: Enables wafer stockpiling up to a certain fabrication step (usually metallization).
- **Post-Fabrication Personalization**: Final logic is defined by customizing metal layers, significantly reducing mask and fabrication cost.
- **Reduced Time-to-Market**: Design turnaround is faster since only a fraction of the usual mask cost (typically 1/8 to 1/4) is incurred.

2.4.3 Disadvantages and Trade-offs

- **Wasted Area**: All transistors are pre-fabricated; unused transistors occupy valuable chip area.
- **Fixed Placement and Configuration**: Limits the achievable optimization in area and performance.
- **Memory/Logic Ratio Constraints**: Vendors must estimate suitable ratios of RAM/ROM to logic blocks in advance.

2.4.4 Typical Gate Array Floorplan

- Arrays of sites are separated by routing channels.
- Directional routing rules are applied:
  - Metal layers typically run horizontally.
  - Polysilicon or other routing layers run vertically.
- I/O cells surround the logic core and are mask-programmable.

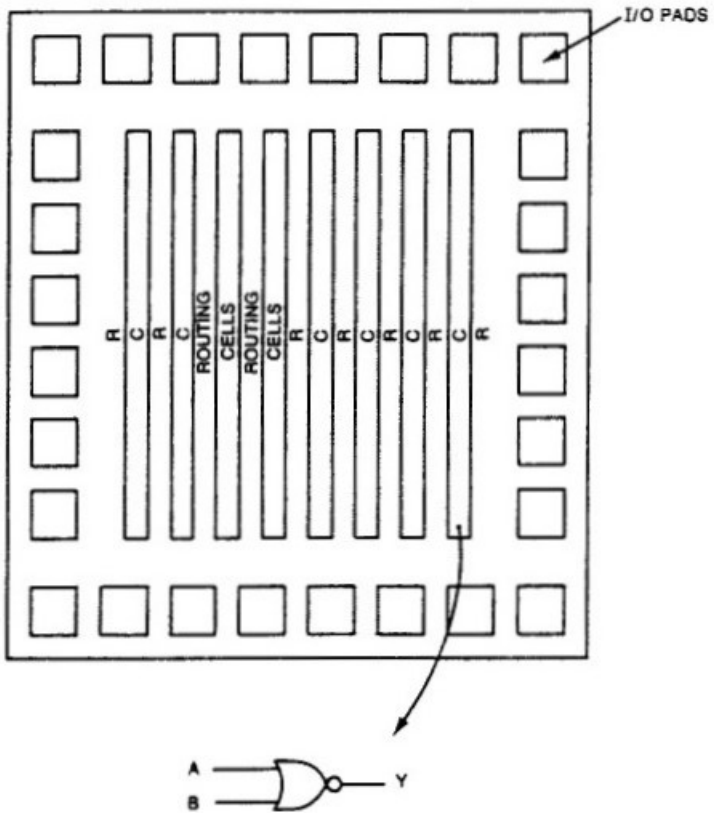


Figure 7: Typical Gate Array Floorplan

Example Cell: CMOS Site Structure

- A typical 6-transistor CMOS site can implement a variety of 3-, 2-, or 1-input logic gates.
- Routing:
  - Metal connections internal to the cell are used for logic.
  - External connectivity uses vertical metal and horizontal polysilicon runners.

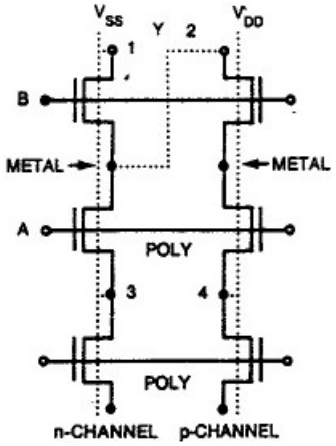


Figure 8: Typical 6-transistor CMOS site

2.4.5 Design Considerations

Designers must address the following during gate array implementation:

- Transistor sizing.
- Routing channel width,
- Placement and routing of discretionary wiring.

2.4.6 Design Flow Example: IGC-20000D

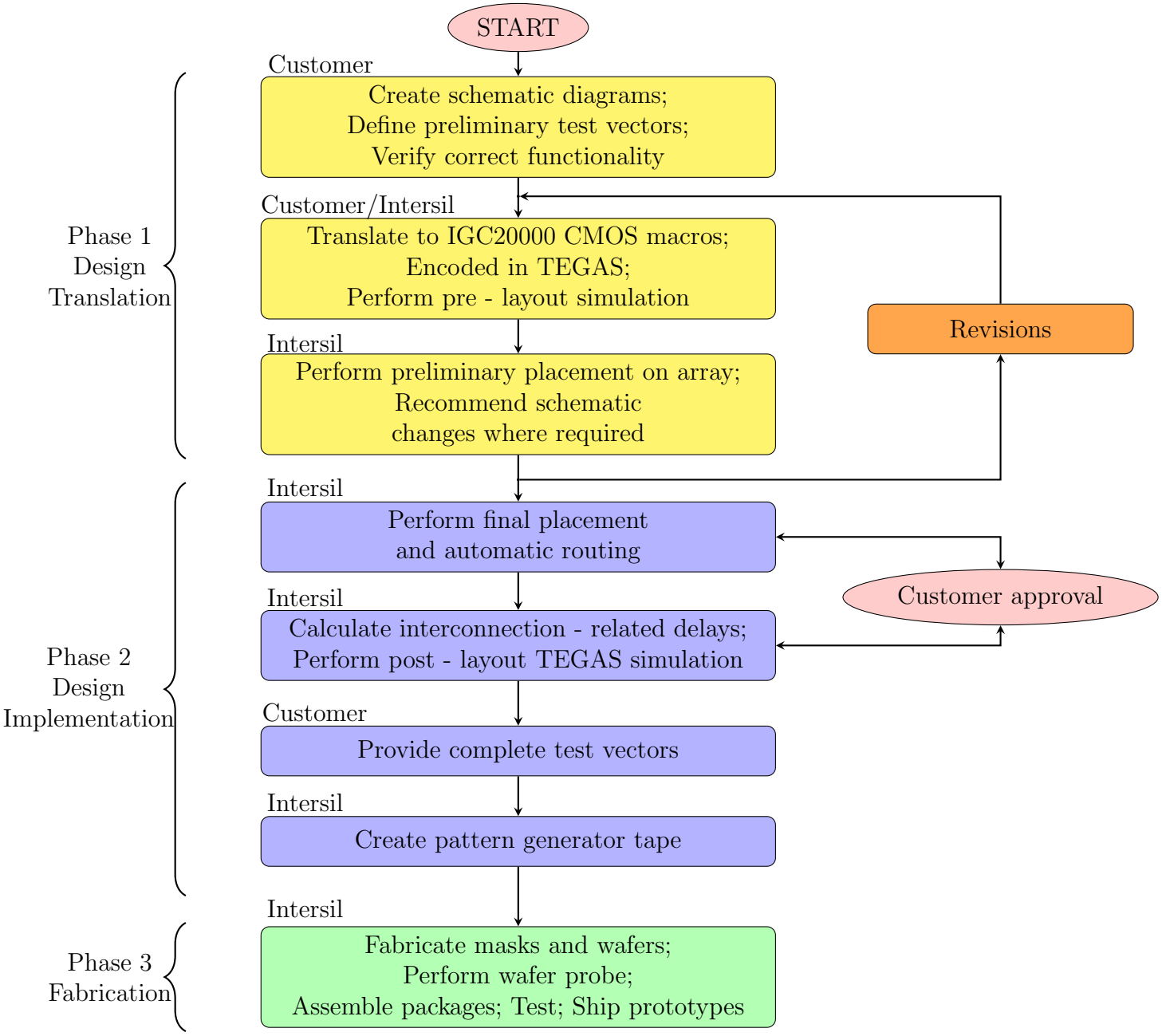


Figure 9: Gate array design flow

- Customer provides a logic schematic and test vectors.
- Logic is verified using test vectors.
- Logic schematic is converted to CMOS gate-array macros.
- Initial simulation is performed.
- Cells are placed and automatically routed.
- Necessary revisions are sent to the customer.
- Once finalized, a full simulation including parasitics is done.
- Final chip is manufactured and tested with customer test vectors.

Meaning of Intersil and TEGAS

Intersil: A semiconductor manufacturing company known for analog and mixed-signal ICs. In the diagram, "Intersil" refers to the internal design or fabrication team responsible for tasks like macro translation, simulation, placement, routing, and chip fabrication.

TEGAS: A design and simulation environment, possibly proprietary or domain-specific. TEGAS is used for pre-layout and post-layout simulation of CMOS designs to verify functionality before and after physical implementation.

2.5 Standard Cell Design

- Standard cell design is a method of VLSI implementation using a library of predefined logic and circuit cells.
- These cells enable efficient digital system design and are modular in nature, similar to TTL-based gate arrays.
- Designers familiar with ITL (Integrated TTL Logic) databooks can easily relate to this approach.

2.5.1 Types of Standard Cells

Standard cells vary in complexity:

- **SSI (Small Scale Integration)**: Basic gates and latches.
- **MSI/LSI (Medium/Large Scale Integration)**: More complex modules such as ROMs, RAMs, and PLAs.

2.5.2 Availability and Commercial Adoption

- Initially, standard cell systems were available only in large companies.
- Over time, commercial tools have made standard cell design accessible to broader markets, facilitating faster and more affordable custom chip design.

2.5.3 Mask Requirements

- Standard cell design typically requires a full mask set.
- However, design trade-offs can be introduced to reuse chips for various applications.
- For example, personalizing a ROM or control PLA block allows flexibility in functionality without redesigning the entire chip.

2.5.4 Comparison with Gate Arrays

Table 1: Comparison between Standard Cell and Gate Array Design

Feature	Standard Cell Design	Gate Array Design
Flexibility	Cells can be placed anywhere on the chip	Uses a fixed base array of cells
Customization	Full customization of all mask layers	Only upper layers are customized
Performance	Generally higher due to optimized layout	Lower performance

2.5.5 Floorplanning and Layout

A typical floor plan for a chip designed with standard cells is shown in figure below:

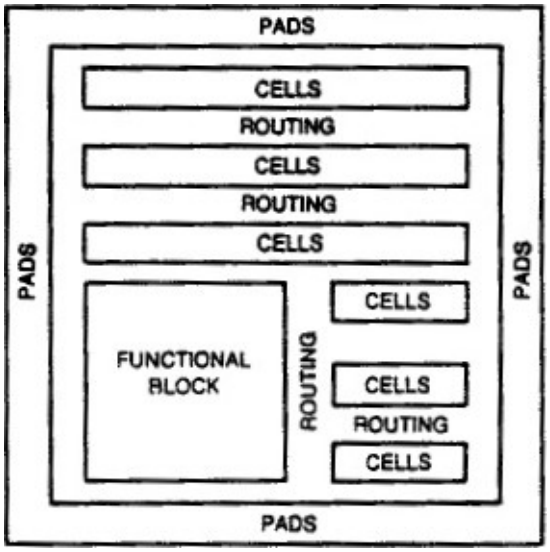


Figure 10: Typical standard cell floor plan

Standard cell chips use a structured floorplan:

- Cells are placed in **rows** separated by **routing channels**.
- Rows are arranged in **columns**.
- Larger functional blocks (e.g., RAMs) are positioned for optimal interconnect with surrounding logic.

- **MSI cells** often have a fixed height and variable width to accommodate logic complexity, as illustrated in figure below:

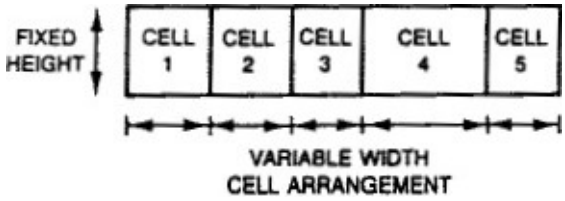


Figure 11: Standard MSI cells format

2.5.6 Hierarchical Design Approach

Modern standard cell systems support hierarchical design:

- Primitive cells are grouped into functional blocks.
- Functional blocks are further grouped to define the complete chip.
- Enables better design reuse, scalability, and modularity.

2.5.7 Skeleton Cell Layout

A skeleton view of a standard cell is shown in figure below:

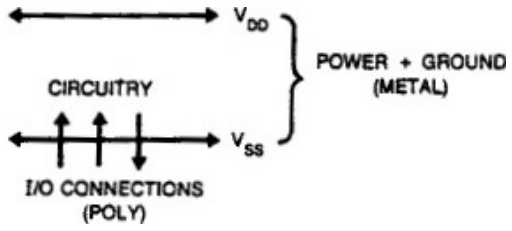


Figure 12: Standard cell skeleton

- **Power rails** typically run horizontally using the first metal layer.
- **I/O connections** are routed vertically using polysilicon or second metal layer.
- I/O access styles can vary depending on the cell library and are illustrated in figure below:

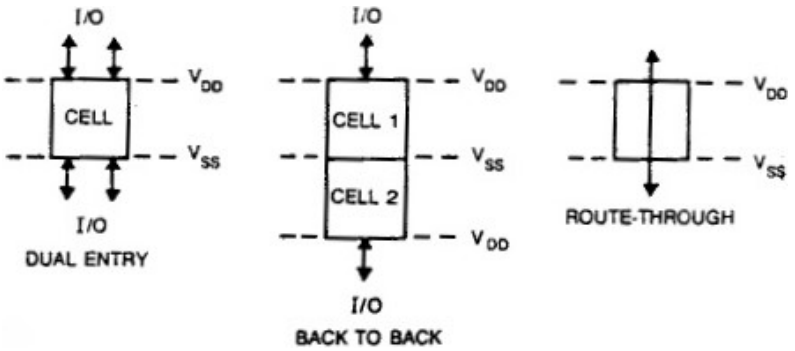


Figure 13: I/O access styles

2.5.8 Advantages of Standard Cell Design

- High performance due to optimized layout and routing.
- Reuse of verified, modular cell libraries.
- Scalability to both small and large VLSI systems.

2.6 Symbolic Layout Methods

In earlier sections, we discussed several techniques to **reduce the complexity** of IC (Integrated Circuit) design tasks.
These techniques included:

- **Hierarchy**
- **Regularity**
- **Modularity**
- **Locality**

Building upon these ideas, there is now a growing interest in **symbolic layout methods**. These approaches aim to **simplify the lower-level details** of IC design by:

- **Hiding process design rules**
- **Capturing structural and physical domain information** in a **looser, more abstract format**

Essentially, **symbolic layout** creates a representation similar to an **"assembly language"** in software, offering:

- **Simplified low-level design** at the bottom of the design hierarchy
- **Ease of use for system designers**, without needing to handle the detailed process-specific layout rules

Key Points:

- Symbolic layout methods focus on **abstraction** to speed up and simplify IC layout tasks.
- They represent **structural and physical properties** symbolically rather than with strict geometrical precision.
- Designers work at a **higher level of abstraction**, improving productivity while **hiding complex fabrication constraints**.

3 Testing

3.1 Introduction

- In LSI (Large Scale Integration) and VLSI (Very Large Scale Integration) design, an important consideration is the **testing of circuits**.
- Testing must be incorporated **during** the design phase itself and should proceed **concurrently** with architectural development.
- It must not be delayed until fabricated parts are available.

3.1.1 Testing Combinational Circuits

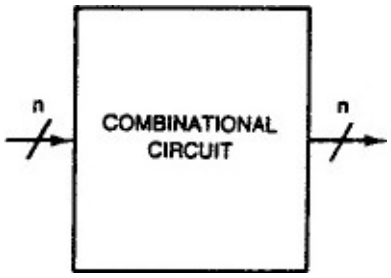


Figure 14: Combinational testing

Figure shows a **combinational circuit** with n inputs.

- To test this circuit exhaustively, a sequence of 2^n input combinations, known as **test vectors**, must be applied.
- Each input combination must be observed to fully verify the behavior of the circuit.

3.1.2 Testing Sequential Circuits

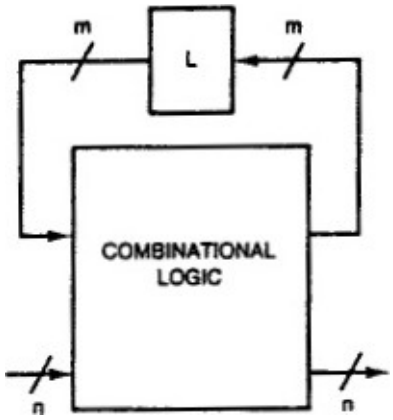


Figure 15: Sequential testing

By adding m storage latches to the combinational circuit, it is **converted into a sequential circuit**.

- In a sequential circuit, the output depends not only on the **current inputs** but also on the **previous state**.
- Consequently, an exhaustive test now requires 2^{n+m} input combinations.

3.1.3 Practical Illustration of Testing Complexity

Williams [1] highlights the impracticality of exhaustive testing through an example:

- Consider a network with:
  - $N = 25$ inputs
  - $M = 50$ storage elements
- The total number of required test patterns:

$$2^{(25+50)} = 2^{75} \approx 3.8 \times 10^{22}$$

- If test patterns are applied at a rate of **1 microsecond per pattern**, the total time needed would exceed **one billion years** (10^9 years).
- Exhaustive testing is **completely impractical** for complex LSI and VLSI circuits. Thus, **efficient testing strategies** must be adopted.

References

[1] Williams, "Testing techniques for LSI and VLSI circuits," 1983.

3.1.4 Important Areas of Testing

Three key areas are critical for effective testing:

1. **Test Generation:**
   - Concerned with creating a **minimal** number of tests required to verify the circuit's behavior.
   - Aims to achieve maximum coverage of internal nodes efficiently.
2. **Test Verification:**
   - Focuses on measuring the **effectiveness** of the generated tests.
   - Commonly evaluated through **fault simulation**, where faults are inserted intentionally to check if the test set detects them.
3. **Design for Test (DFT):**
   - Involves designing the circuit in such a way that testing becomes **simpler and more manageable**.
   - Proper DFT reduces the effort and complexity involved in test generation and test verification.

3.1.5 Sources of Test Inputs

Test inputs can originate from two major sources:

1. **Designer-Supplied Tests:**
   - Test vectors provided by the designer to verify the circuit's functionality.
   - Example: A variety of programs that run on a microprocessor (if the microprocessor is the device under test).
2. **Manufacturer-Supplied Tests:**
   - Applied by the manufacturer before shipping the product.
   - These tests aim to verify a high percentage of **internal good nodes**, ensuring product reliability.

3.2 Fault Models

- In digital circuit testing, fault models are used to represent physical failures in a simplified logical form.
- These models help in generating test vectors and understanding the effectiveness of tests.

3.2.1 Stuck-At Fault Model

- One of the most widely used fault models is the **Stuck-At** fault model.
- In this model, a gate input or output is assumed to be stuck at logic '0' (**S-A-0**) or stuck at logic '1' (**S-A-1**).
- When an input sequence is applied to a circuit, the **fault coverage** is defined as the percentage of S-A-0 or S-A-1 faults that can be detected by that sequence.

Limitations of Stuck-At Fault Models

- Not all physical faults can be captured using S-A models.
- Many faults arise from **short circuits** or **open circuits**, which alter the circuit behavior in ways not modeled by S-A-0/1.
- Figure drawn below illustrates this with examples:
  - Short **S1** behaves like a S-A-0 fault at input A.
  - Short **S2** changes the actual logic function of the gate.
- Hence, it is necessary to model faults at the **transistor level**, where the complete structure is visible.
- For example, in a CMOS NAND gate, the intermediate node in the series n-transistor path is hidden at the schematic level.

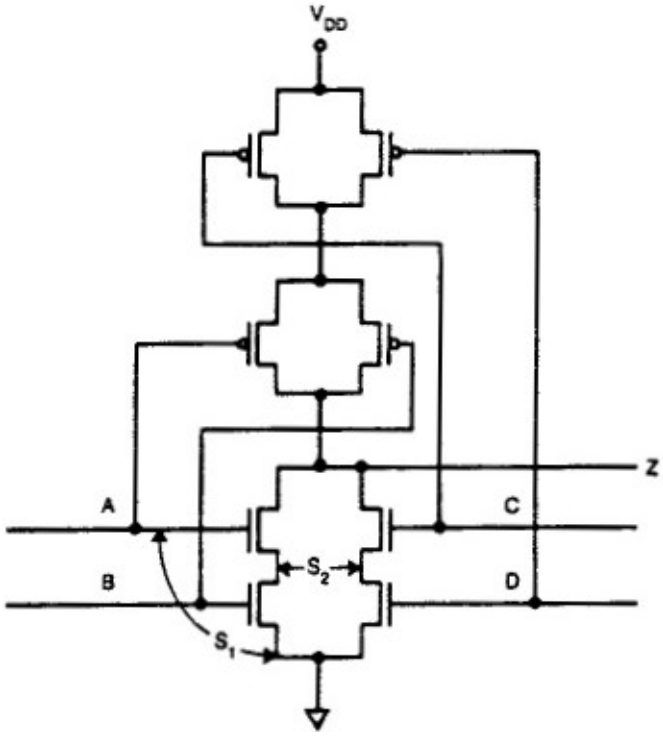


Figure 16: Faults in CMOS

3.2.2 Switch-Level Faults in CMOS

- Test generation should consider faults at the **switch level**, including possible shorts and opens.
- Although switch-level modeling is more accurate, most test systems still use Boolean logic models.

3.2.3 Sequential Behavior from Faults in CMOS

- A unique issue in CMOS is that certain faults may **convert a combinational circuit into a sequential circuit**.
- For example, a 2-input NOR gate with a stuck-open transistor may retain part of its previous state.

  If the n-transistor connected to A is stuck open:

  $$F = \overline{A + B} + A \cdot \overline{B} \cdot F_n$$

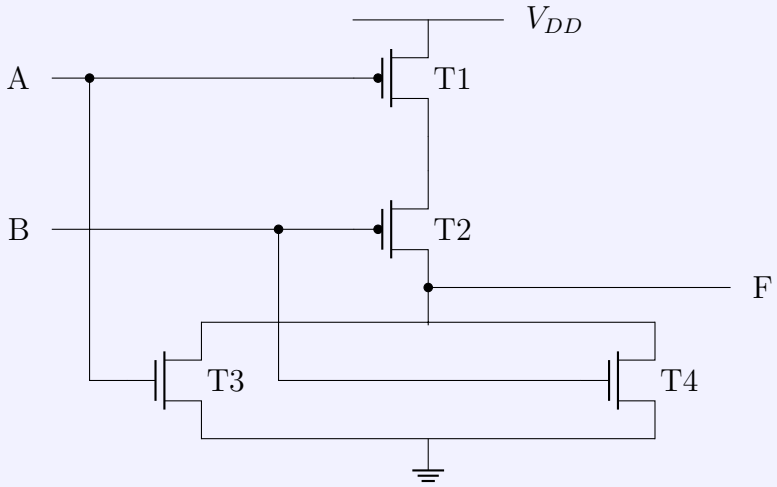
  If the n-transistor connected to B is stuck open:

  $$F = \overline{A + B} + \overline{A} \cdot B \cdot F_n$$

- Here, F_n is the **previous state** of the output.
- If a p-transistor is missing, the output node may remain arbitrarily charged until discharged, causing unreliable behavior.

Sequential Behavior from Faults in CMOS

- In CMOS circuits, certain faults can cause a purely combinational logic gate to exhibit sequential behavior.
- This is particularly problematic because such behavior is unintended and can lead to unpredictable results.
- Consider the example of a 2-input CMOS NOR gate, where a transistor is stuck-open (e.g., due to a missing gate, source, or drain connection).
- The output node may retain its previous logic level under some input conditions because no path exists to discharge or charge it properly.
- Let us examine such a scenario with the circuit below.



- Now, suppose the nMOS transistor controlled by input A is stuck open.
- When $A = 1$ and $B = 0$, the path for discharging the output node is broken.
- If the output was previously low, it remains low; otherwise, it could stay high due to the absence of a pull-down path — hence, the output F depends on the previous state F_n.
- The logic becomes:

$$F = \overline{A + B} + A \cdot \overline{B} \cdot F_n$$

  This is not a NOR gate anymore — it's a sequential element, as it depends on the prior state.

3.2.4 CMOS Fault Modeling for Test Generation

- Due to such behavior, researchers have proposed **new fault models and test generation techniques**.
- One such method is the use of the **D-algorithm** for logic-level test generation.

Logic Model Representation

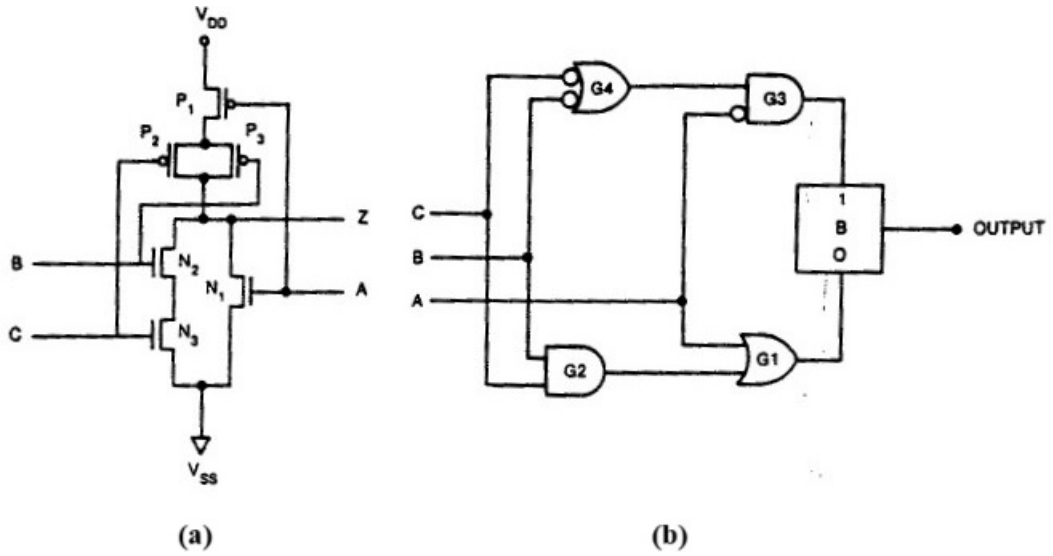


Figure 17: Logic fault model for CMOS

- **Figure (a)** shows a CMOS gate at the circuit level.
- **Figure (b)** shows the corresponding logic model, composed of:
  - A **p-tree** and **n-tree** representing the pull-up and pull-down networks.
  - A **B-block**, which captures output behavior as per input states.

Deriving Logic from CMOS Transistor Circuit

CMOS Transistor Logic (Fig. a) consists of:

- **PMOS Network:** Pulls Z high when A = 0 and (B = 0 or C = 0); **Logic:** $\overline{A} \cdot (\overline{B} + \overline{C})$
- **NMOS Network:** Pulls Z low when A = 1 or (B = 1 and C = 1); **Logic:** $A + (B \cdot C)$
- **output Z** $= \overline{A + (B \cdot C)}$

Gate-Level Equivalent (Fig. b) expresses the same logic:
For output low:

- G2: $B \cdot C$
- G1: $A + B \cdot C$

For output high:

- G4: $\overline{B} + \overline{C}$
- G3: $\overline{A} \cdot \overline{B} + \overline{C}$

Thus, Fig. (b) represents the Boolean logic implemented by the CMOS circuit in Fig. (a).

- The table below shows the characteristics of B-block:

Table 2: B-block output behavior

S_1	S_0	Output Y
1	1	X (unknown or biased)
1	0	1
0	1	0
0	1	0
0	0	m (memory state)

- The **m-state** indicates memory behavior (retains previous output).
- The **X-state** is unpredictable and may lean towards 0 or 1 depending on logic and fabrication.

Understanding B-block

What is the B-block?

- The B-block is part of a logic-level fault modeling abstraction for CMOS gates.
- It is used in test generation, especially for methods like the D-algorithm, to simulate the behavior of faulty CMOS logic gates without analyzing every transistor individually.
- It originates from a logical model of a CMOS gate proposed for structured fault simulation. In this abstraction:
  - The p-tree (pull-up PMOS network) and the n-tree (pull-down NMOS network) are treated as separate logic blocks.
  - The B-block is then used to model the output behavior based on the signals from both the p-tree and n-tree.
- This abstraction simplifies fault analysis by converting switch-level faults (shorts, opens, stuck-at faults at transistors) into logic-level equivalents.

How does the B-block work?

- The B-block uses two control signals:
  - S_1: the status of the p-tree (pull-up)
  - S_0: the status of the n-tree (pull-down)
- The output Y is then determined by these, as shown in the truth table:

S_1 (p-tree)	S_0 (n-tree)	Y (Output)
1	1	X (Conflict)
1	0	1 (Driven High)
0	1	0 (Driven Low)
0	1	0 (Driven Low)
0	0	m (Memory/Previous State)

Why is $S_1 = 0, S_0 = 1$ Listed Twice?
While both rows yield the same output ($Y = 0$), their **origins may differ** in physical fault scenarios:

- One case may result from a normally functioning n-tree pulling the output low, with the p-tree inactive.
- The second instance might represent a *fault condition*, such as a stuck-open p-transistor that fails to assert high when needed.

This apparent duplication improves:

- Fault tracing back to transistor-level causes.
- Diagnostic precision in automated test pattern generation.
- Mapping of transistor-level behavior to logic-level abstractions.

Thus, the repetition is intentional and ensures completeness in the logic model used for test generation and fault simulation.

Mapping Circuit Faults to Logic Faults

- Table below shows how faults in the CMOS circuit map to logical fault models.

Table 3: Equivalent Faults Mapping

MOS Circuit Fault	Equivalent Logic Circuit Fault
A S-A-0/1	A S-A-0/1
B S-A-0/1	B S-A-0/1
C S-A-0/1	C S-A-0/1
Z S-A-0/1	Z S-A-0/1
P1 short	G3, A S-A-0
P1 open	G3, A S-A-1
N1 short	G1, A S-A-1
N2 open	G1, A S-A-0

3.3 Design for Testability

- Testing is an essential step in ensuring that fabricated digital circuits function correctly.
- Designing for testability enhances fault detection and diagnosis, making circuits easier to test and maintain.

3.3.1 Key Concepts

Two important concepts in testability are:

- **Controllability**: The ability to *set and reset every internal node* in the circuit to a known logic value via the inputs.
- **Observability**: The ability to *observe the state of any internal node*, either directly at the outputs or indirectly through other observable nodes.

Given a circuit structure, tools such as **SCOAP (Sandia Controllability/Observability Analysis Program)** are available to calculate the controllability and observability metrics of internal nodes.

3.3.2 Approaches to Design for Testability

There are three main approaches to designing circuits that are testable:

1. **Ad Hoc Testing**
   - Involves manual techniques or guidelines.
   - Examples: avoiding asynchronous logic, simplifying state machines, and inserting test points manually.
   - Advantage: Quick to apply.
   - Limitation: Not scalable or systematic for complex designs.
2. **Structured Design for Testability**
   - Uses well-defined methods and design styles to simplify testing.
   - Common techniques include:
     - **Scan Design**: Flip-flops are connected into a scan chain to facilitate controllability and observability.
     - **Level-Sensitive Scan Design (LSSD)**.
   - Enables automated test pattern generation and improved fault coverage.
3. **Self-Test and Built-In Testing**
   - Also known as **Built-In Self-Test (BIST)**.
   - Circuit includes extra hardware to test itself without full reliance on external test equipment.
   - Components include:
     - **Test Pattern Generator** (e.g., Linear Feedback Shift Register — LFSR)
     - **Output Response Analyzer** (e.g., Signature Analyzer)
   - Allows efficient testing in the field.

3.4 Ad Hoc Testing

- Ad hoc testing refers to a collection of informal and practical techniques used to make circuits easier to test, especially by reducing the complexity of test generation.
- These methods aim to prevent the **combinational explosion** that can occur when testing large digital systems.

Key Techniques

- **Partitioning Large Sequential Circuits**:
  - Large circuits, such as long counters, are split into smaller blocks.
  - Each block can then be tested using fewer test vectors.
  - Example: A 16-bit counter might be partitioned into four 4-bit counters.
- **Adding Test Points**:
  - Insert test points to improve *controllability* and *observability*.
  - Helps expose internal faults by allowing access to hidden nodes.
- **Using System Bus for Testing**:
  - In bus-oriented systems, the bus can be reused for transferring test patterns and capturing test responses.
  - Efficient for testing multiple modules connected to the same data bus.
- **Testing Bit-Sliced Systems**:
  - Bit-sliced systems use modular logic slices (e.g., 4-bit ALUs chained to build larger words).
  - These can be tested using methods derived from testing *Iterative Logic Arrays (ILAs)*.

3.4.1 Iterative Logic Arrays (ILAs)

ILAs are regular, modular structures composed of identical logic cells connected in a fixed pattern. They are commonly used in arithmetic circuits and bit-sliced architectures.

- **C-testable ILA**:
  - Can be tested with a *constant number of input patterns*, regardless of array size.
  - Reduces test vector count significantly for large arrays.
- **I-testable ILA**:
  - Ensures that the *response of each cell is identical*.
  - Enables testing using **equality circuits** that compare cell outputs.
- **CI-testable ILA**:
  - Combines both C-testability and I-testability.
  - Highly testable and efficient in terms of test generation.

Example: Testable ILA Cell

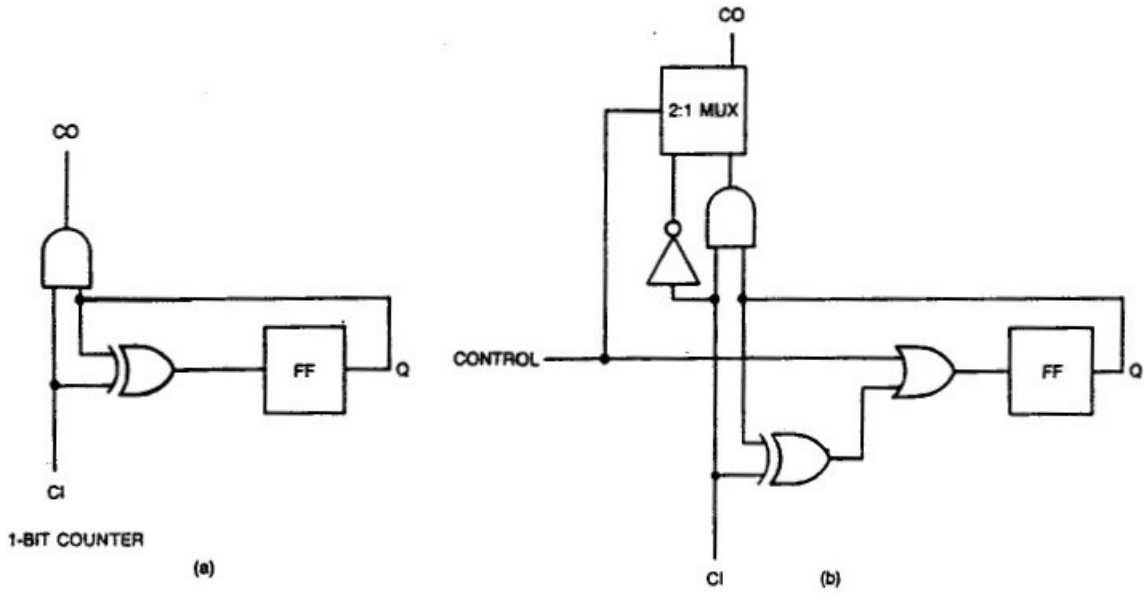


Figure 18: Bit-slice testing

- An example of a modified ILA for I-testability involves a cascaded 1-bit counter cell, to which two extra gates are added.
- These gates enable control and observation of the cell's operation during testing.
- Such modifications allow the entire ILA to be tested using minimal test patterns and simple comparison logic.

3.5 Structured Design for Testability

Structured design for testability involves systematic methods that ensure a circuit is easily testable, based on the fundamental principles of **controllability** and **observability**.

Level-Sensitive Scan Design (LSSD):

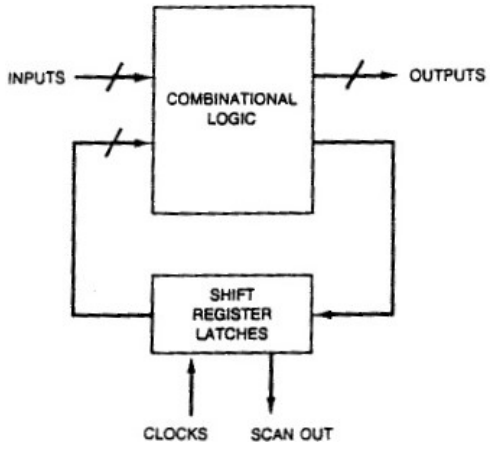


Figure 19: LSSD testing

- One of the most widely used structured approaches is **Level-Sensitive Scan Design (LSSD)**, originally developed by IBM.
- This method utilizes specially designed latches known as **Shift Register Latches (SRLs)**.
- In **normal mode**, SRLs function as standard storage elements within the sequential circuit.
- In **test mode**, all SRLs are connected in series to form a scan chain, allowing data to be shifted through the latches.

Testing Procedure Using LSSD:

1. Shift in a known pattern to initialize all SRLs (achieving controllability).
2. Allow the circuit to operate for one clock cycle, propagating values through the combinational logic.
3. Capture the output into the SRLs.
4. Shift out the captured results for analysis (achieving observability).

Advantages:

- Simplifies test generation, since the combinational portion can be tested separately using Automatic Test Pattern Generation (ATPG) tools.
- Improves fault coverage in large sequential designs.

Drawbacks:

- Increased circuit complexity due to additional logic in the latches.
- Additional I/O pins may be required for scan control and data.
- Physical placement challenges, as latches across the chip must be chained into a scan path.

Static Latch Implementations:
Various versions of static latches used in LSSD are shown in figures (a) to (d) below. These are based on the basic **static-D latch**.

- In Figure (a), input `D` is the normal data input, and input `I` comes from the preceding latch in the scan chain.
- A 2-input multiplexer selects between normal data and test input.
- In Figure (b), the same function is implemented with reduced speed but simpler design.
- Figures (c) and (d) show optimized layouts, increasing transistor count from 10 to 14.

Trade-off: Designers must weigh the increased area, power, and possible speed penalties against the benefits of enhanced testability.

Structured design for testability trades off complexity and area for efficient test access and fault detection.

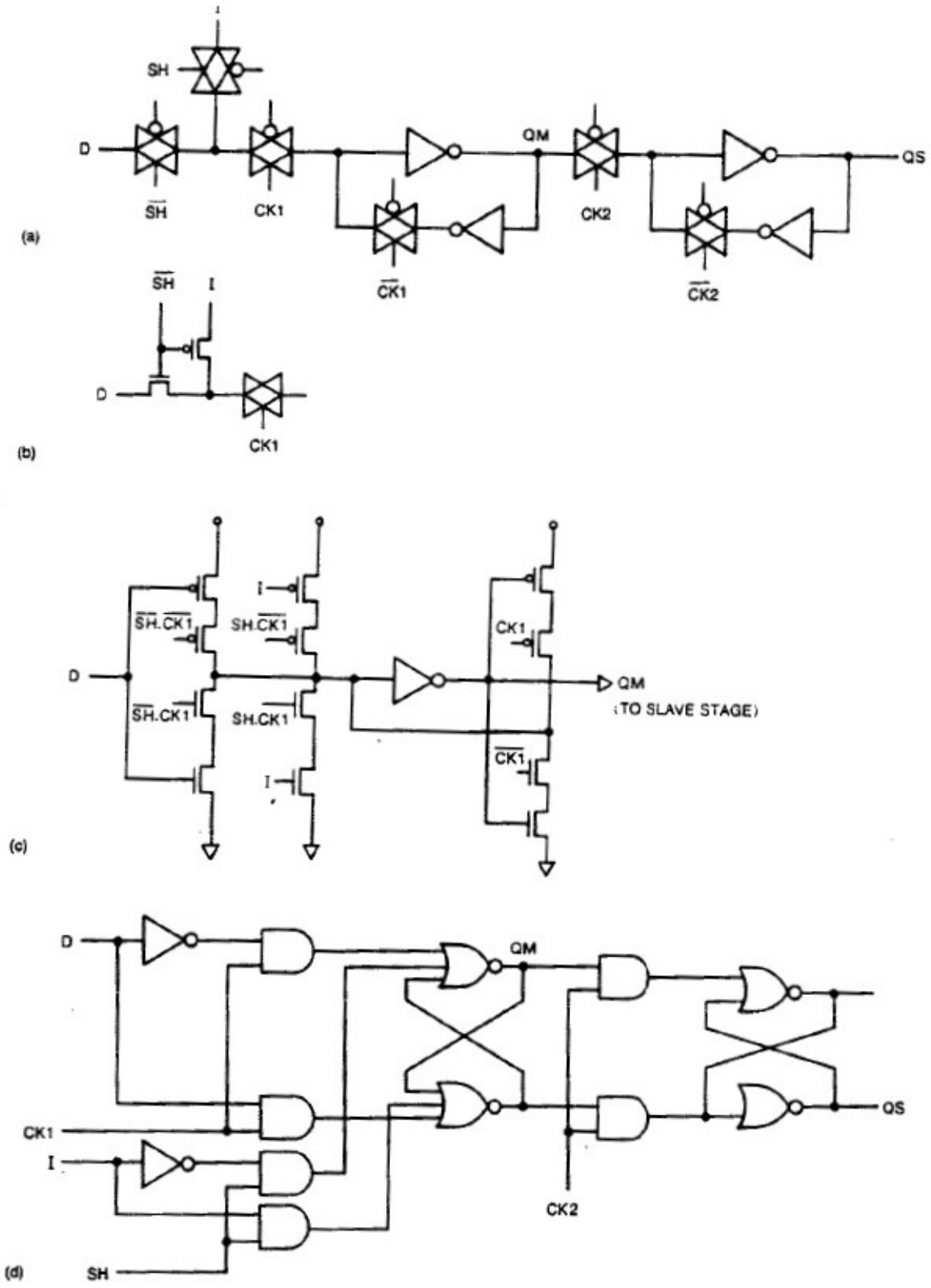


Figure 20: LSSD implementations

3.6 Self-Test and Built-In Test

Built-in test (BIT) techniques are used to enhance testability by embedding test generation and response evaluation circuitry within the chip itself. This allows for at-speed testing and reduces dependence on external Automatic Test Equipment (ATE).

The following are some key types of Built-In Test techniques used in VLSI systems:

1. Signature Analysis and Cyclic Redundancy Checking (CRC)

One common method of implementing BIT is through **signature analysis** or **cyclic redundancy checking**, which uses a **Linear Feedback Shift Register (LFSR)**, as shown in figure below:

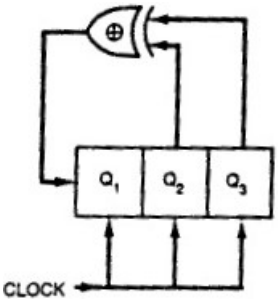


Figure 21: A linear feedback shift register

- The LFSR is initialized, and the resulting register value becomes a function of:
  - the number of latch inputs,
  - their values,
  - and the transformation polynomial of the analyzer.
- A correct circuit produces a known signature. A faulty circuit will produce a different signature.

2. BILBO: Built-In Logic Block Observation

The BILBO structure combines signature analysis with LSSD to form a compact and flexible test element. It is shown in below.

- A 3-bit register with associated logic can operate in multiple modes depending on control inputs C_0 and C_1:

Table 4: BILBO modes

Mode	Control	Function
A	$C_0 = 1$, $C_1 = 1$	Parallel register
B	$C_0 = 0$, $C_1 = 0$	Scan register
C	$C_0 = 1$, $C_1 = 0$	Signature analyzer or PRSG
Reset	$C_0 = 0$, $C_1 = 1$	Register reset

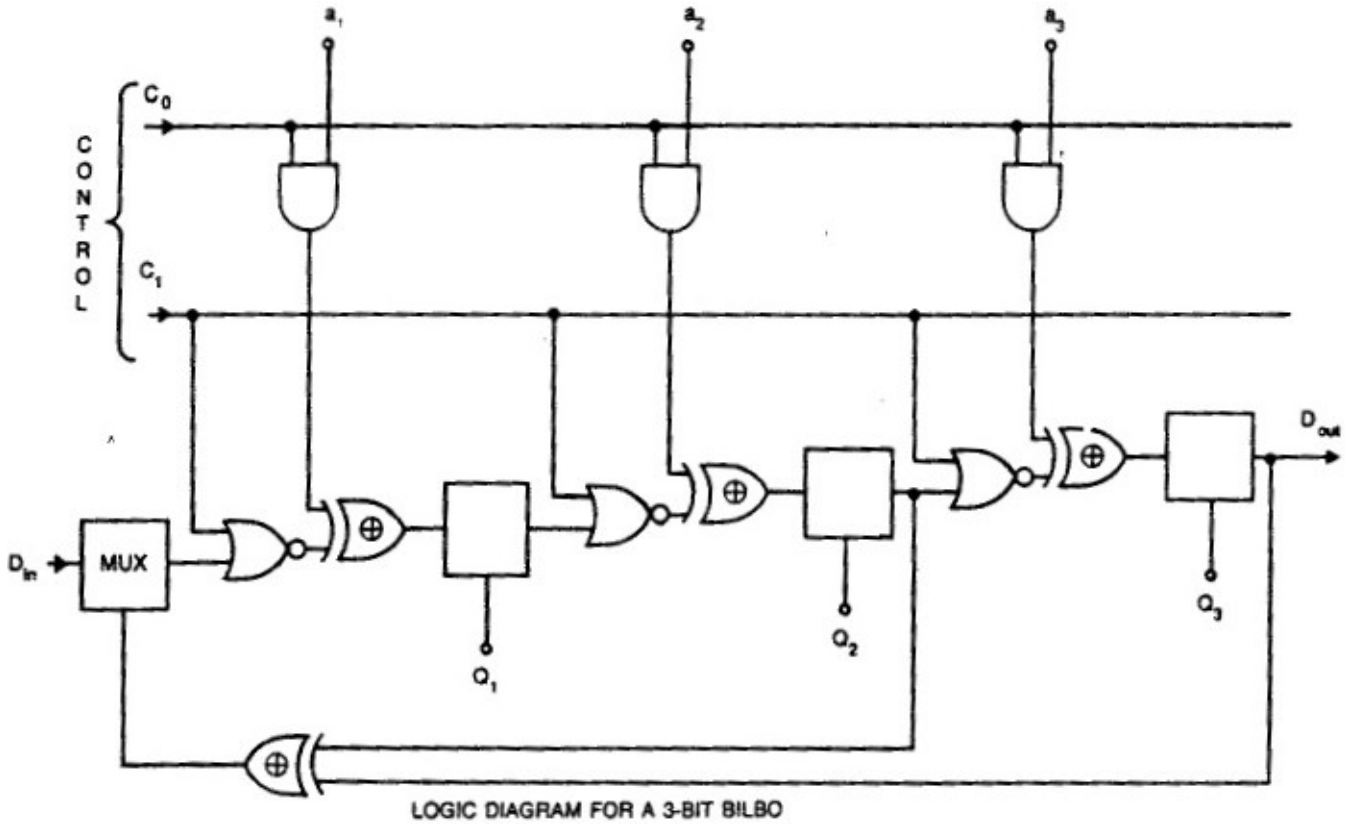


Figure 22: BILBO circuitry

- A complete BIT structure using BILBO is illustrated in figure below:

Figure 23: BILBO usage

3. Design for Autonomous Test

This technique partitions the circuit into small modules using multiplexers and tests them exhaustively without requiring fault models or test generation algorithms.

- Figure (a) shows a circuit with inserted multiplexers.
- Figure (b) presents the circuit in normal operation.
- Figure (c) shows the circuit configured to test module A.

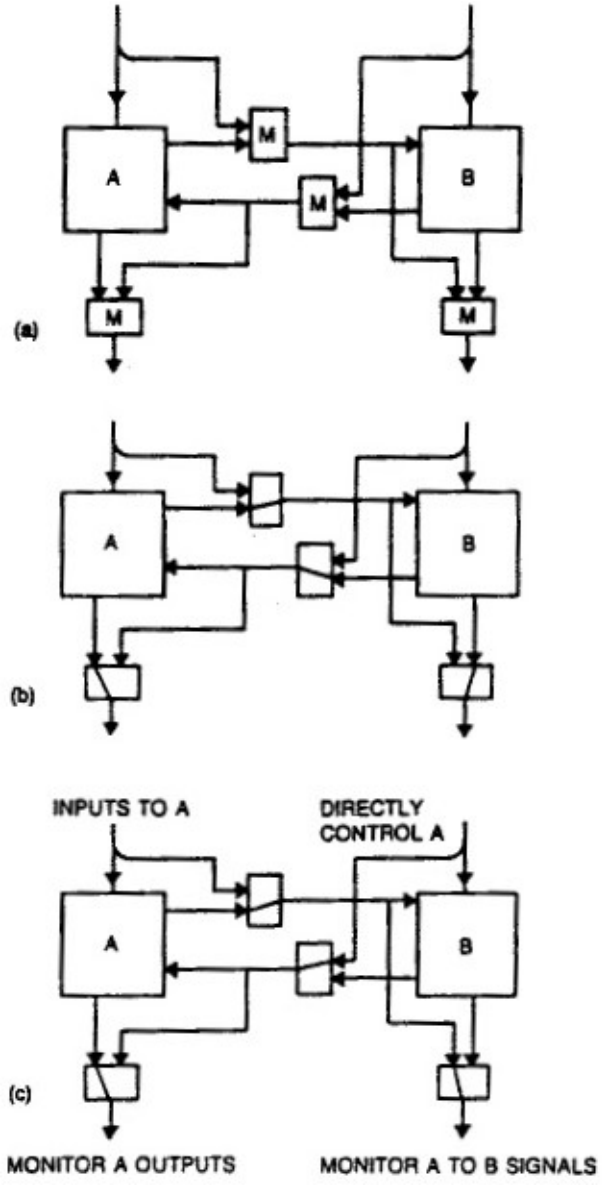


Figure 24: Multiplexer segmenting for autonomous test

- Additional test hardware includes:
  - Pattern generator (LFSR),
  - Signature analyzer,
  - Test control logic.

Limitation: This method is ineffective for stuck-open faults.

4. Stuck-Open Fault Detection

To handle stuck-open faults, the circuit shown in figure below introduces a **Charge/Discharge (C/D)** unit at the gate output.

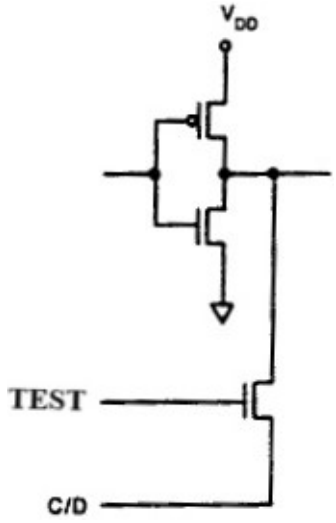


Figure 25: C/D circuitry

- The test sequence involves:
  - Asserting `TEST`,
  - Strobing the `C/D` control line high and then low.
- Correct outputs respond by toggling, while stuck-open lines remain charged or discharged.

5. Syndrome Testing

Syndrome testing is an exhaustive method where all possible input patterns are applied and the number of logical 1s in the output is counted.

- This count (the "syndrome") is compared with that of a fault-free circuit.
- Required components:
  - Pattern generator,
  - Output counter,
  - Comparison circuit.

6. Other Techniques

Additional BIT approaches include:

- **Double or Triple Modular Redundancy** with majority voting logic.
- Use of on-chip **ROM-based stimulus generators**.

Self-test and BIST techniques reduce dependency on external testers and improve test coverage in complex VLSI systems.

3.7 Layout for Improved Testability

Key Concept:
Physical layout influences the testability of VLSI circuits. Proper layout techniques can reduce the likelihood of faults such as shorts and opens.

Observations by Galiay (GaCV80):

- Study based on fabricated **nMOS circuits**.
- Most common failures:
  - Shorts in the metal layer
  - Opens in the metal layer
  - Shorts in the diffusion layer

Implications:

- Specific layout rules can improve testability.
- These rules are likely **technology-dependent**.
- A different set of guidelines may be needed for **CMOS** technologies.

Research Direction:
Further research is needed to explore layout-level strategies for improved testability, especially for newer technologies like CMOS.

3.8 Summary — Testing

Main Idea:
Design decisions should consider **testability** from the outset, as testing becomes harder and more costly if added later.

Design Techniques:

- Incorporate test structures early in the design cycle.
- Be aware of trade-offs: testability may affect:
  - Chip area
  - Speed/performance

Example:

- A **data-path approach** might be easier to test than a **random logic** approach, with only minor area differences.

Conclusion:

- Embed testability early in the design process.
- Choose architectures that balance functionality and ease of testing.