Electronic Design Automation

Zoran Stamenković

stamenkovic@ihp-microelectronics.com

International Organization for Migration (IOM)
Organisation internationale pour les migrations (OIM)
Organización Internacional para las Migraciones (OIM)

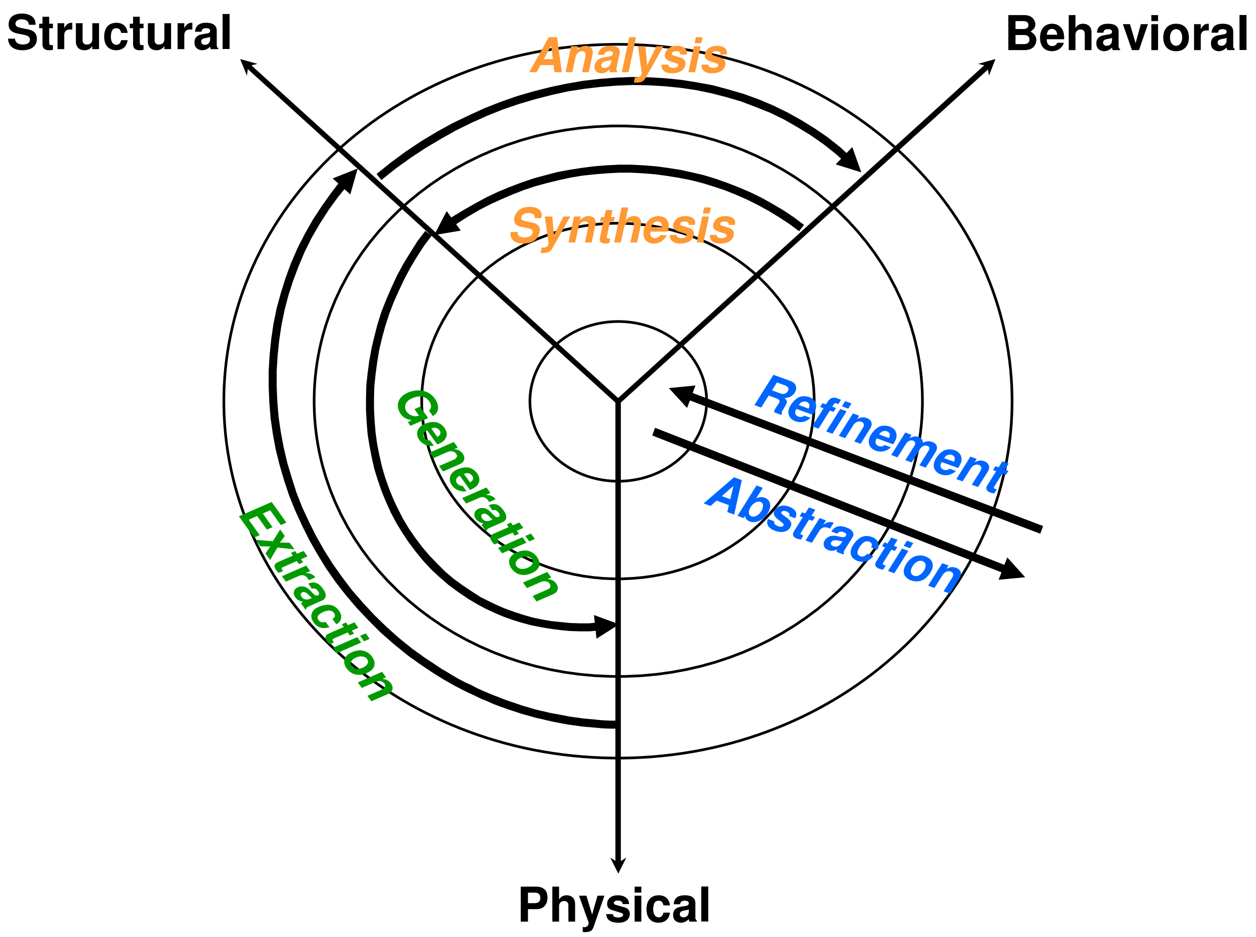
Topics

- Modeling and HDL
- Simulation and Verification
- Logic Synthesis
- Design for Testability
- Layout Generation
- Design for Manufacturability

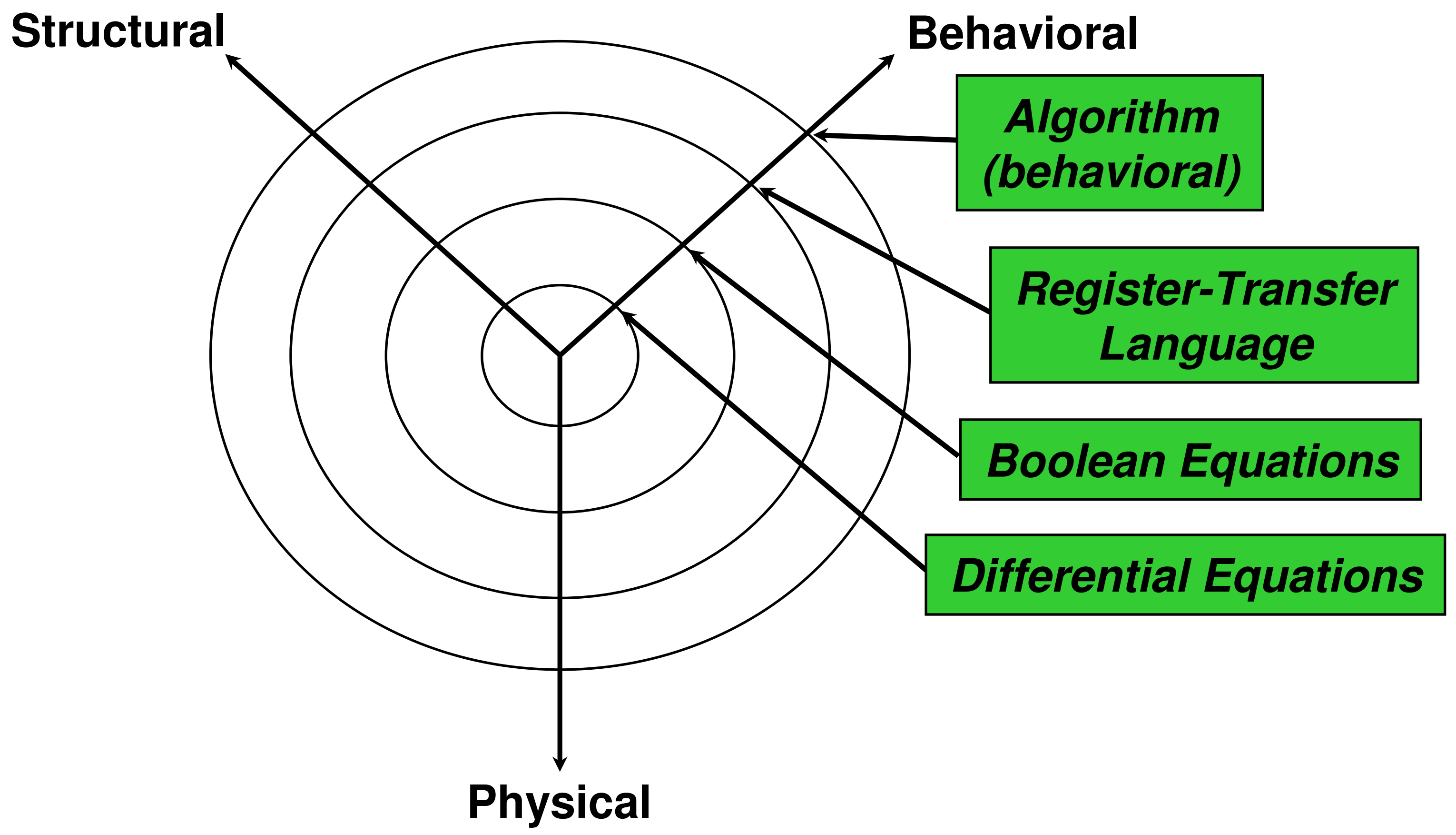
Modeling and HDL

- Domains and Levels
- Basics of HDL

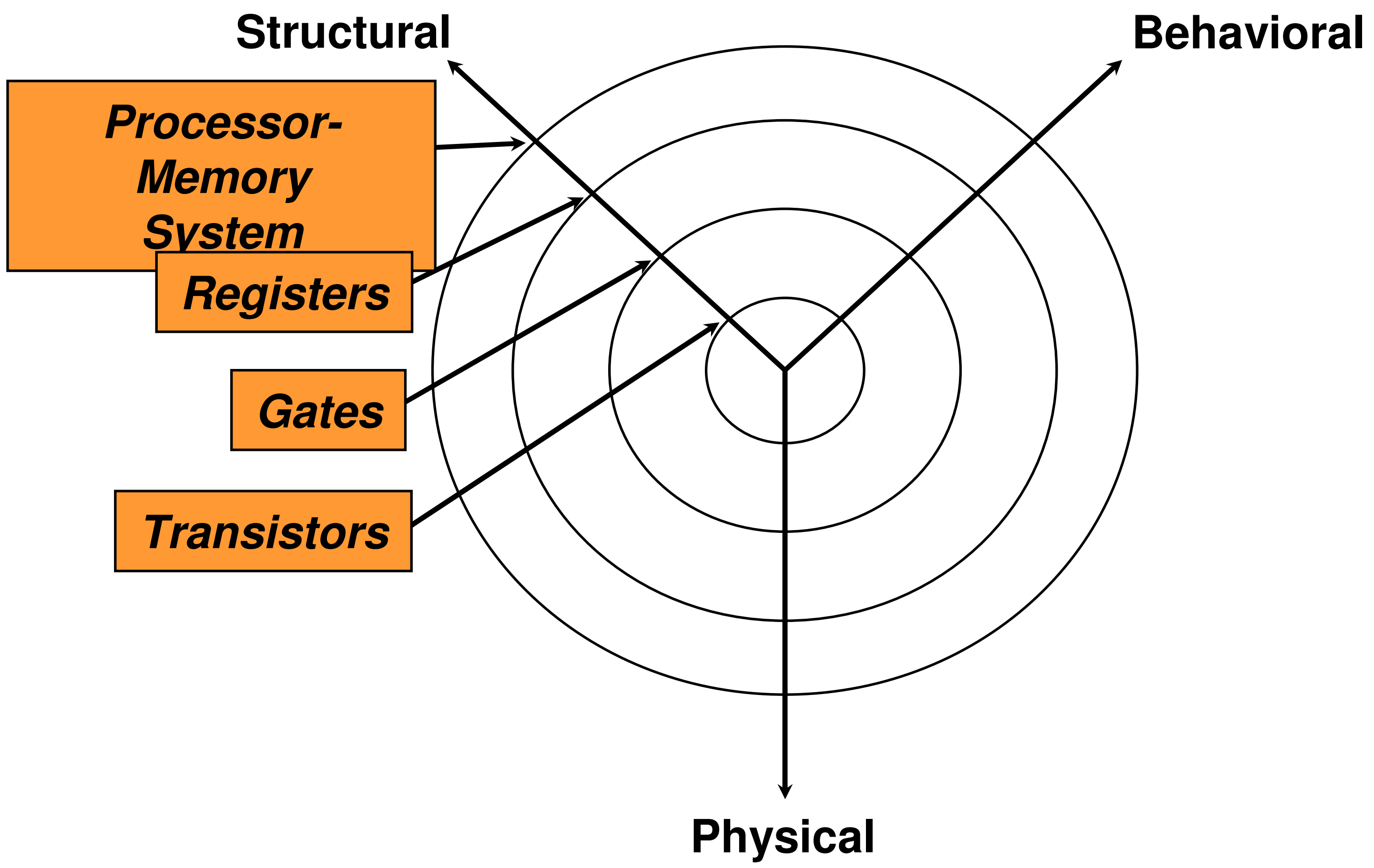
Domains and Levels



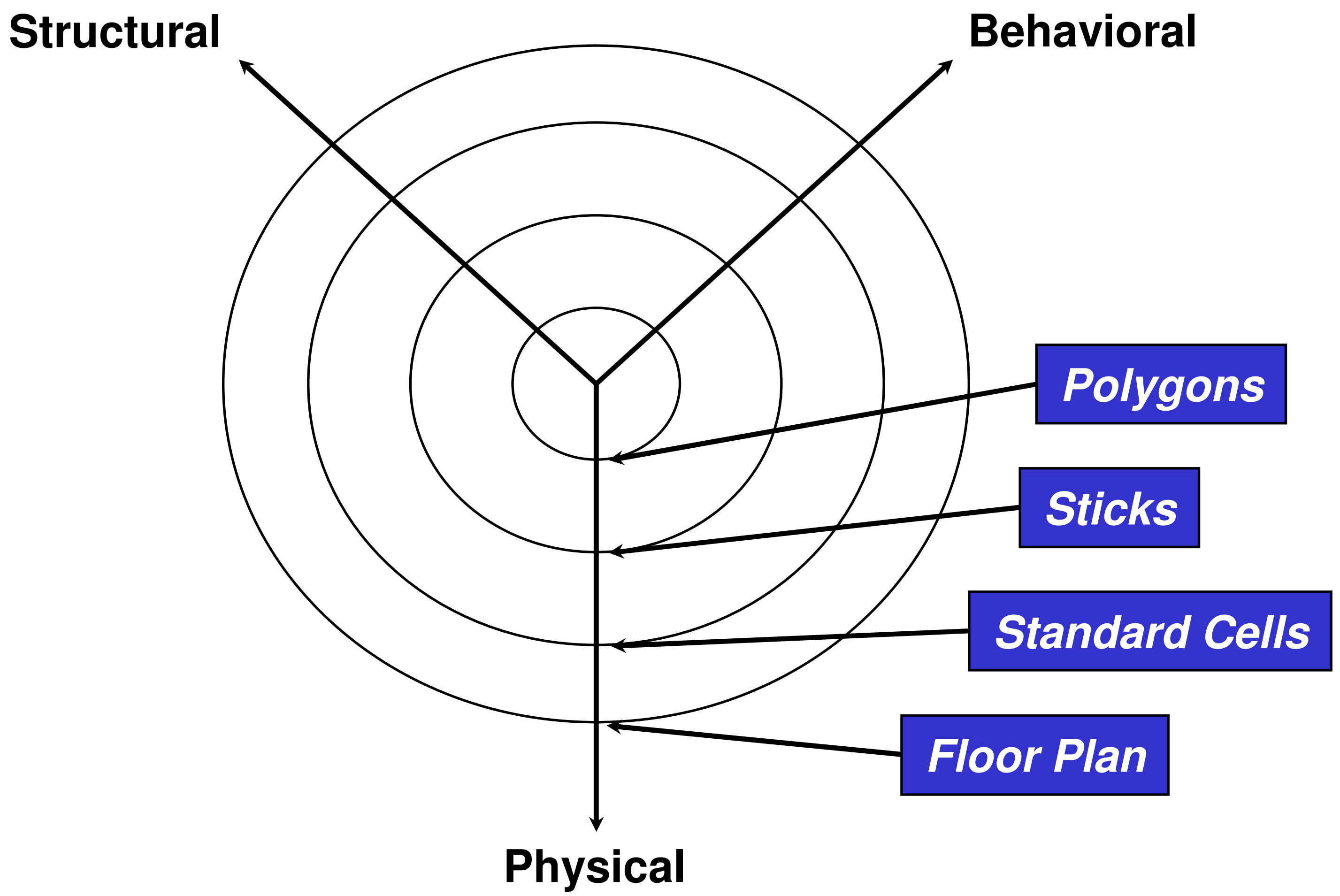
Behavioral Domain



Structural Domain



Physical Domain

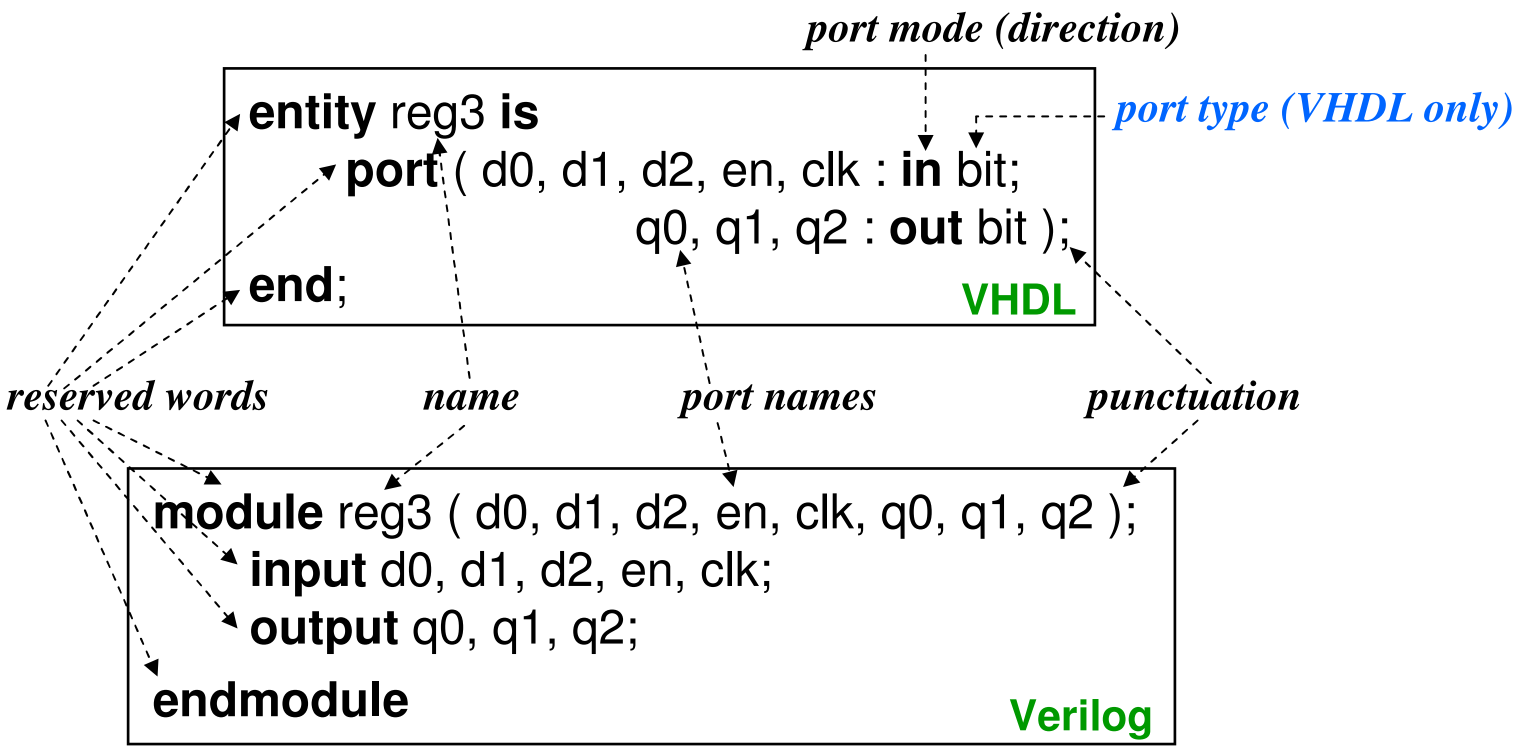


Hardware Description Languages

- Motivation for HDL
  - Increased hardware complexity
  - Design space exploration
  - Inexpensive alternative to prototyping
- General features
  - Support for describing circuit connectivity
  - High-level programming language support for describing behavior
  - Support for timing information (constraints, etc.)
  - Support for concurrency
- VHDL
  - IEEE Standard 1076-1987
  - IEEE Standard 1076-1993
  - Extension VHDL-AMS-1999
- Verilog
  - IEEE Standard 1364-1995
  - IEEE Standard 1364-2000

Modeling Interfaces

- ***Entity (VHDL)*** **or** ***Module (Verilog)*** **declaration**
  Describes the input/output *ports* of a module



```
entity reg3 is
    port ( d0, d1, d2, en, clk : in bit;
                       q0, q1, q2 : out bit );
end;
```

VHDL

```
module reg3 ( d0, d1, d2, en, clk, q0, q1, q2 );
    input d0, d1, d2, en, clk;
    output q0, q1, q2;
endmodule
```

Verilog

Modeling Behavior

- *Architecture* Body (VHDL)
  - Describes an implementation of an entity
  - May be several per entity
- *Module* (Verilog)
  - Is unique
- *Behavioral* Architecture
  - Describes the algorithm performed by the module
  - Contains
    - *Procedural Statements*, each containing
    - *Sequential Statements*, including
    - *Assignment Statements* and
    - *Wait Statements*

Behavior Example

```vhdl
entity reg3 is
        port ( d0, d1, d2, en, clk : in bit;
                   q0, q1, q2 : out bit );
end;
architecture behav of reg3 is
begin
        process ( d0, d1, d2, en, clk )
          begin
                if en = '1' and clk = '1' then
                  q0 <= d0 after 5 ns;
                  q1 <= d1 after 5 ns;
                  q2 <= d2 after 5 ns;
                end if;
        end process;
end;
```

VHDL

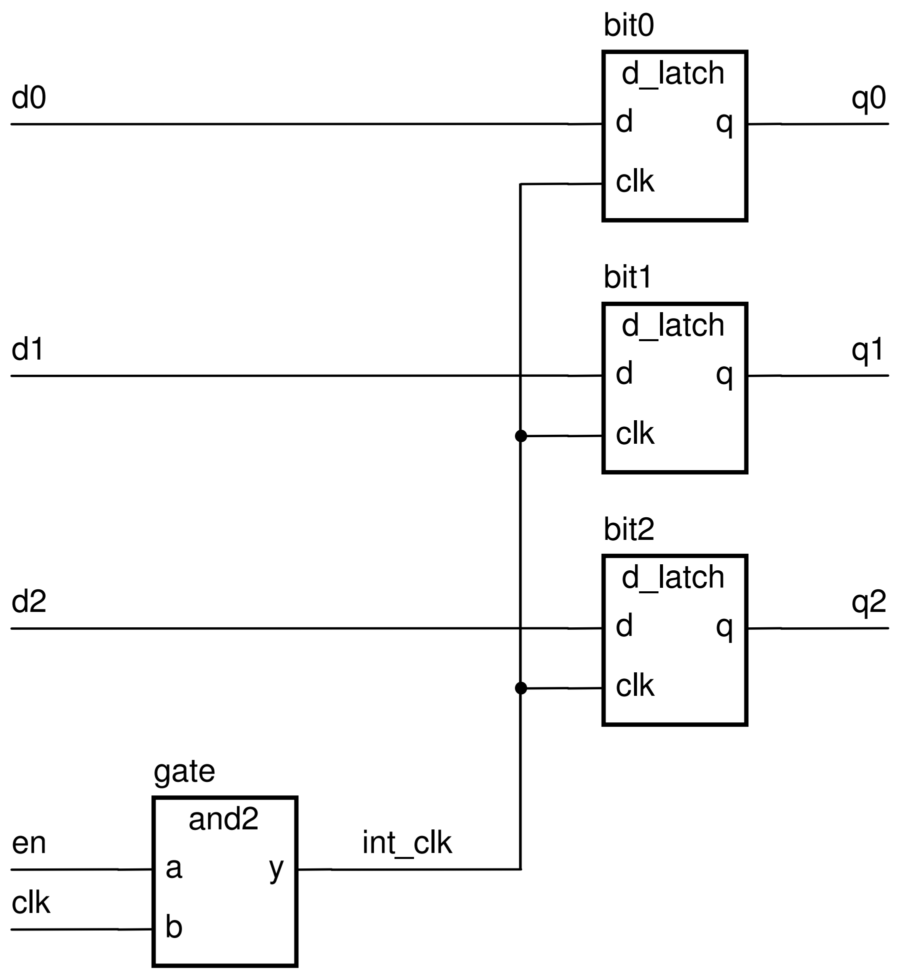
```verilog
`timescale 1ns/10ps
module reg3 ( d0, d1, d2, en, clk,
                           q0, q1, q2 );
        input d0, d1, d2, en, clk;
        output q0, q1, q2;
        reg q0, q1, q2;
always @ ( d0 or d1 or d2 or en or clk )
    if ( en & clk )
      begin
           q0 <= #5 d0;
           q1 <= #5 d1;
           q2 <= #5 d2;
      end
endmodule
```

Verilog

Modeling Structure

- *Structural* Architecture
  - Implements the module as a composition of components
  - Contains
    - *Signal Declarations* (entity ports are also signals)
      - Declare internal connections
    - *Component Instances*
      - Instantiate previously declared entity/architecture pairs
    - *Port Maps* in component instances
      - Connect signals to component ports
    - *Wait Statements*
      - Suspend a process or procedure

Structure Example



Structure Example

```
entity d_latch is
        port ( d, clk : in bit;  q : out bit );
end;
architecture basic of d_latch is
begin
        process ( d, clk )
        begin
                if clk = '1' then
                        q <= d after 5 ns;
                end if;
        end process;
end;
```

VHDL

```
`timescale 1ns/10ps
module d_latch ( d, clk, q );
        input d, clk;
        output q;
        reg q;
always @ ( d or clk )
   if ( clk )
     begin
       q <= #5 d;
     end
endmodule
```

Verilog

```
entity and2 is
        port ( a, b : in bit;  y : out bit );
end;
architecture basic of and2 is
begin
        process ( a, b )
        begin
                y <= a and b after 5 ns;
        end process;
end;
```

VHDL

```
`timescale 1ns/10ps
module and2 ( a, b, y );
        input a, b;
        output y;
        reg y;
always @ ( a or b )
   begin
       y <= #5 ( a & b );
   end
endmodule
```

Verilog

Structure Example

```vhdl
entity reg3 is
        port ( d0, d1, d2, en, clk : in bit;
                   q0, q1, q2 : out bit );
end;
architecture struct of reg3 is
        component d_latch
                port ( d, clk : in bit;  q : out bit );
        end component;
        component and2
                port ( a, b : in bit;  y : out bit );
        end component;
        signal int_clk : bit;
begin
        bit0 : d_latch port map ( d0, int_clk, q0 );
        bit1 : d_latch port map ( d1, int_clk, q1 );
        bit2 : d_latch port map ( d2, int_clk, q2 );
        gate : and2 port map ( en, clk, int_clk );
end;
```

VHDL

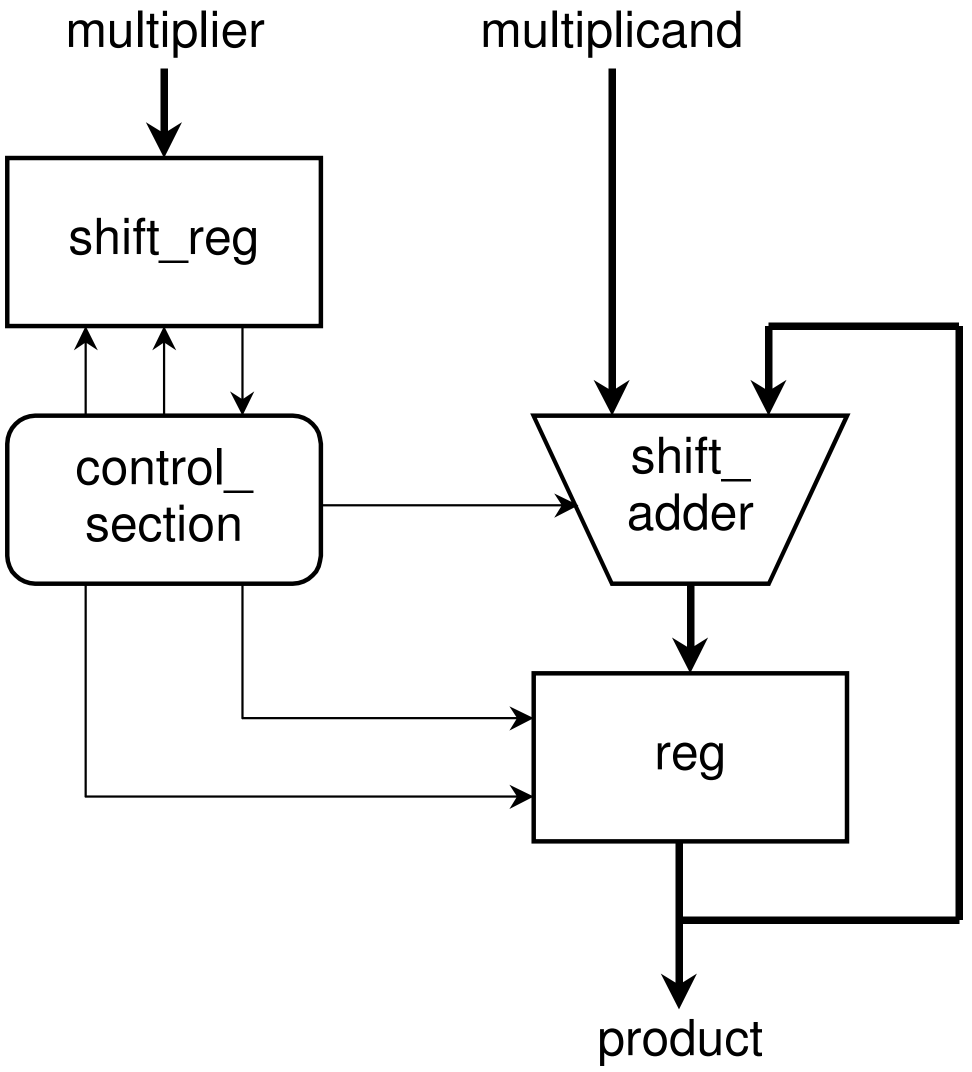
```verilog
module reg3 ( d0, d1, d2, en, clk,
                        q0, q1, q2 );
        input  d0, d1, d2, en, clk;
        output q0, q1, q2;
    wire int_clk;
    d_latch bit0 ( d0, int_clk, q0 );
    d_latch bit1 ( d1, int_clk, q1 );
    d_latch bit2 ( d2, int_clk, q2 );
    and2 gate ( en, clk, int_clk );
endmodule
```

Verilog

Mixing Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - Process Statements and Component Instances
    - Collectively called *Concurrent Statements*
  - Processes can read and assign to signals
- Example: Register-transfer-language model
  - Data-path described *structurally*
  - Control section described *behaviorally*

Mixed Example



Mixed Example

```
entity multiplier is
        port ( clk, reset : in bit;
                  multiplicand, multiplier : in integer;
                   product : out integer );
end;

architecture mixed of multiplier is
        signal partial_product, full_product : integer;
        signal arith_control, result_en, mult_bit, mult_load : bit;
begin
        arith_unit : entity work.shift_adder(behavior)
                port map ( addend => multiplicand,  augend => full_product,
                                        sum => partial_product,
                                        add_control => arith_control );
        result : entity work.reg(behavior)
                port map ( d => partial_product,  q => full_product,
                                        en => result_en,  reset => reset );
        ...
```

Mixed Example

```
        …
        multiplier_sr : entity work.shift_reg(behavior)
                port map ( d => multiplier,  q => mult_bit,
                                        load => mult_load,  clk => clk );
        product <= full_product;

        control_section : process is
                -- variable declarations for control_section
                -- …
        begin
                -- sequential statements to assign values to control signals
                -- …
                wait on clk, reset;
        end process control_section;
end;
```

Simulation and Verification

- Simulation
- Verification
- Annotation

Simulation

- Simulation
  - Tests the functionality of a design’s elaborated model
  - Needs a test bench and a simulation tool
  - Advances in discrete time steps
- Test Bench
  - Includes an instance of the design under test
  - Applies sequences of test values to inputs
  - Monitors signal values on outputs using simulator
- Simulation Tools
  - NCSIM (Cadence)
  - VSIM (Mentor Graphics)
  - VCS (Synopsys)

Event-Driven Simulation

- Event-driven simulation is designed for digital circuit characteristics

  Small number of signal values

  Relatively sparse activity over time

- Event-driven simulators try to update only those signals which change in order to reduce CPU time requirements

  An event is a change in a signal value

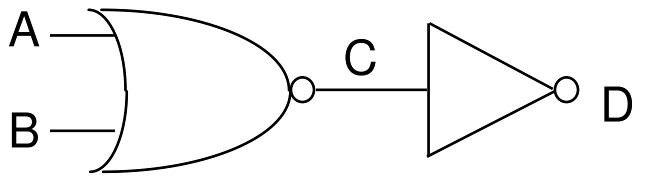
  A time-wheel is a queue of events

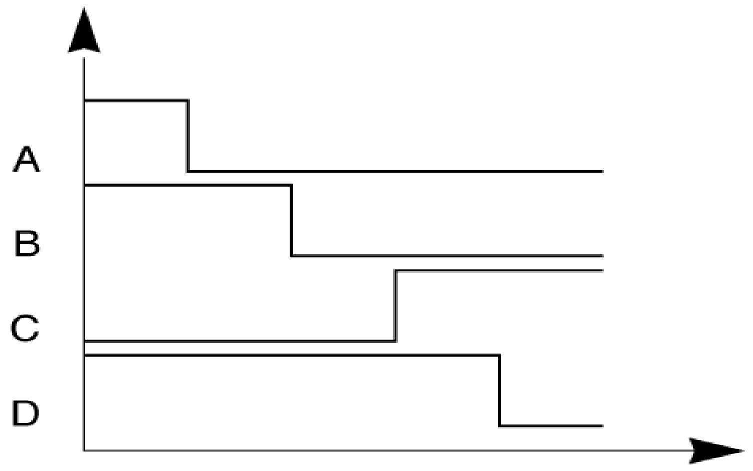
- Simulator traces structure of circuit to determine causality of events

  Event at input of one gate may cause new event at gate’s output

Switch Simulation

- Special type of event-driven simulation optimized for MOS transistors
  - Treats the transistor as a switch
  - Takes capacitance into account to model charge sharing
- Can also be enhanced to model the transistor as a resistive switch





Test Bench Example

```
entity test_bench is
end;

architecture test_reg3 of test_bench is
    signal d0, d1, d2, en, clk, q0, q1, q2 : bit;
begin
    dut : entity work.reg3(behav)
        port map ( d0, d1, d2, en, clk, q0, q1, q2 );
    stimulus : process is
    begin
        d0 <= '1';  d1 <= '1';  d2 <= '1';  wait for 20 ns;
        en <= '0';  clk <= '0';  wait for 20 ns;
        en <= '1';  wait for 20 ns;
        clk <= '1';  wait for 20 ns;
        d0 <= '0';  d1 <= '0';  d2 <= '0';  wait for 20 ns;
        …
        wait;
    end process stimulus;
end;
```

Verification

- To test a refinement of a design
  - Low-level structural model must be functionally the same as a corresponding behavioral model
- To include two instances of a design in the test bench
  - To stimulate both with same test values on inputs
  - To compare values of outputs for equality
- To take account of timing differences
  - Zero delay
  - Unit delay
  - Gate delay
  - RC delay

Verification Example

```
architecture regression of test_bench is
        signal d0, d1, d2, d3, en, clk : bit;
        signal q0a, q1a, q2a, q3a, q0b, q1b, q2b, q3b : bit;
begin
        dut_a : entity work.reg4(struct)
                port map ( d0, d1, d2, d3, en, clk, q0a, q1a, q2a, q3a );
        dut_b : entity work.reg4(behav)
                port map ( d0, d1, d2, d3, en, clk, q0b, q1b, q2b, q3b );
        stimulus : process is
        begin
                d0 <= ’1’;  d1 <= ’1’;  d2 <= ’1’;  d3 <= ’1’;  wait for 20 ns;
                en <= ’0’;  clk <= ’0’;  wait for 20 ns;
                en <= ’1’;  wait for 20 ns;
                clk <= ’1’;  wait for 20 ns;
                …
                wait;
        end process stimulus;
        ...
```

Verification Example

```
        …
        verify : process is
        begin
                wait for 10 ns;
                assert q0a = q0b and q1a = q1b and q2a = q2b and q3a = q3b
                        report ”implementations have different outputs”
                        severity error;
                wait on d0, d1, d2, d3, en, clk;
        end process verify;
end architecture regression;
```

Annotation

- Standard Delay Format (SDF) annotation
  - Design timing is stored in an SDF file
  - Used to iteratively improve design
- Updates a more-abstract design with information from later design stages
  - Annotation of logic schematic with extracted parasitic resistances and capacitances
- Back annotation requires tools to know more about each other
  - Simulation tools
  - Synthesis tools
  - Layout tools

Standard Delay Format

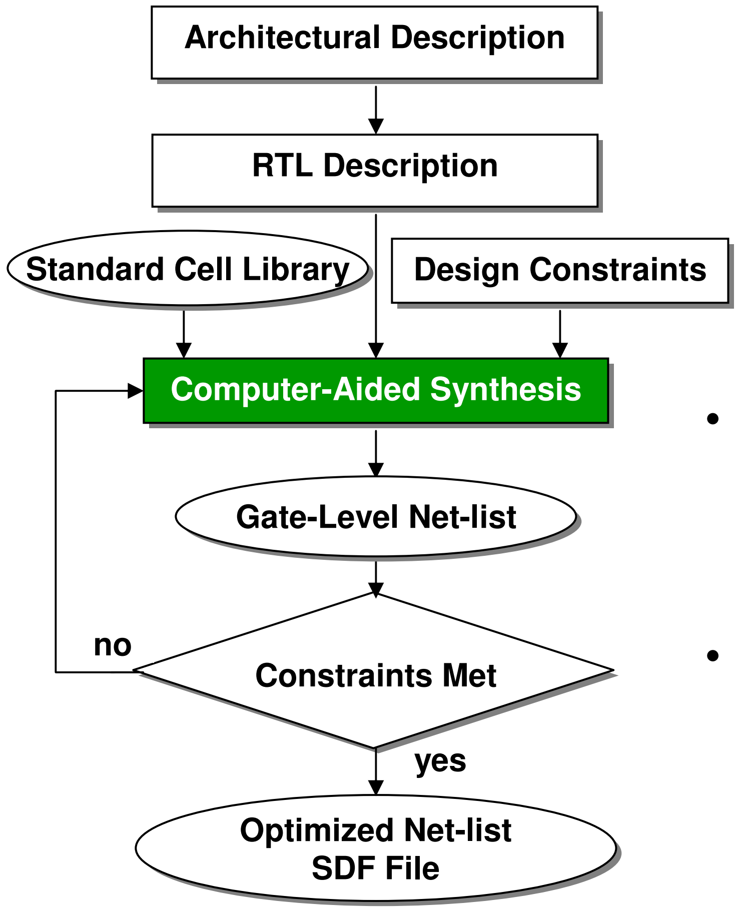
```
(DELAYFILE
(SDFVERSION "OVI 1.0")
(DESIGN "tcp_1_chip")
(DATE "Fri Apr 30 09:48:22 2004")
(VENDOR "cdr3synPwcslV225T125")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "2003.06")
(DIVIDER /)
(VOLTAGE 2.25:2.25:2.25)
(PROCESS)
(TEMPERATURE 125.00:125.00:125.00)
(TIMESCALE 1ns)
(CELL
  (CELLTYPE "tcp_1_chip")
  (INSTANCE)
  (DELAY
    (ABSOLUTE
    (INTERCONNECT U5/x U81/a (0.000:0.000:0.000))
    (INTERCONNECT U73/x U74/a (0.000:0.000:0.000))
...
    )
  )
)
```

```
(CELL
  (CELLTYPE "exnor2_1")
  (INSTANCE i_aes_wr/U_ALG/U6533)
  (DELAY
    (ABSOLUTE
    (IOPATH a x (0.662:1.045:1.045) (0.682:1.076:1.076))
    (IOPATH b x (1.379:1.416:1.416) (1.454:1.492:1.492))
    )
  )
)
...
(CELL
  (CELLTYPE "mux2_2")
  (INSTANCE
       i_mips/u0/ejt_tap\/pa_addr_reg_next\/bit_00i/U1)
  (DELAY
    (ABSOLUTE
    (IOPATH d0 x (0.395:0.395:0.395) (0.464:0.464:0.464))
    (IOPATH d1 x (0.387:0.403:0.403) (0.447:0.477:0.477))
    (IOPATH sl x (1.768:1.781:1.781) (1.879:1.892:1.892))
    )
  )
)
```

Logic Synthesis

- Logic Synthesis Flow
- Optimization
- Technology Mapping
- Low-Power Techniques

Logic Synthesis Flow



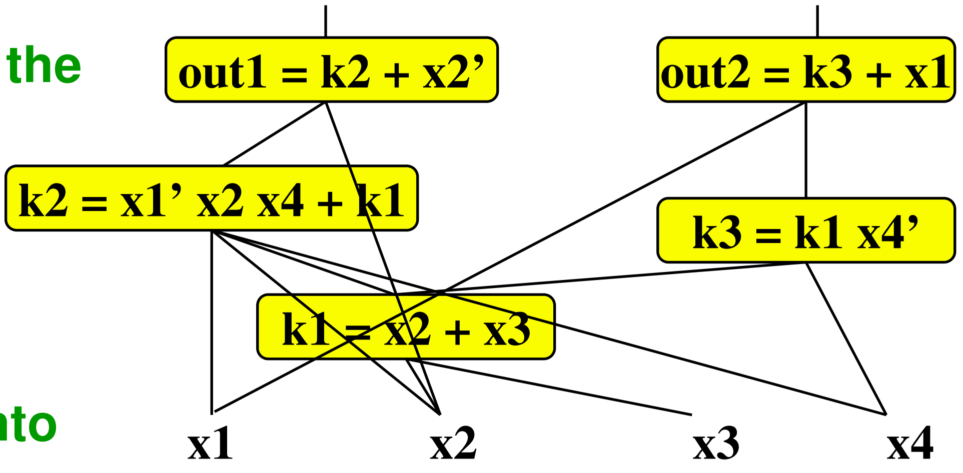
- Goal is to create a logic gate network which performs a given set of functions
  - Input is Boolean formulae
  - Output is gates implementing Boolean functions
  - Several iterations needed for generation of the optimized gate-level description
- Logic synthesis
  - Maps onto available gates
  - Restructures for delay, area, testability, power, etc.
- Automated logic synthesis enables
  - Enormous reduction of the time needed for conversion of a design from high-level to gate-level description
  - Saving of resources for architectural and RTL design, and optimization of the standard cell library

Logic Synthesis Phases

- Technology-independent optimizations
  - A Boolean network is the main representation of the logic functions
    - Each node can be represented as sum-of-products (or product-of-sums)
    - Functions in the network need not correspond to logic gates
- Technology mapping (library binding)
  - Design transformation from technology-independent to technology-dependent
- Technology-dependent optimizations
  - Work in the available set of logic gates

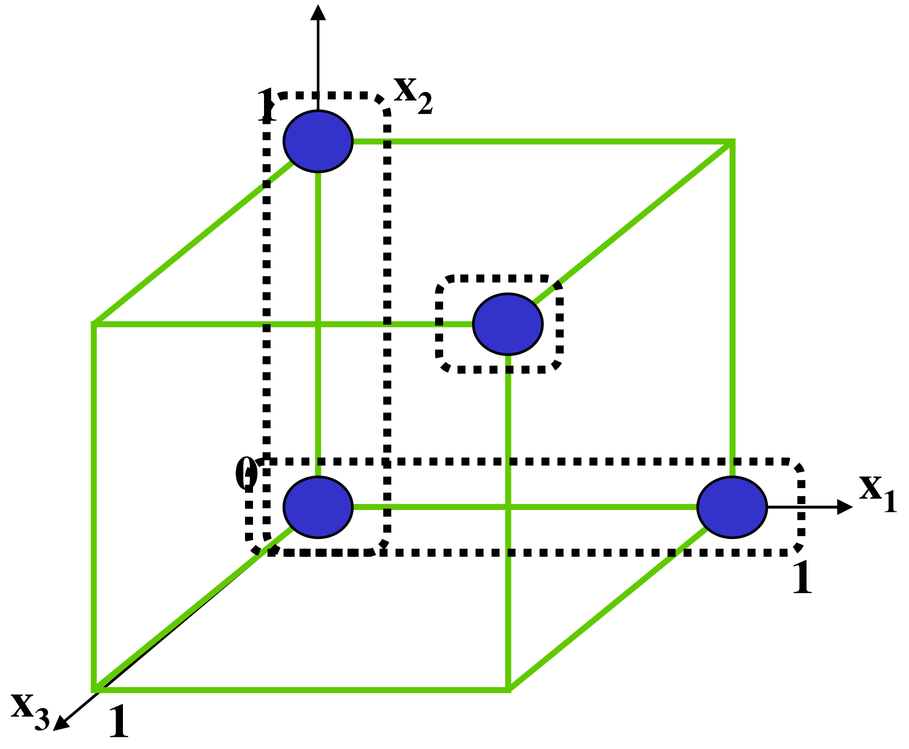
Technology-Independent Optimization

- Area is estimated by number of literals
  - Literal is true or complement form of a variable
- Simplification
  - Rewrites a node to reduce the number of literals in it
- Network restructuring
  - Introduces new nodes for common factors
  - Collapses several nodes into one new node
- Delay restructuring
  - Changes factorization to reduce path length



Covers and Cubes

- Function is defined by
  - On-set: set of inputs for which output is 1
  - Off-set: set of inputs for which output is 0
  - Don't-care-set: set of inputs for which output is don't-care
- Each way to write a function as a sum-of-products is a cover
  - It covers the on-set
- A cover is composed of cubes
  - Cubes are product terms that define a subspace cube in the function space



Covers and Optimizations

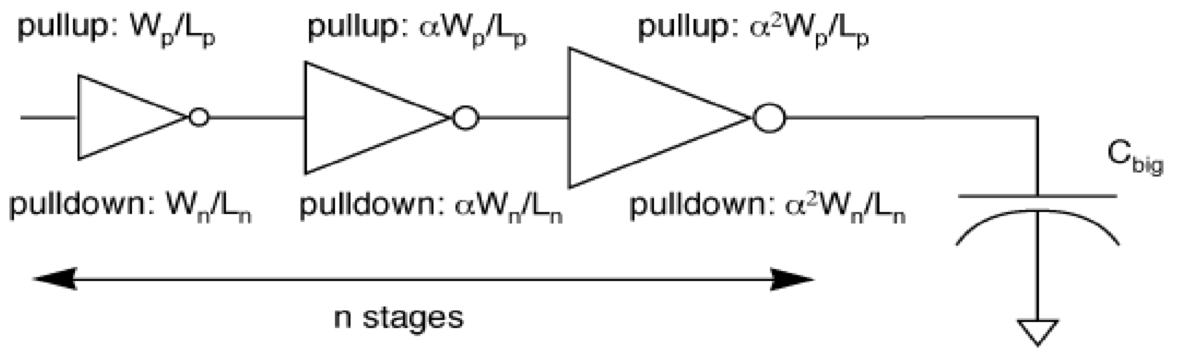
- Larger cover
  - x1’ x2’ x3’ + x1 x2’ x3’ + x1’ x2 x3’ + x1 x2 x3
  - Requires four cubes (12 literals)
- Smaller cover
  - x2’ x3’ + x1’ x3’ + x1 x2 x3
  - Requires three cubes (7 literals)
  - x1’ x2 x3’ is covered by two cubes
- Don’t-cares
  - Can be implemented in either on-set or off-set
  - Provide the greatest opportunities for minimization in many cases
- Espresso
  - A two-level logic optimizer
  - Expands, makes irredundant and reduces
  - Optimization loop refines cover to reduce its size

Factorization

- Based on division
  - Formulate candidate divisor
  - Test how it divides into the function
  - If g = f/c, we can use c as an intermediate function
- Algebraic division
  - Doesn’t take into account Boolean simplification
  - Less expensive then Boolean division
- Three steps
  - Generate potential common factors and compute literal savings if used
  - Choose factors to substitute into network
  - Restructure the network to use the new factors
- Algebraic/Boolean division is used to implement first step

Optimal Sizing

- Sometimes, large loads must be driven
  - Off-chip or by long wires on-chip
- Sizing up the driver transistors only pushes back the problem
  - Driver now presents larger capacitance to earlier stage
- Use a chain of inverters
  - Each driver has transistors larger than previous stage
  - α is the driver size ratio, $C_{big}/C_d = \alpha^n$, $\ln(C_{big}/C_d) = n \ln\alpha$
- Minimize total delay through the driver chain
  - $t_{tot} = \ln(C_{big}/C_d)(\alpha/\ln\alpha)t_d$
- Optimal driver size ratio is $\alpha_{opt} = e$
- Optimal number of stages is $n_{opt} = \ln(C_{big}/C_d)$


pullup: W_p/L_p
pullup: $\alpha W_p/L_p$
pullup: $\alpha^2 W_p/L_p$
C_{big}
pulldown: W_n/L_n
pulldown: $\alpha W_n/L_n$
pulldown: $\alpha^2 W_n/L_n$
n stages

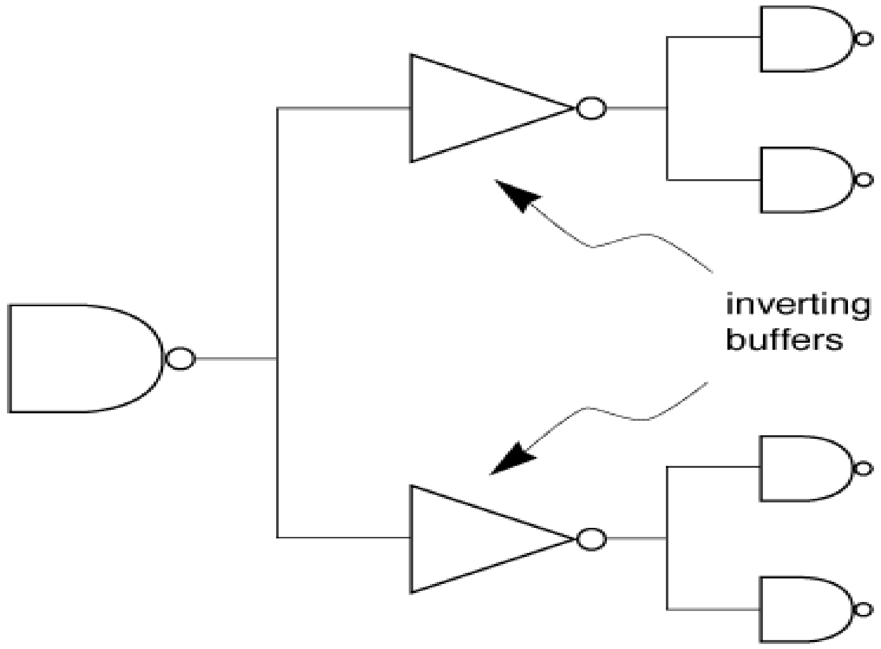
Driving Large Fan-out

- **Fan-out adds capacitance**
- **Increase sizes of driver transistors**

  Must take into account rules for driving large loads

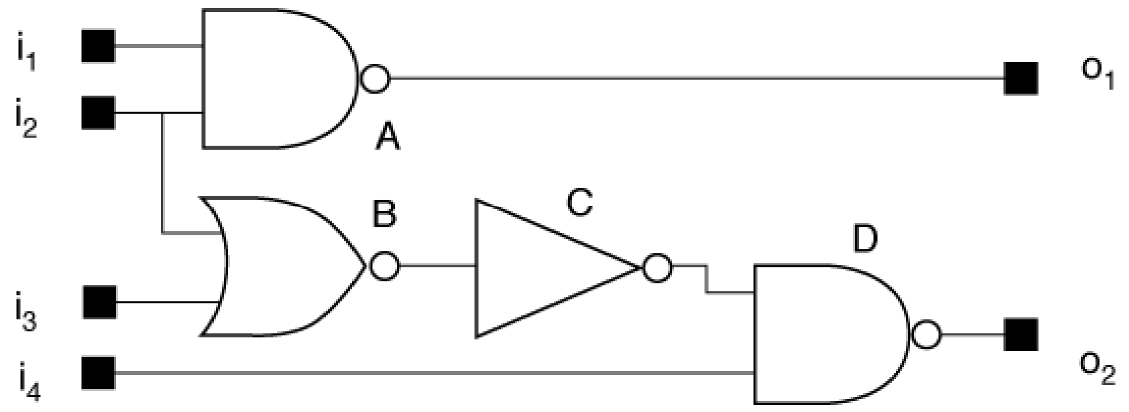
- **Add intermediate buffers**

  This may require/allow restructuring of the logic



Path Delay

- Network delay is measured over paths through network
  - Can trace a causality chain from inputs to worst-case output
- Critical path creates longest delay
  - Can trace transitions which cause delays that are elements of the critical path delay
- To reduce circuit delay, speed up the critical path
  - Reducing delay off the path doesn't help
- There may be more than one path of the same delay
  - Must sp



Logic Transformations

- Rewrite by using sub-expressions

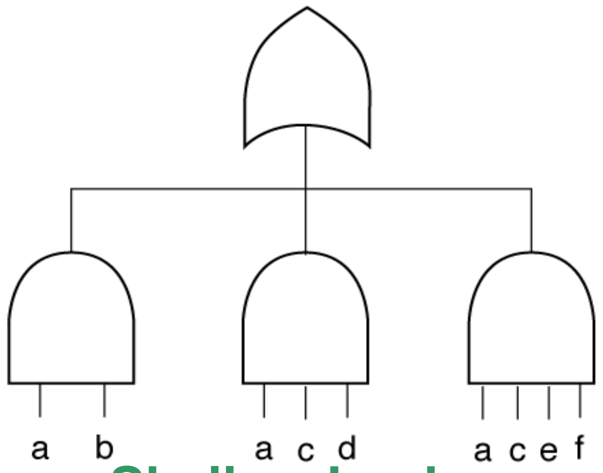
  Logic rewrites may affect gate placement

- Flattening logic

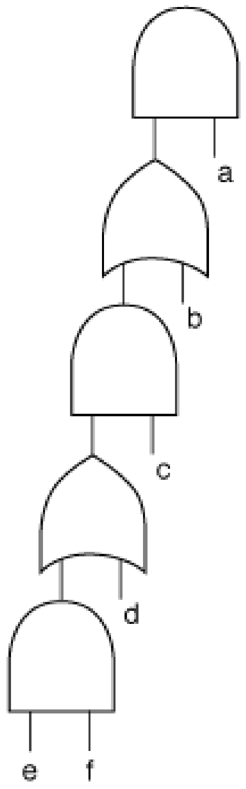
  Increases gate fan-in

- Logic synthesis programs

  Transform Boolean expressions into logic gate networks in a particular library



Shallow Logic

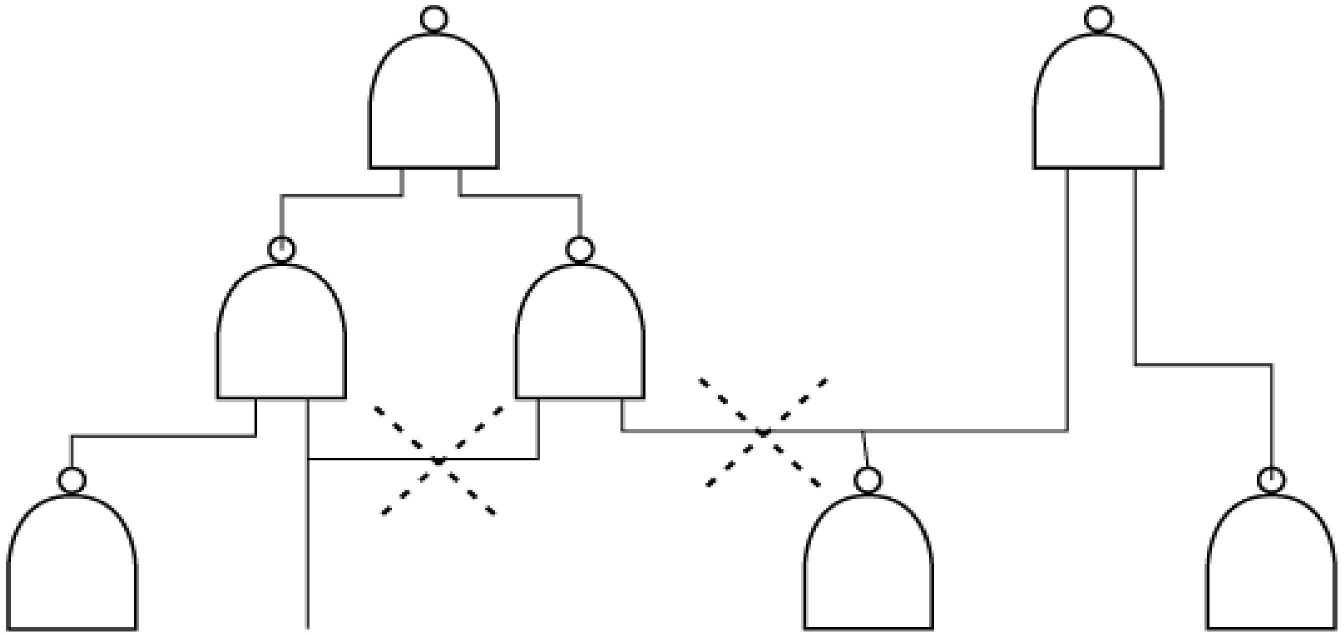


Deep Logic

Logic Optimization

- Optimization goals
  - Minimize area, meet delay constraint
- Technology-independent optimization
  - Works on Boolean expression equivalent
  - Estimates size based on number of literals
  - Uses factorization, resubstitution, minimization, etc.
  - Uses simple delay models
- Technology-dependent optimization
  - Maps Boolean expressions into a particular cell library
  - May perform some optimizations on addition to simple mapping
  - Allows more accurate delay models

Breaking into Trees

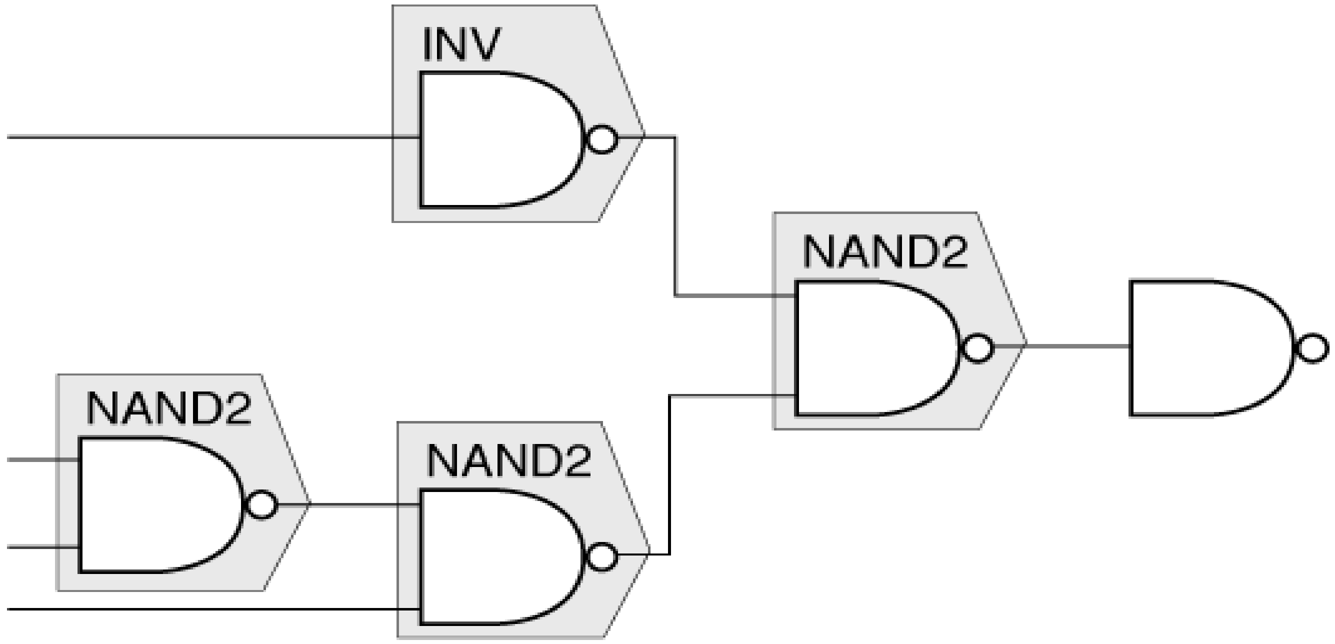


not optimal, but reasonable cuts usually work well

Technology Mapping

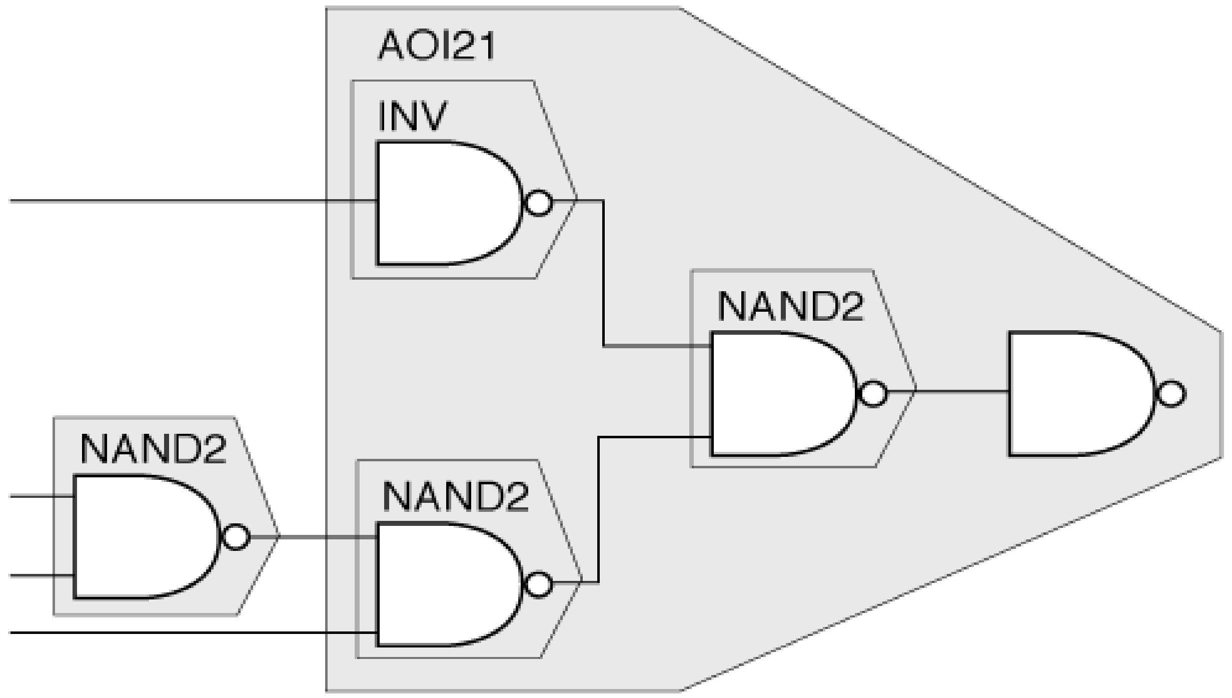
- Rewrites Boolean network
  - In terms of available logic functions
- Optimizes for
  - Area
  - Delay
- Can be viewed as a pattern matching problem
  - Find pattern match which minimizes area/delay cost
- Procedure
  - Write Boolean network in canonical NAND form
  - Write each library gate in canonical NAND form
  - Assign cost to each library gate
  - Use dynamic programming to select minimum-cost cover of network by library gates

Mapping Example



after three levels of matching

Mapping Example



after four levels of matching

Low Power Techniques

- Architecture-driven supply voltage scaling

  Add extra logic to increase parallelism so that system can run at lower frequency

  Power improvement for n parallel units over Vref

  $P_n(n) = [1 + C_i(n)/nC_{ref} + C_x(n)/C_{ref}](V/V_{ref})$

- Dynamic voltage and frequency scaling

  Decreased to parts of the circuit where it does not adversely affect the performance

  Dynamic scaling is regulated by software based on system load

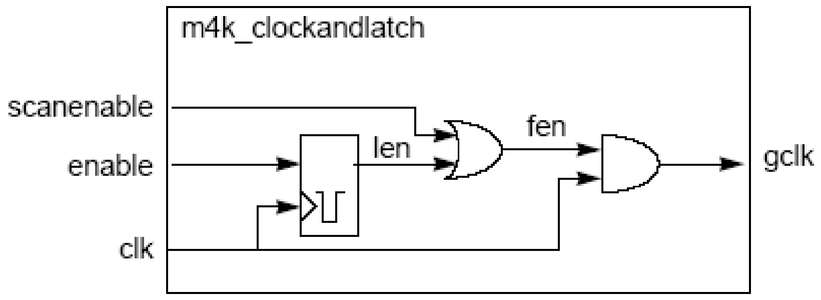
- Reducing capacitances

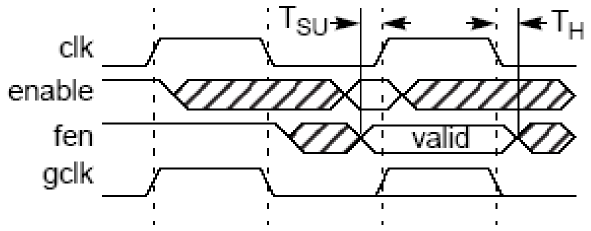
  Parasitic capacitances of the transistors

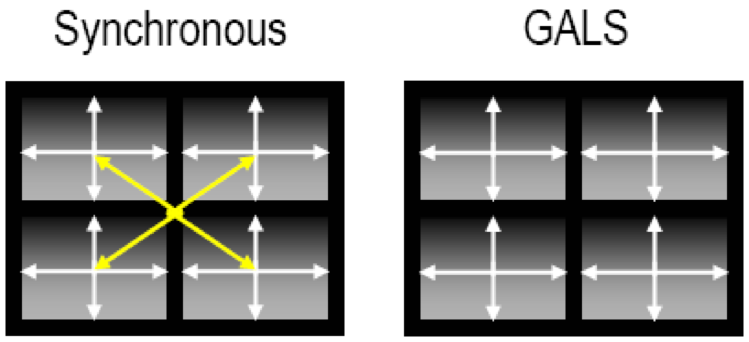
  Parasitic capacitances of the wires

Low Power Techniques

- **Reducing switching activity**
  - **Deactivate the clock to unused registers (clock gating)**
  - **Deactivate signals if not used (signal gating)**
  - **Deactivate V_{DD} for unused hardware blocks (power gating)**
- **Distributed clocks: Globally asynchronous locally synchronous**
  - **Eliminating centrally synchronous clocks and utilizing local clocks**
  - **Distinct local clocks, possibly running at different frequencies**







Design for Testability

- DFT Methods
- Scan Design
- Test Pattern Generation
- Built-In Self-Test

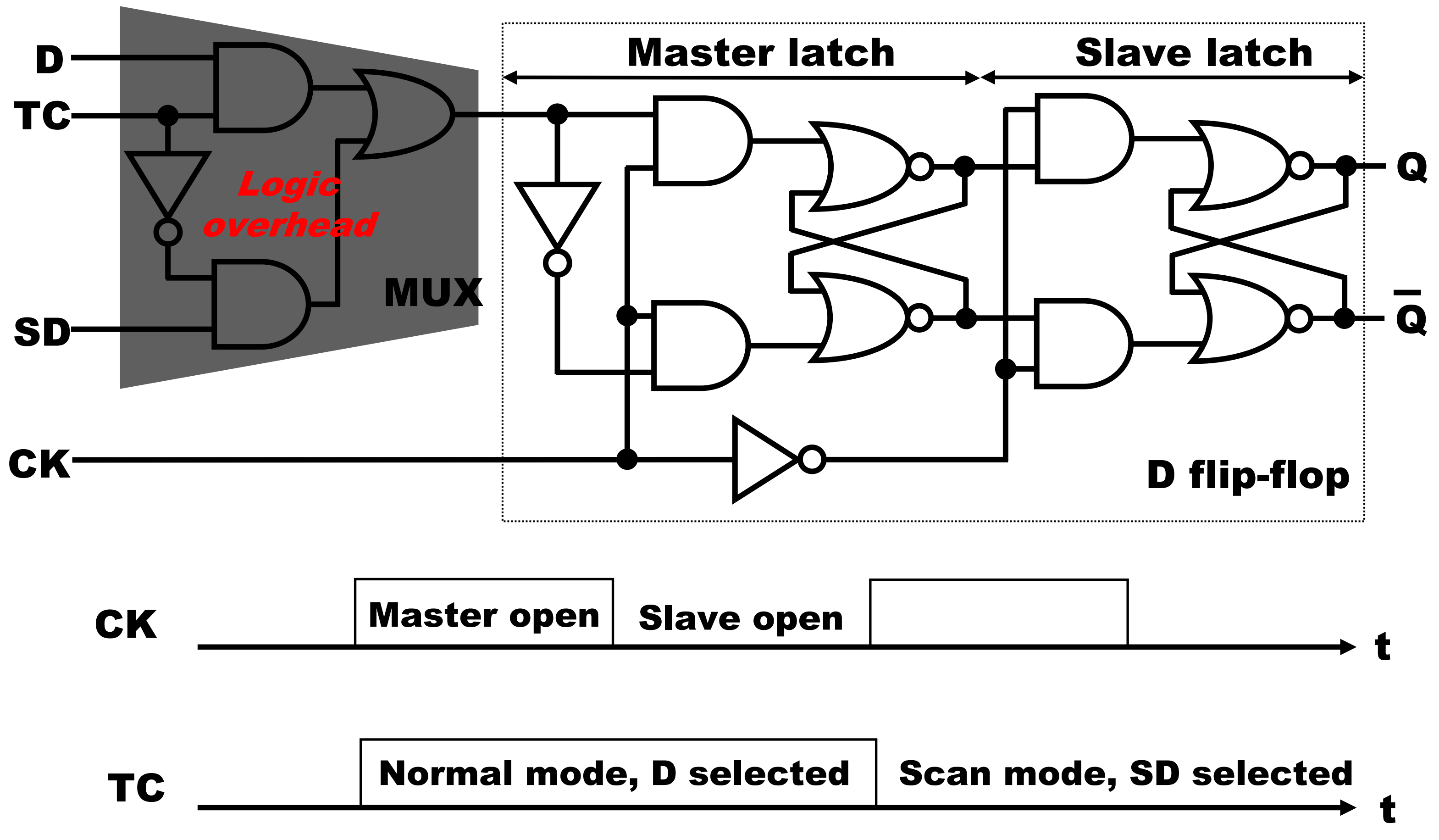
Design for Testability Methods

- Make the system as testable as possible
  - Keep minimum cost in hardware and testing time
  - Use knowledge of architecture to help in selection of testability points
  - Modify architecture to improve testability
- DFT for digital circuits
  - Ad-hoc methods
    - *Avoid asynchronous feedback*
    - *Make flip-flops initializable*
    - *Avoid redundant gates, large fan-in gates and gated clocks*
    - *Provide test control for difficult-to-control signals*
    - *Consider ATE requirements (tri-states, etc.)*
  - Structured methods
    - *Scan Design*
    - *Built-in self-test (BIST)*
    - *Boundary scan*

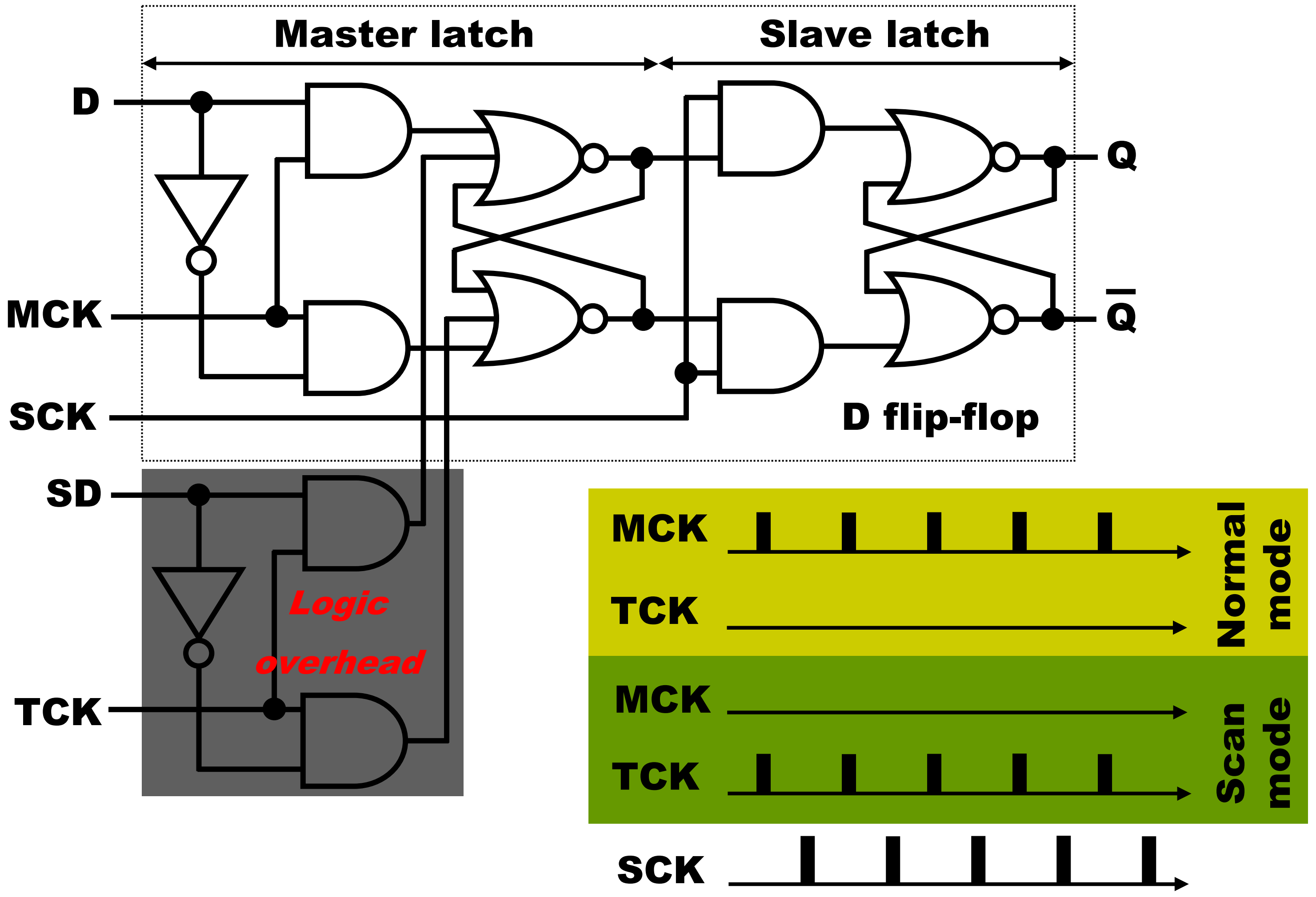
Scan Design

- Circuit is designed using pre-specified design rules
- Test structure (hardware) is added to the verified design
  - Add a *test control* (TC) primary input
  - Replace flip-flops by *scan flip-flops* (SFF) and connect to form one or more shift registers (scan-chains) in the test mode
  - Make input/output of each scan-chain controllable/observable from primary input/primary output
- Use combinational ATPG to obtain tests for all testable faults in the combinational logic
- Add shift register tests and convert ATPG tests into scan sequences for use in manufacturing test
- Full scan is expensive
  - Must roll out and roll in state many times during a set of tests
- Partial scan selects some registers (not all) for scanability to reduce the chain length
  - Analysis is required to choose which registers are best for scan

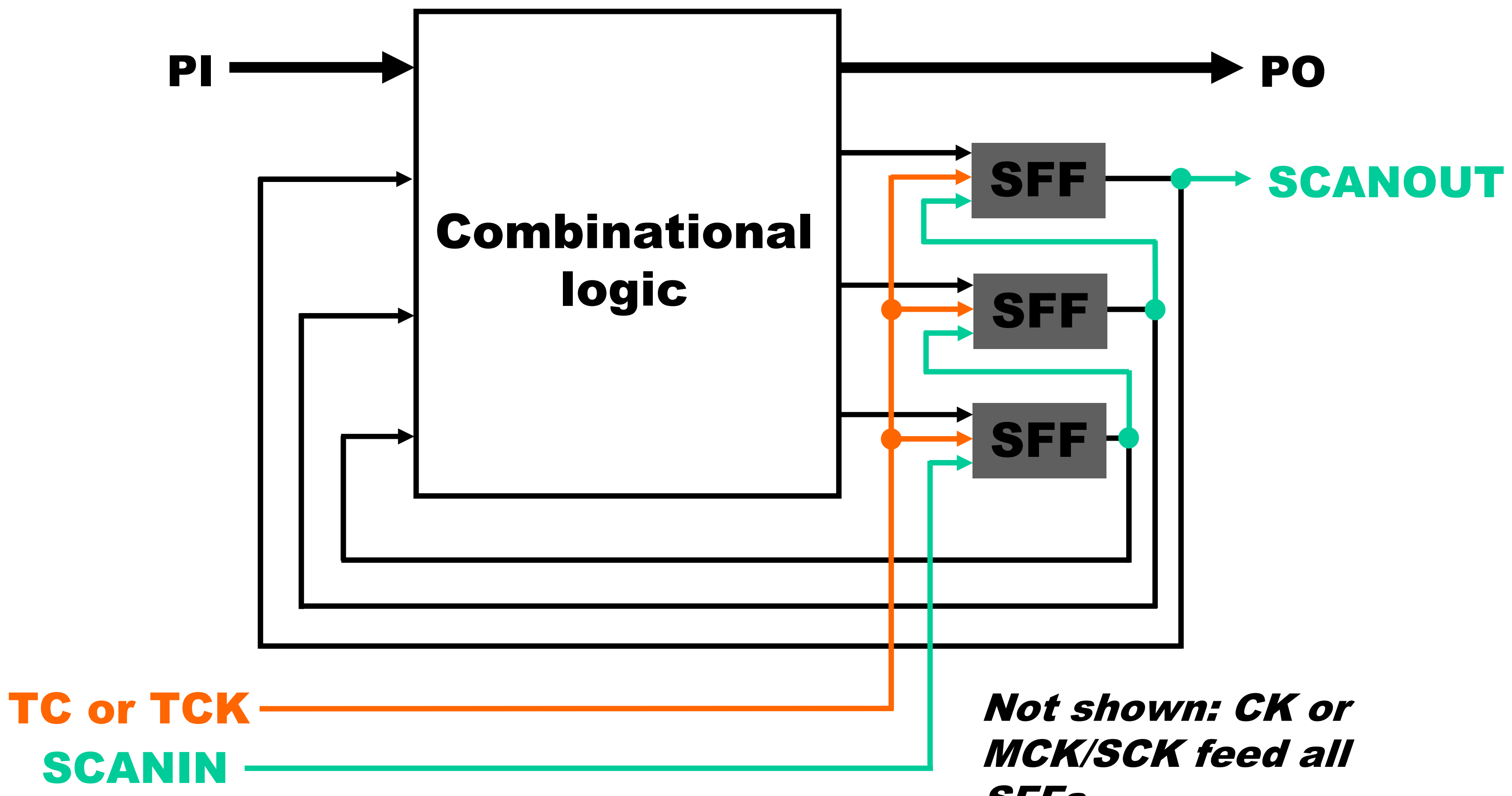
Scanable Flip-Flop



Level-Sensitive Scanable Flip-Flop

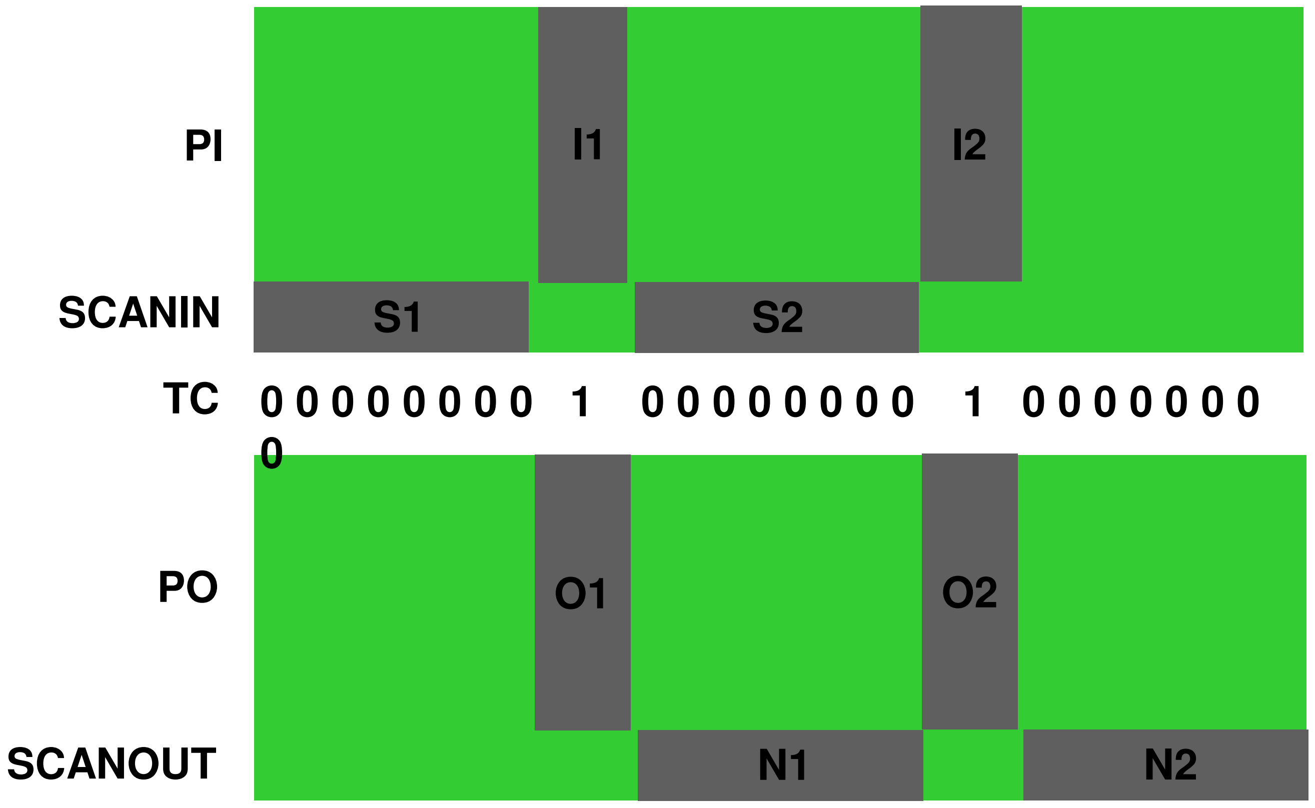


Scan Structure



Not shown: CK or MCK/SCK feed all SFFs

Combinational Test Vectors



Sequence length = $(n_{\text{comb}} + 1)\, n_{\text{sff}} + n_{\text{comb}}$ *clock periods*

n_{comb} *= number of combinational vectors*

n_{sff} *= number of scan flip-flops*

Testing Scan Chain

- Scan-chain must be tested prior to application of scan test sequences
- A shift sequence 00110011 . . . of length n_{sff}+4 in scan mode (TC=0)

  Produces 00, 01, 11 and 10 transitions in all flip-flops

  Observes the result at SCANOUT output
- Total scan test length

  $(n_{comb} + 2)\, n_{sff} + n_{comb} + 4$ *clock periods*
- Example

  2,000 scan flip-flops, 500 comb. vectors, total scan test length ~ 10^6 clocks
- Multiple scan-chains reduce test length

Testing and Faults

- Errors are introduced during manufacturing
  - Testing weeds out infant mortality
- Varieties of testing
  - Functional testing
  - Performance testing
- Fault model
  - Possible locations of faults
  - I/O behavior produced by the fault
  - With a fault model, we can test the network for every possible instantiation of that type of fault
  - It is difficult to enumerate all types of manufacturing faults
- Testing procedure
  - Set inputs
  - Observe output
  - Compare fault-free and observed output

Stuck-At-0/1 Faults

- Logic gate output is always stuck at 0 or 1 independently on input values
- Correspondence to manufacturing defects depends on logic family
- Experiments show that 100% stuck-at-0/1 fault coverage corresponds to high overall fault coverage

- Testing NAND

  Three ways to test it for stuck-at-0

  Only one way to test it for stuck-at-1

a	b	OK	SA0	SA1
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	0	1

- Testing NOR

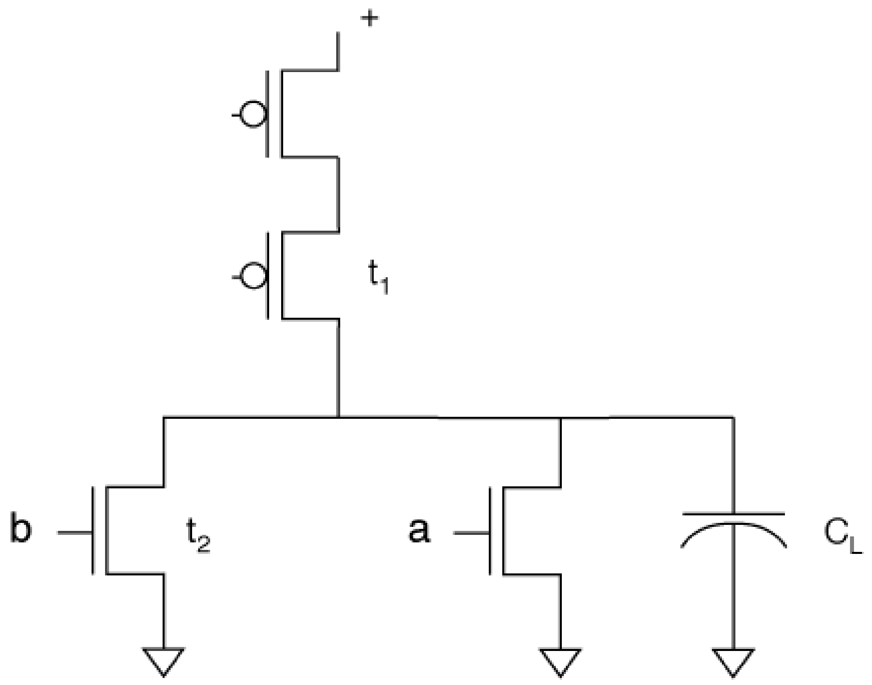
  Three ways to test it for stuck-at-1

  Only one way to test it for stuck-at-0

a	b	OK	SA0	SA1
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
1	1	0	0	1

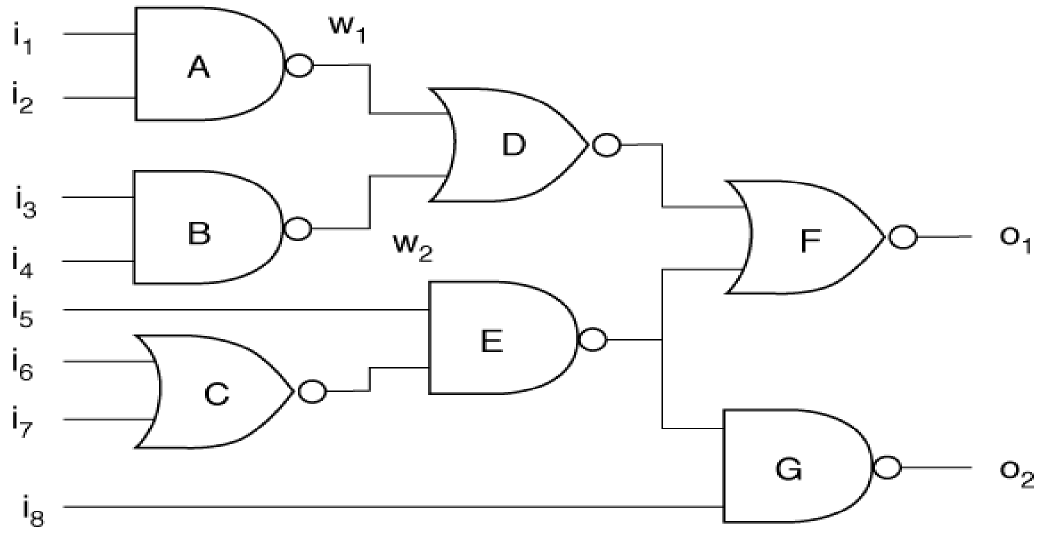
Stuck-At-Open/Closed Model

- Transistors always on/off
- t_1 is stuck open (switch cannot be closed)
  - No path from V_{DD} to output capacitance
- Testing requires two cycles
  - Must discharge capacitor
  - Try to operate t_1 to charge capacitor



Combinational Testing Example

- **Two parts of testing**
  - Controlling the inputs of (possibly interior) gates
  - Observing the outputs of (possibly interior) gates



- **Delay faults**
  - Gate delay model assumes that all delays are lumped into one gate
  - Path delay model takes into account delay of a path through network
  - Performance problems
  - Functional problems in some types of circuits

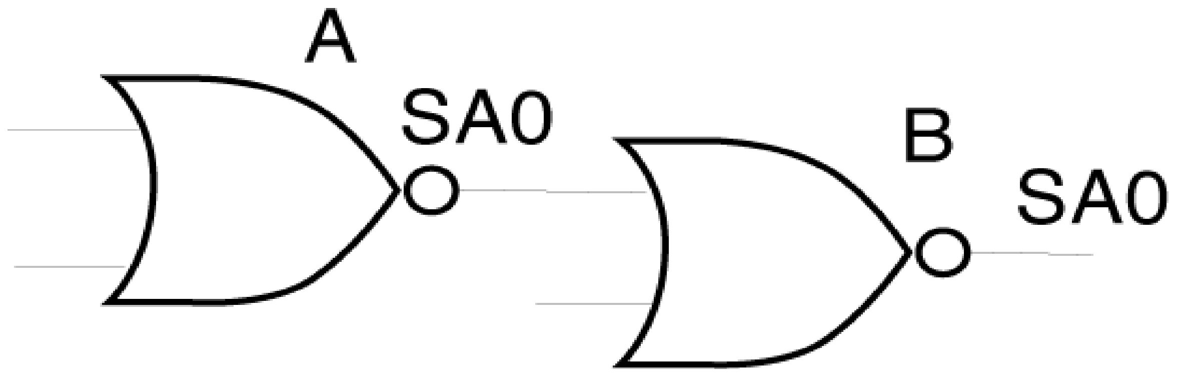
Testing Procedure

- Goal
  - Test gate D for stuck-at-0 fault
- First step
  - Justify 0 values on gate inputs
- Work backward from gate to primary inputs
  - w1 = 0 (A output = 0)
  - i1 = i2 = 1
- Observe the fault at a primary output
  - o1 gives different values if D is true/faulty
- Work forward and backward
  - F's other input must be 0 to detect true/fault
  - Justify 0 at E's output
- In general, may have to propagate fault through multiple levels of logic to primary outputs

Redundancy and Testing

- **Redundant logic can mask faults**
- **Testing NOR for SA0 requires setting both inputs to 0**
- **Network topology ensures that one NOR input (for instance *b*) will always be 1**
- **Function reduces to 0**

  f = ((a+b)’ + b)’ = (a + b)b’ = 0



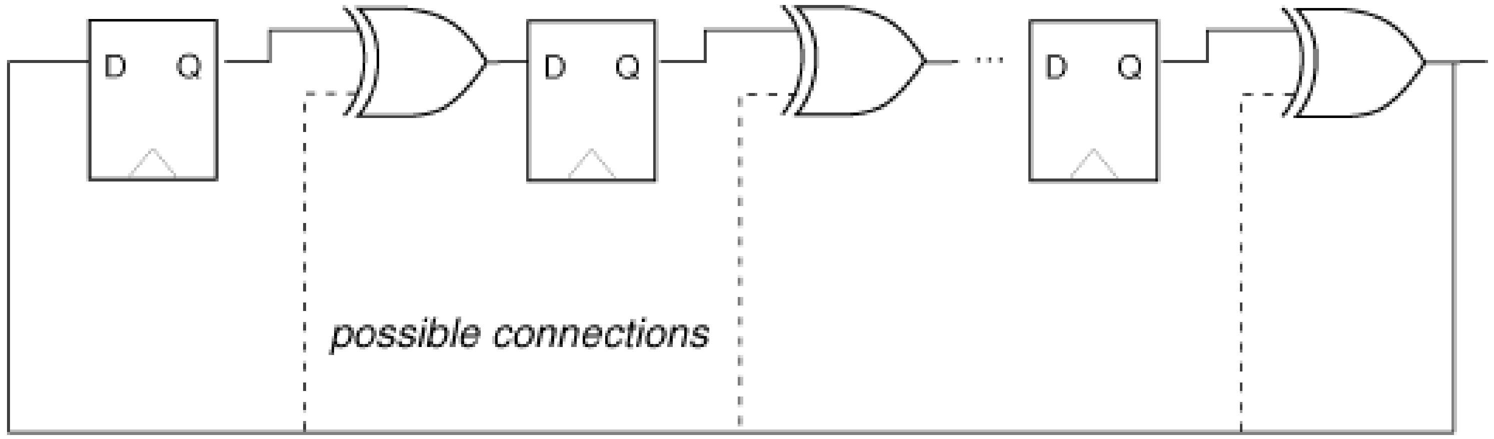
- **Redundant logic can introduce delay faults and other problems**

Test Pattern Generation

- Automatic test pattern generator (ATPG) generates a set of test vectors
  - Boolean network (combinational ATPG)
  - Sequential machine (sequential ATPG)
- D (from Discrepancy) allows us to quickly write fault
  - D value on a node means that good and faulty circuits have different values at that point
- If a test for a particular fault exists, D-algorithm will find it by an exhaustive search of all sensitized paths
  - Start at the faulty gate
  - Suppose initially a stuck-at fault on gate output
  - "Primitive D-cube of failure" (PDCF) of gate summarizes minimal assignment of input values to highlight fault
- Propagation D-cube (PDC) has D or D' on output and on at least one input
  - Summarizes "non-controlling" values for other inputs to allow propagation of D signal

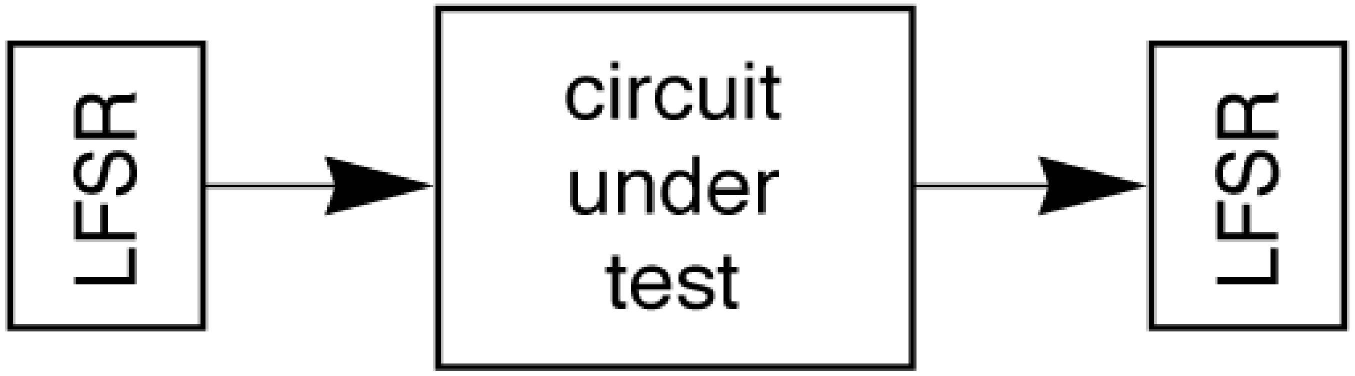
Built-In Self-Test (BIST)

- Includes on-chip machine responsible for
  - Generating tests
  - Evaluating correctness of tests
- Allows many tests to be applied
- Can't afford large memory for test results
  - Rely on compression and statistical analysis
- Uses a linear-feedback shift register (LFSR) to generate a pseudo-random sequence of bit vectors



BIST Architecture

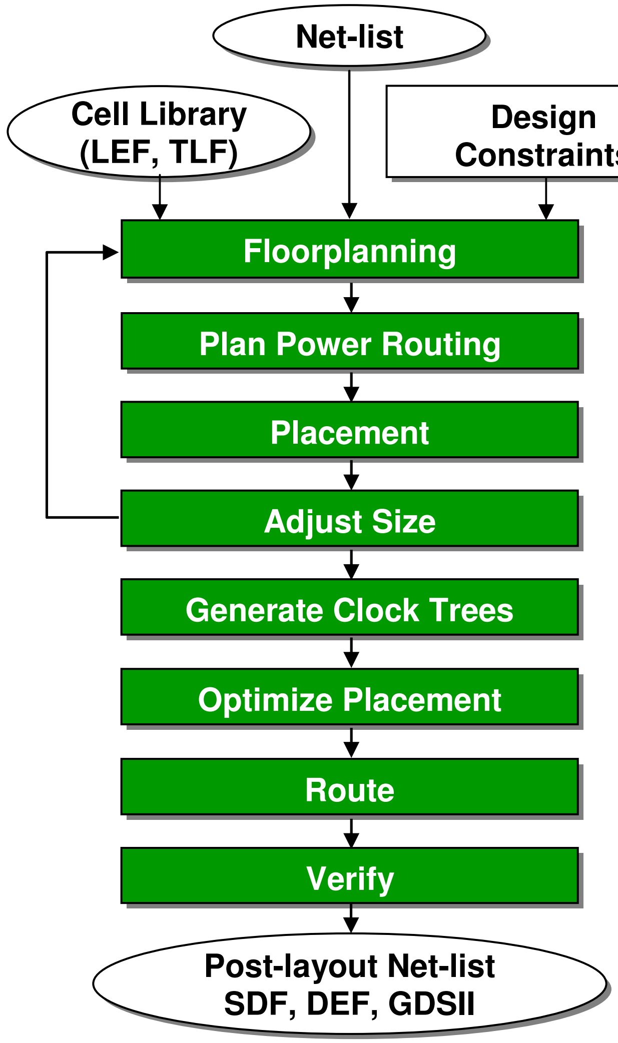
- One LFSR generates test sequence
- Another LFSR captures/compresses results
- Can store a small number of signatures which contain expected compressed results for valid system
- Usually used for testing memory blocks



Layout Generation

- Layout Generation Flow
- Design Rules
- Layout Tools
- Standard Cells
- Floorplanning
- Placement
- Routing
- Clock Tree
- Pads

Layout Generation Flow



- Library Exchange Format (LEF) files

  To create a library database (standard cells, I/O cells, and macro blocks)

- Timing Library Format (TLF) file

  Timing constraints

- General Constraints Format (GCF) file

  Design constraints

- Verilog net-list

  To create a design database

Layout Generation Flow

- **Floorplanning**

  To create a core area with rows (or columns) and I/O rows around the core area

- **Power planning and routing**

  To plan, modify and rout power paths, power rings and power stripes

- **Placement**

  An I/O constraints file may be used to place the I/O pads

  Block placement

  Cell placement

- **Size adjustment**

  To estimate the die size

  To resize the design to make it routable

Layout Generation Flow

- **Generating clock trees**

  The clock buffer space and clock net must be defined

  Generating clock trees is iterative process

  At this point, the physical net-list differ from the logical (original) net-list

- **Placement optimization**

  To resize gates and insert buffers to correct timing and electrical violations

- **Routing**

  To perform both global and final route on a placed design

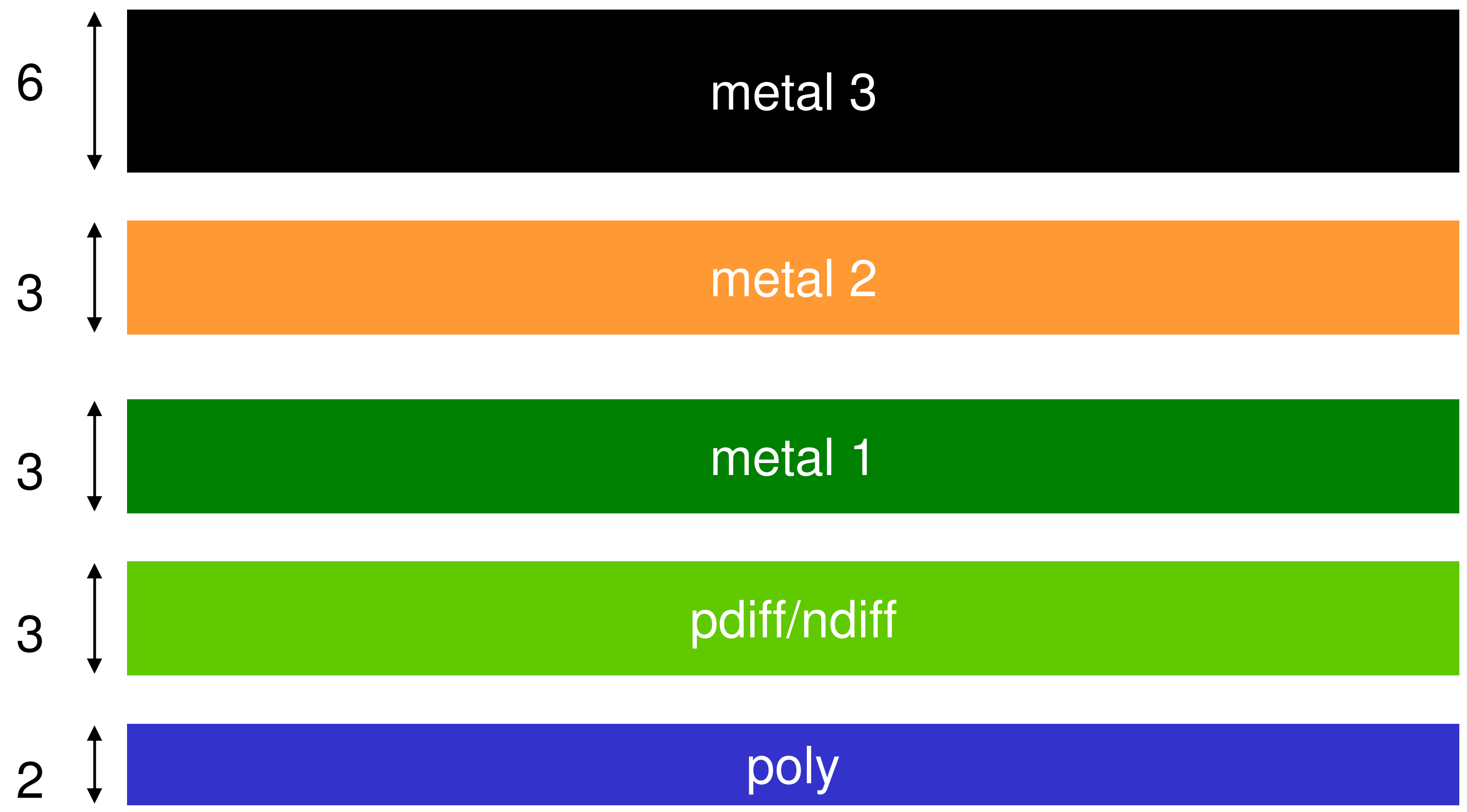
- **Verification**

  To check for shorts and design rule violations

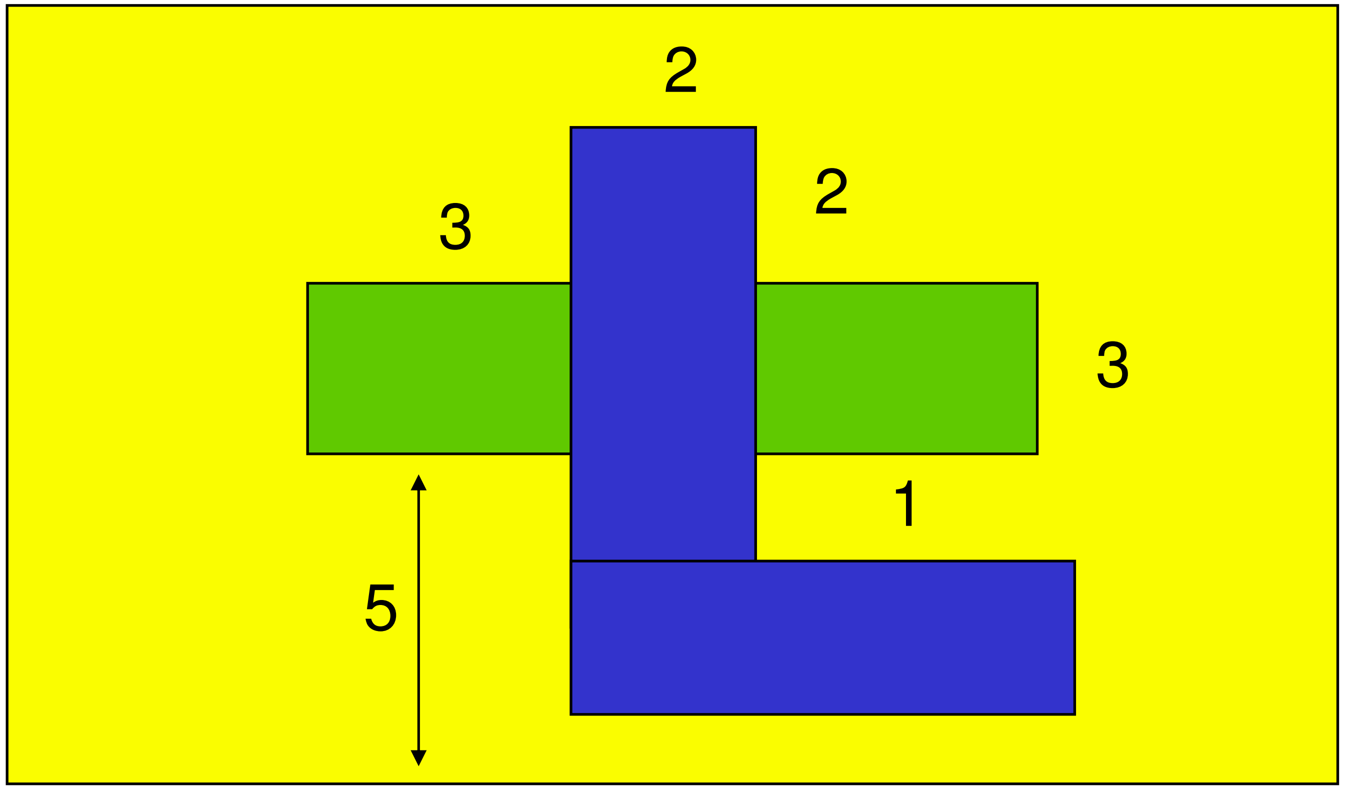
Design Rules

- Masks are tools for manufacturing
- Manufacturing processes have inherent limitations in accuracy
- Design rules specify geometry of masks which will provide reasonable yields
- Design rules are determined by experience
- MOSIS SCMOS
  - Designed to scale across a wide range of technologies
  - Designed to support multiple vendors
  - Designed for educational use
  - Fairly conservative
- Lambda (λ) design rules
  - Size of a minimum feature defines λ
  - Specifying λ particularizes the scalable rules
  - Parasitics are generally not specified in λ units

Wires

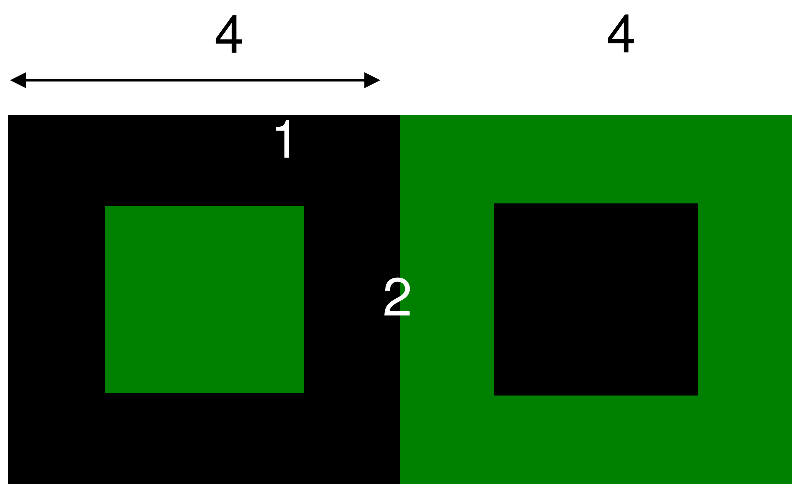


Transistors



Vias

- Types of via
  - Metal1/diff
  - Metal1/poly
  - Metal2/metal1
  - Metal3/metal2
  - ...
- Highest via
  - Cut: 3 x 3
  - Overlap by metal2: 1
  - Minimum spacing: 3
  - Minimum spacing to via1: 2



Spacings

- **Diffusion/diffusion**
  3
- **Poly/poly**
  2
- **Poly/diffusion**
  1
- **Via/via**
  2
- **Metal1/metal1**
  3
- **Metal2/metal2**
  4
- **Metal3/metal3**
  4

Overglass

- Cut in passivation layer

  Connection for bonding wire
- Minimum bonding pad

  100
- Pad overlap of glass opening

  6
- Minimum pad spacing to unrelated metal2/3

  30
- Minimum pad spacing to unrelated metal1, poly, active

  15

Layout Tools

- Layout editors are interactive tools
- Design rule checkers identify errors on the layout
- Circuit extractors extract the net-list from the layout
- Connectivity verification systems (CVS) compare extracted and original net-lists

  CADENCE Virtuoso's Layout-versus-Schematic (LVS) tool

- Standard cell layouts are created from pre-designed cells using the custom routing

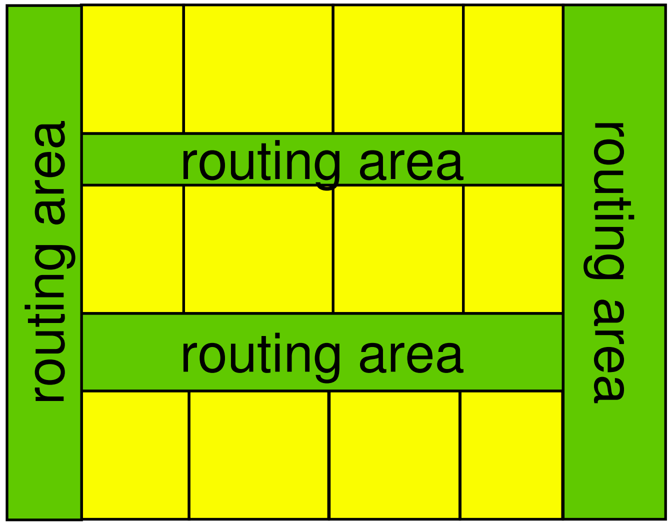
  Silicon Ensemble (CADENCE)

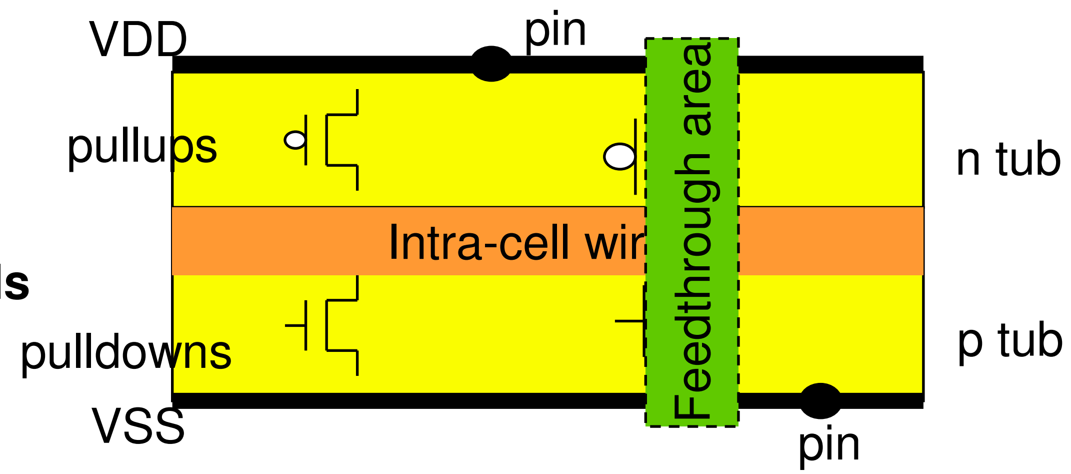
  Encounter (CADENCE)

  Physical Compiler (SYNOPSYS)

Standard Cell Layout

- Layout made of small cells
  - Gates, flip-flops, etc.
  - Cells are hand-designed
- Assembly of cells is automatic
  - Cells arranged in rows
  - Wires routed between and through cells
- Pitch is the height of a cell
  - All cells have same pitch, may have different widths
- VDD/VSS connections are designed to run through cells
- A feedthrough area allows routing wires over the cell





Floorplanning Strategy

- **Floorplanning must take into account**

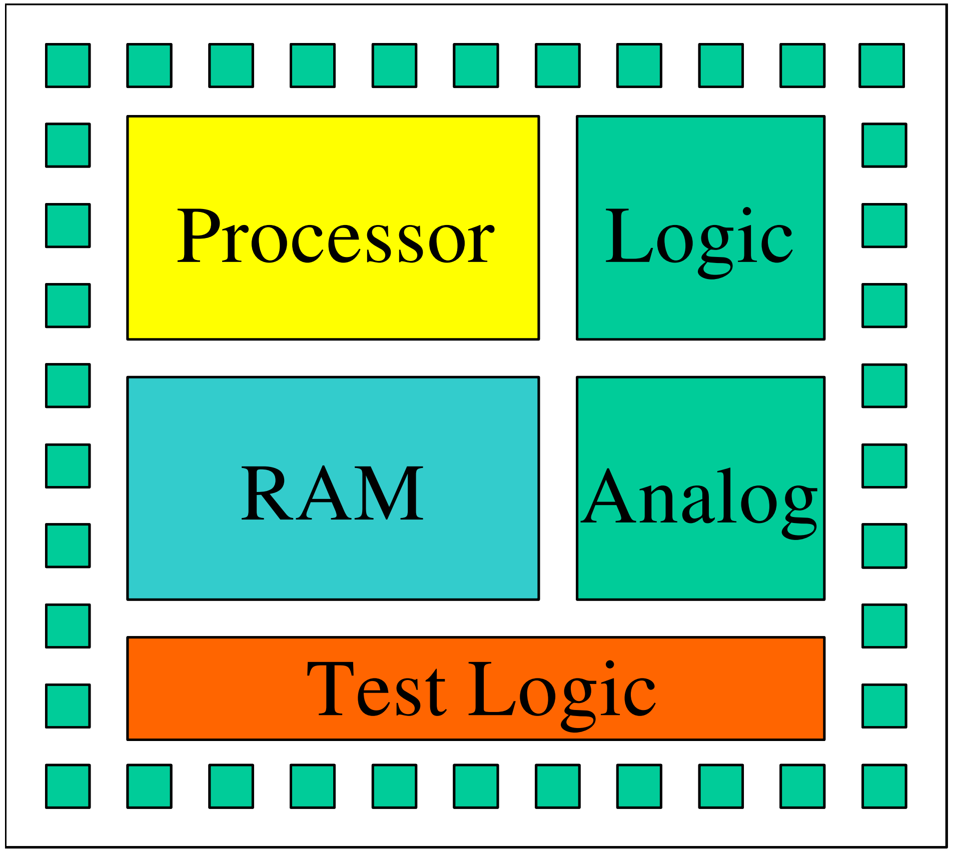
  Blocks of varying function, size, and shape

  Space allocation

  Signal routing

  Power supply routing

  Clock distribution

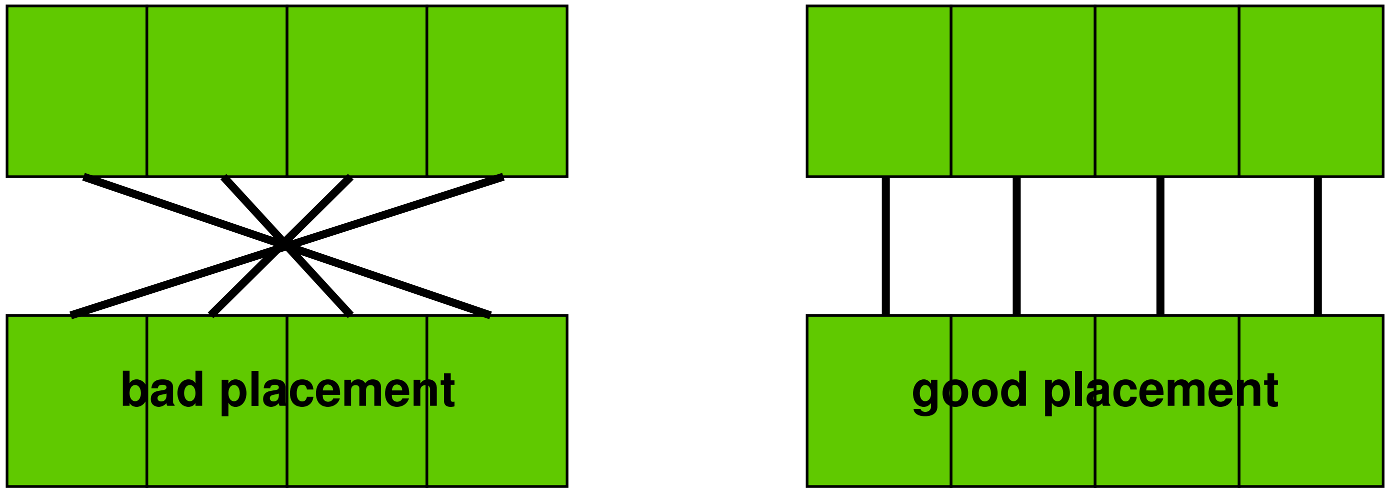


Floorplanning Tips

- Develop a wiring plan
  - Think about how layers will be used to distribute important wires
- Draw separate wiring plans for power and clocking
  - These are important design tasks which should be tackled early
- Sweep small components into larger blocks
  - A floorplan with a single NAND gate in the middle will be hard to work with
- Design wiring that looks simple
  - If it looks complicated, it is complicated
- Design planar wiring
  - Planarity is the essence of simplicity
  - Do it where feasible (and where it doesn't introduce unacceptable delay)

Placement Metrics

- Placement of components interacts with routing of wires
- Quality metrics for layout
  - Area and delay
- Area and delay determined in part by
  - Wiring
- How do we judge a placement without wiring?
  - Estimate wire length without actually performing routing

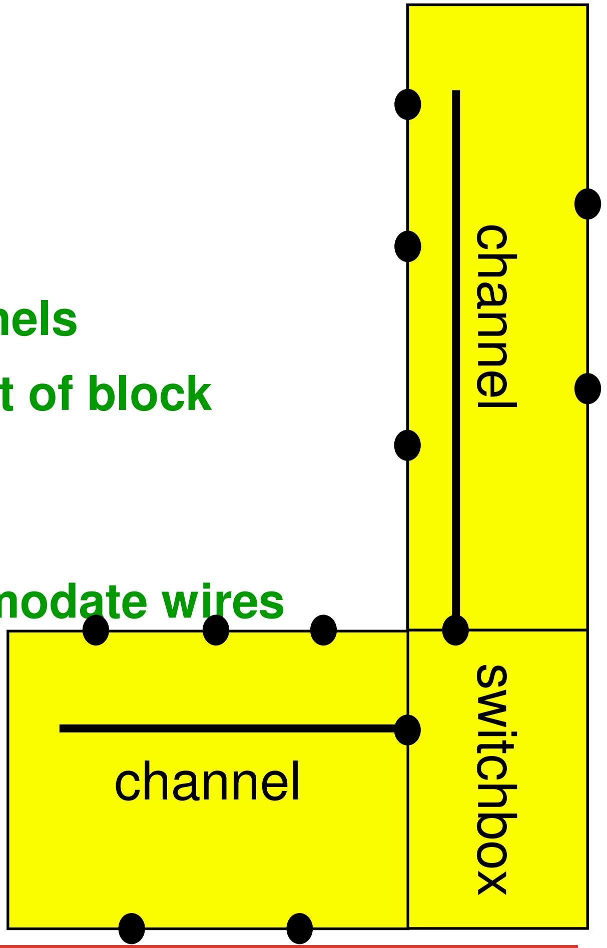


Placement Techniques

- To construct an initial solution
- To improve an existing solution
- Pairwise interchange is a simple improvement metric
  - Interchange a pair, keep the swap if it helps wire length
  - Heuristic determines which two components to swap
- Placement by partitioning
  - Works well for components of fairly uniform size
  - Partition net-list to minimize total wire length using min-cut criterion
- Kernighan-Lin Algorithm
  - Computes min-cut criterion, count total net-cut change
  - Exchanges sets of nodes to perform hill-climbing finding improvements where no single swap will improve the cut
  - Recursively subdivide to determine placement detail

Routing

- Major phases in routing
  - Global routing assigns nets to routing areas
  - Detailed routing designs the routing areas
- Net ordering determines quality of result
  - Net ordering is a heuristic
- Blocks and wiring
  - Blocks divide wiring area into routing channels
  - Large wiring areas may force rearrangement of block placement
- Channel routing
  - Channel grows in one dimension to accommodate wires
  - Pins generally on only two sides
- Switchbox routing
  - Box cannot grow in any dimension
  - Pins are on all four sides

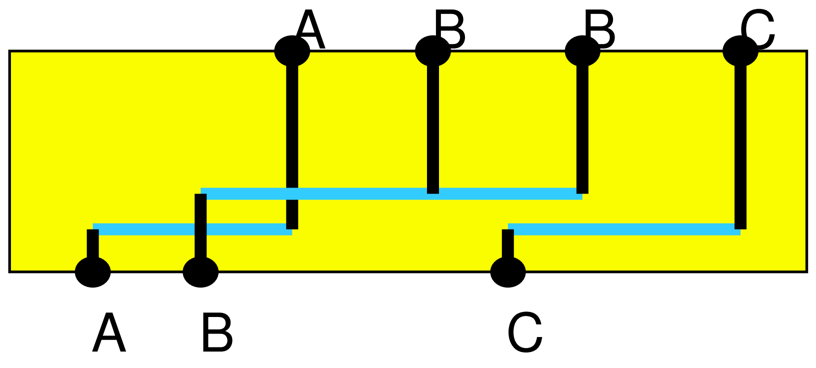


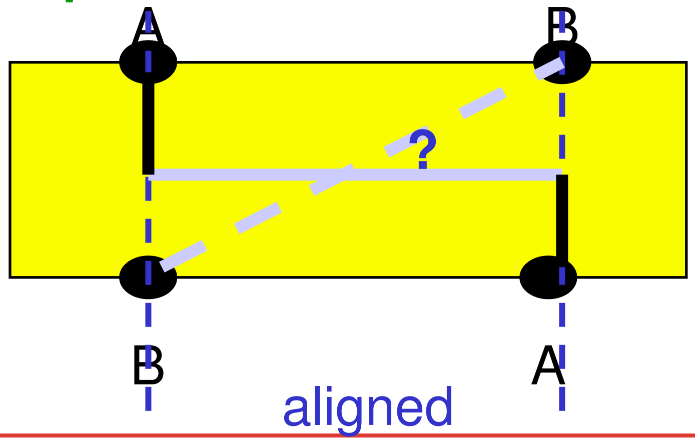
Routing Channels

- Tracks form a grid for routing
  - Spacing between tracks is center-to-center distance between wires
  - Track spacing depends on wire layer used
- Density (vertical and horizontal)
  - Gives the number of wire segments crossing a vertical/horizontal grid segment
- Different layers are used for horizontal and vertical wires
  - Horizontal and vertical wires can be routed relatively independently
- Placement of cells determines placement of pins
- Pin placement determines difficulty of routing problem

Left-Edge Algorithm

- Assumes one horizontal segment per net
- Sweep pins from left to right
  - Assign horizontal segment to lowest available track
- Limitations
  - Some combinations of nets require more than one horizontal segment per net (a dog-leg wire)
- Aligned pins form vertical constraints
  - Wire to lower pin must be on lower track
  - Wire to upper pin must be above lower pin's wire





Global and Detailed Routing

- Global routing

  Assign wires to paths through channels

  Don’t worry about exact routing of wires within channel

  Can estimate channel height using congestion

- Detailed routing

  Dog-leg router breaks net into multiple segments as needed

  Minimize number of dog-leg segments per net to minimize congestion for future nets

  Use left-edge criterion on each dog-leg segment to fill up the channel

Multipoint Nets

- Multipoint nets are harder to design than two-point nets

  Rectilinear Steiner tree problem:

  Find the minimum-length tree interconnecting all net’s points

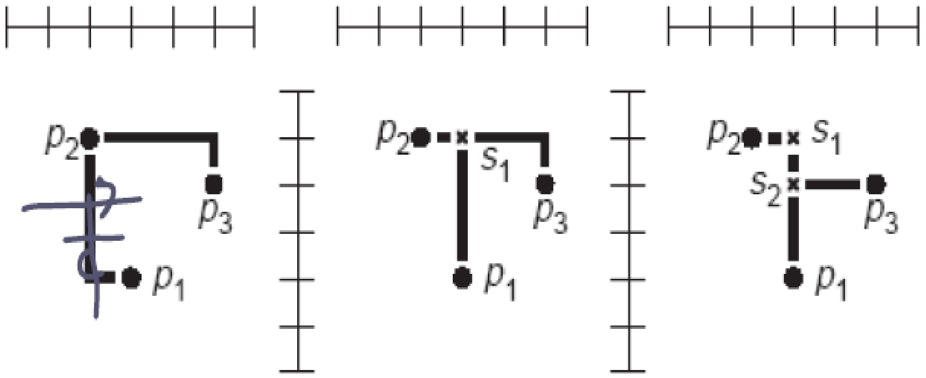
  Find additional points (so-called Steiner points) in the plane, if they contribute to a shorter tree length

- Steiner tree algorithm

  Compute the spanning tree

  Optimize the tree length by flipping L-shaped branches

  Steiner points always have degree three



Layout for Low Power

- Place and route to minimize

  Capacitance of nodes with high glitching activity

- Feed back wiring capacitance values

  To better estimate power consumption

- Size wires to be able to handle current

  Requires designing topology of V_{DD}/V_{SS} networks

- Keep power network in metal

  Requires designing planar wiring

Clock Delay

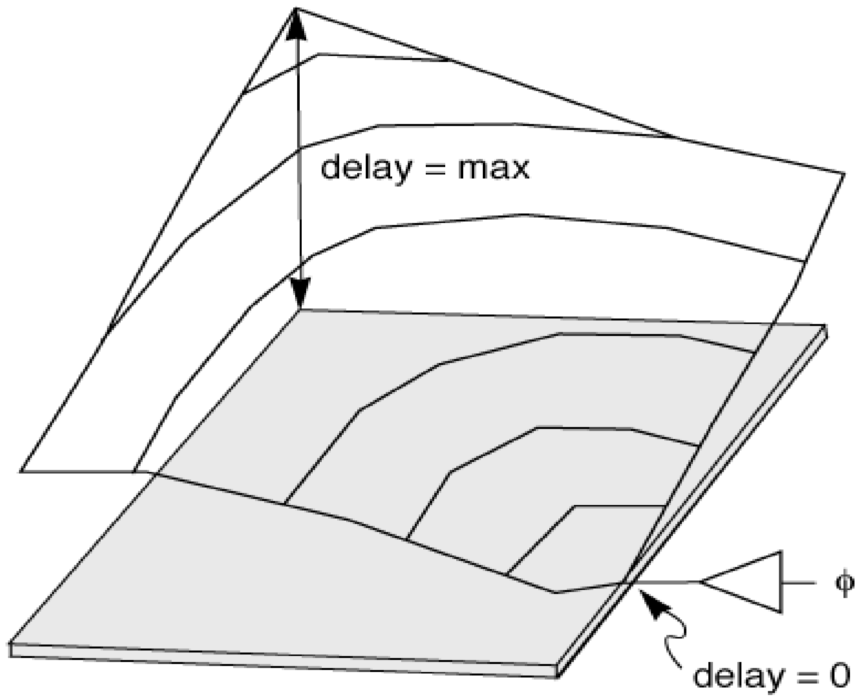
- Clock delay varies with position

  Deliver clock to memory elements with acceptable skew

  Deliver clock edges with acceptable sharpness

- Clocking network design

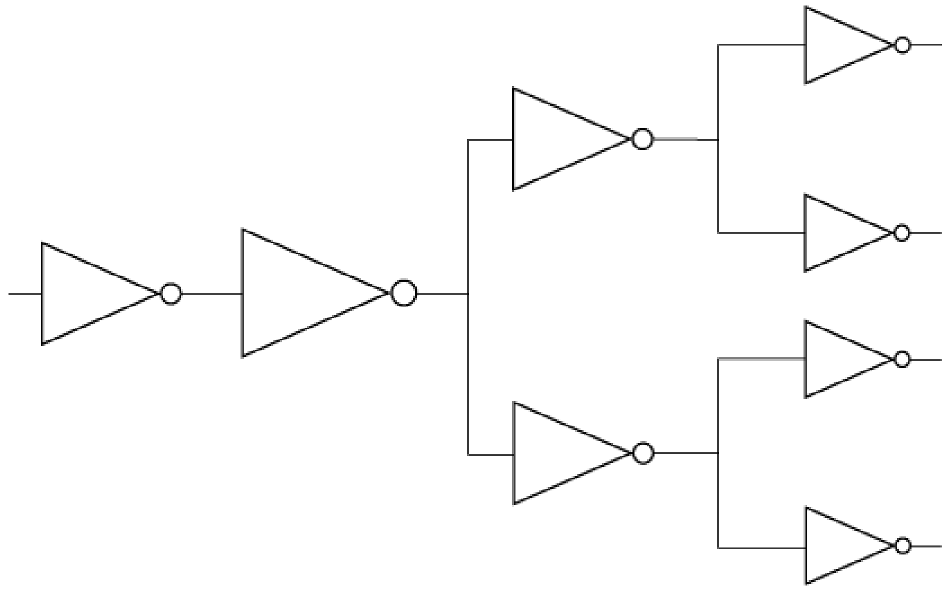
  The greatest challenge in the design of a large chip



Clock Distribution Tree

- Clocks are generally distributed via wiring trees
- Use low-resistance interconnect to minimize delay
- Use multiple drivers to distribute driver requirements

  Use optimal sizing principles to design buffers
- Clock lines can create significant crosstalk



Design for Manufacturability

- Defects and Faults
- Critical Area Modeling
- Critical Area Extraction
- Yield Modeling
- Yield Control

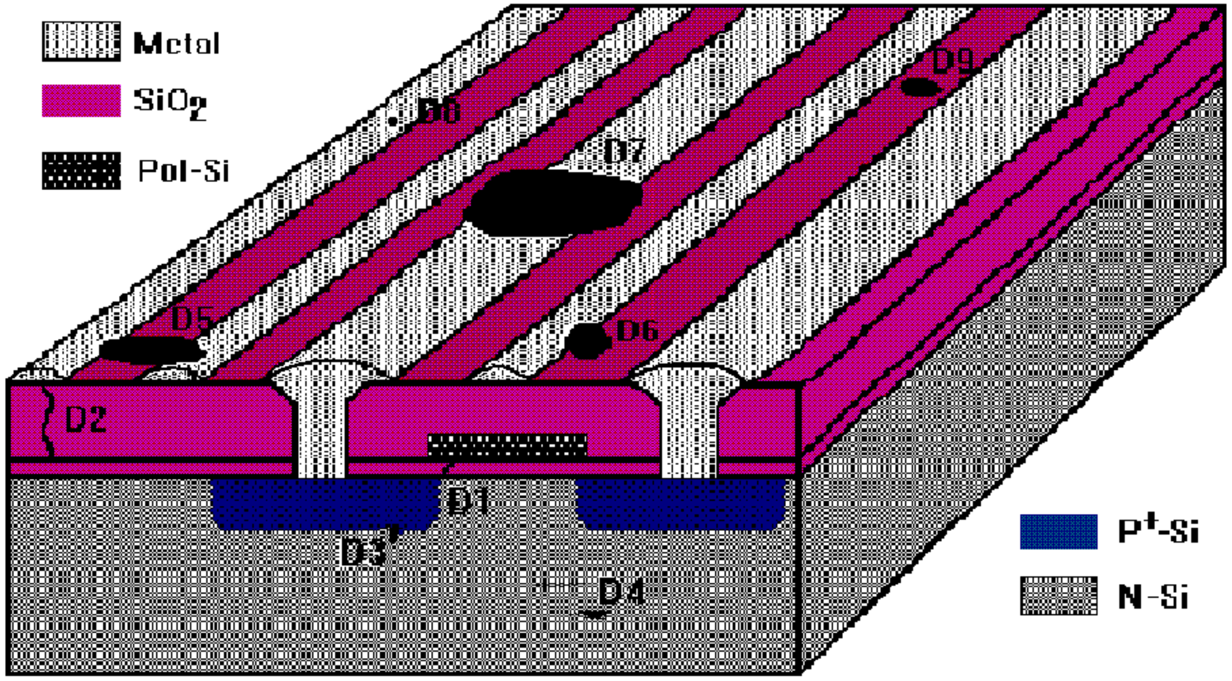
Defects, Faults, and Tests

- Design faults
- Random faults
  - Shorts between lines
  - Breaks of lines
  - Leakage of insulation layers
- Permanent faults
- Dynamic faults
- Transient faults

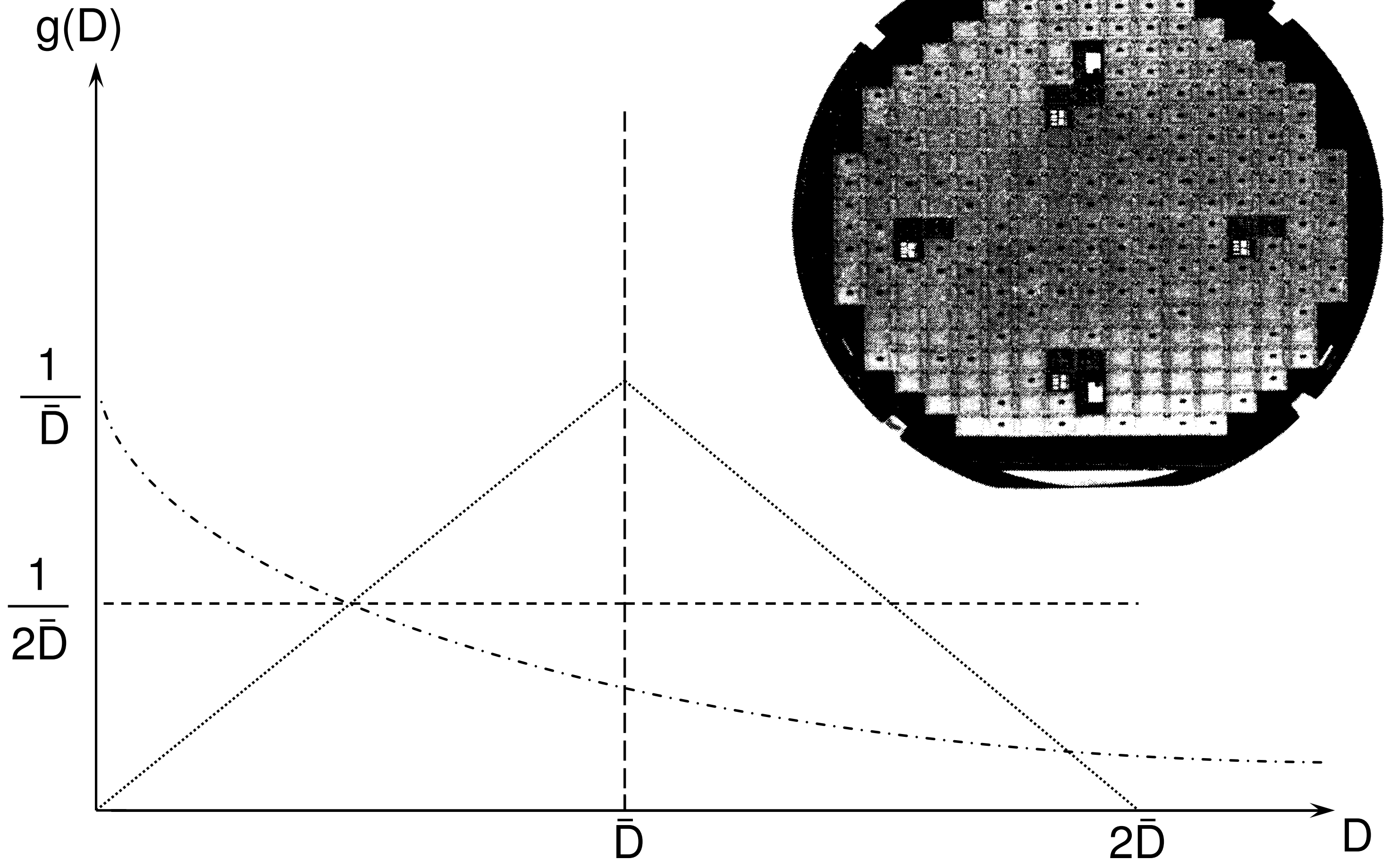
- Parametric test
  - Power consumption
  - Input resistance
  - Input/output current
- Functional test
  - Test signals applied
  - Output observed

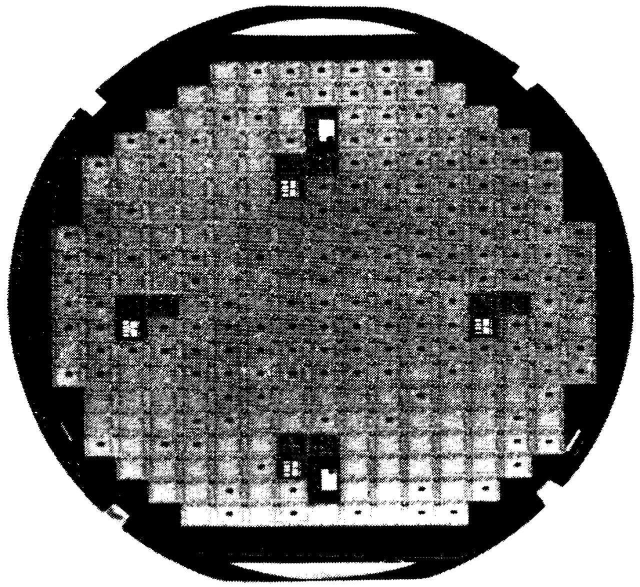
Defects Statistics

- Defect density distribution g(D)
- Defect size distribution h(X)

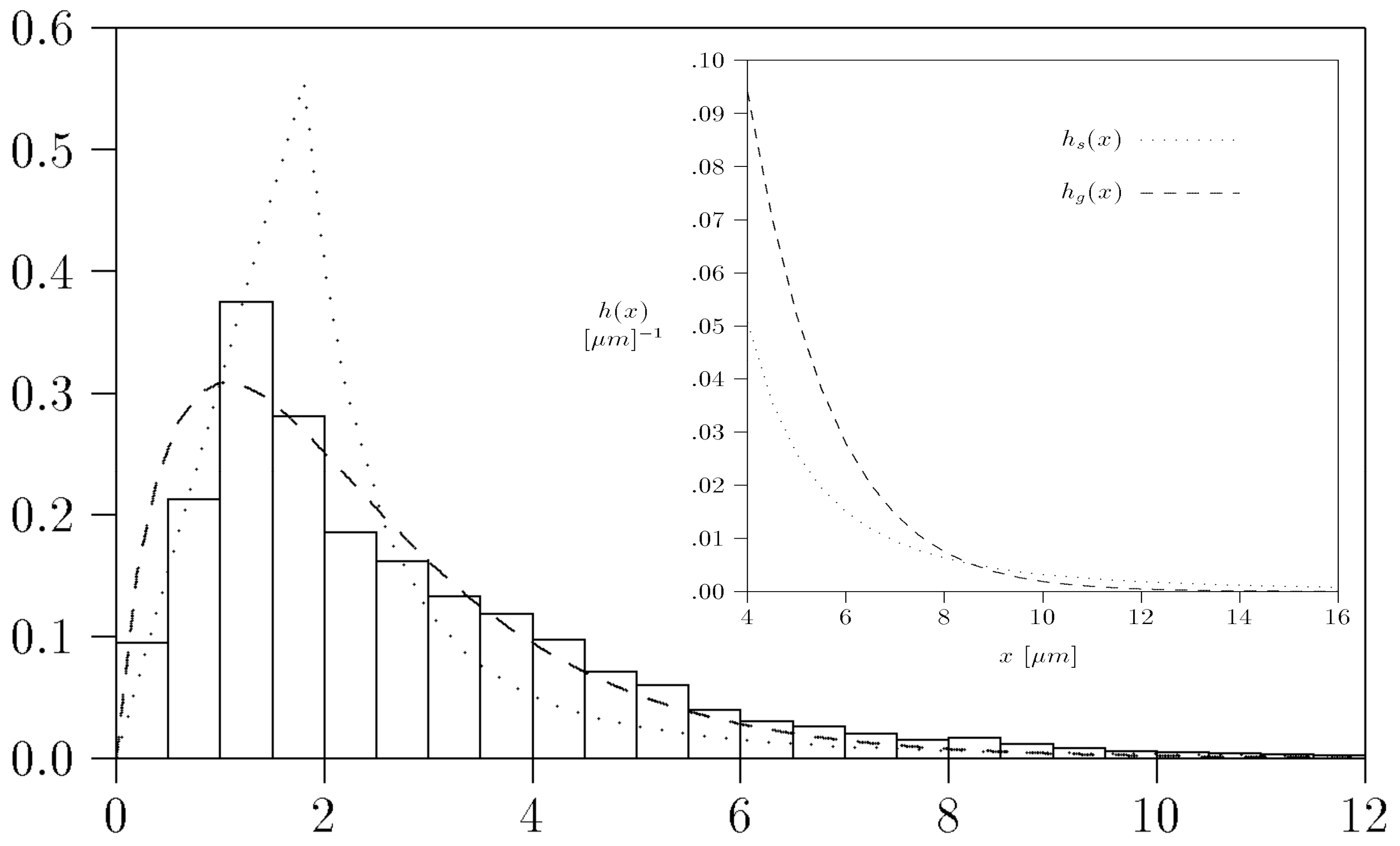


Defect Density Distribution



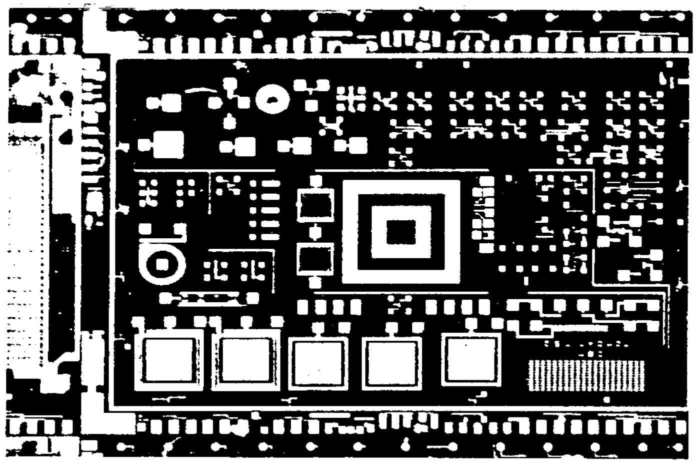


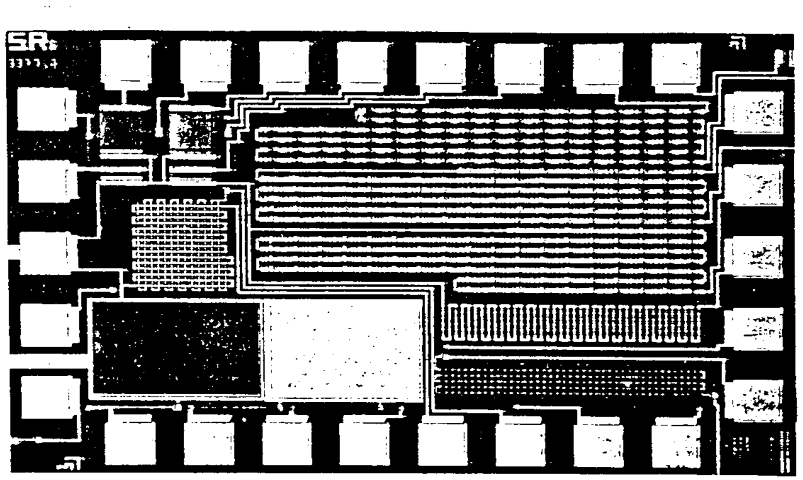
Defect Size Distribution



Test Structures

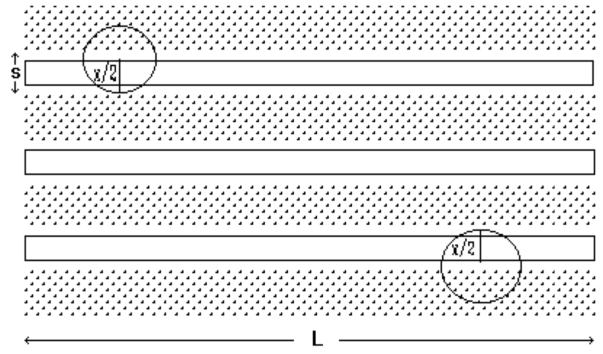
- Sample size
  - Number of measurements
- Structure size
  - High and low defect densities
- Critical dimensions
  - As defect sizes
- Self-isolation
  - Sensitive on one defect type
- Measurability
  - Electrical
  - Simple
  - Reliable



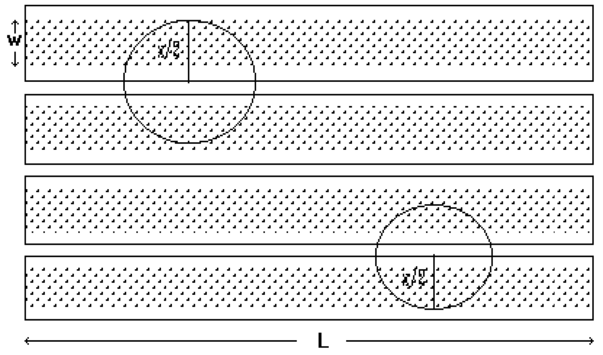


Critical Area Models

Long parallel conductors

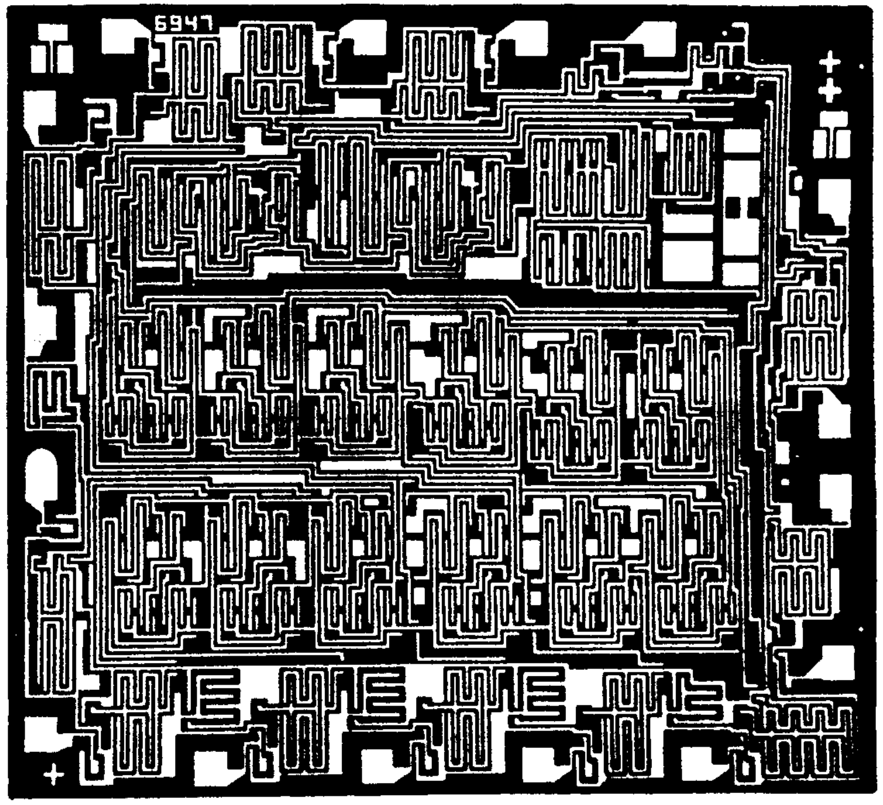


(a)



(b)

Real Patterns



Critical Area Models

$$\overline{A} = \int_0^\infty A(x)h(x)dx$$

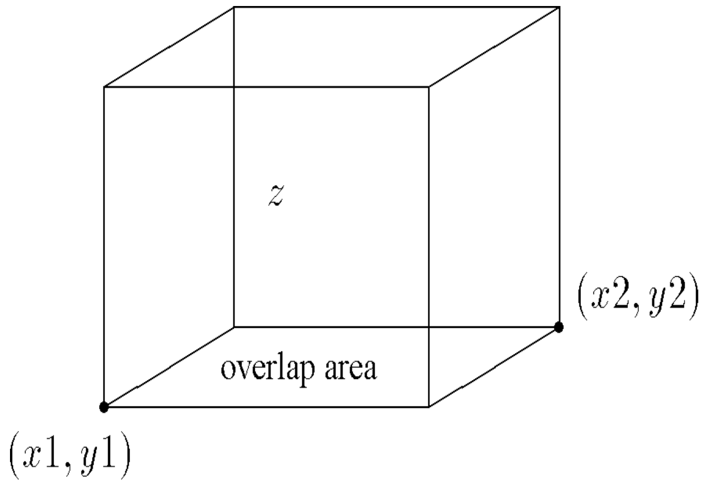
$$h(x) = \frac{x^{\alpha-1}\exp(-x/\beta)}{\Gamma(\alpha)\beta^{\alpha}}$$

$$A_s(x) = \begin{cases} 0 & \text{for} \quad 0 \le x \le s \\ NL(x-s) & \text{for} \quad s \le x \le 2s+w \\ NL(w+s) & \text{for} \quad 2s+w \le x \end{cases}$$

$$A_o(x) = \begin{cases} 0 & \text{for} \quad 0 \le x \le w \\ NL(x-w) & \text{for} \quad w \le x \le 2w+s \\ NL(s+w) & \text{for} \quad 2w+s \le x \end{cases}$$

Critical Area Extraction

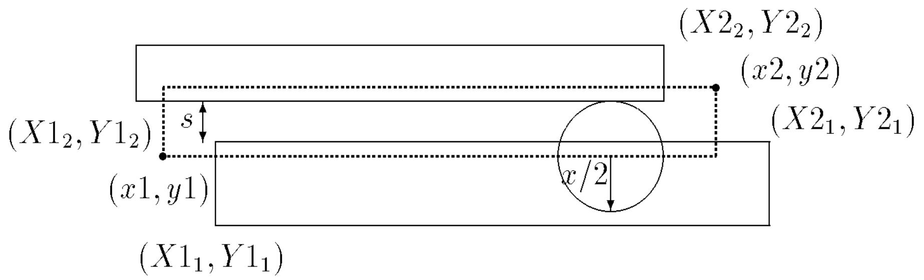
Point defects

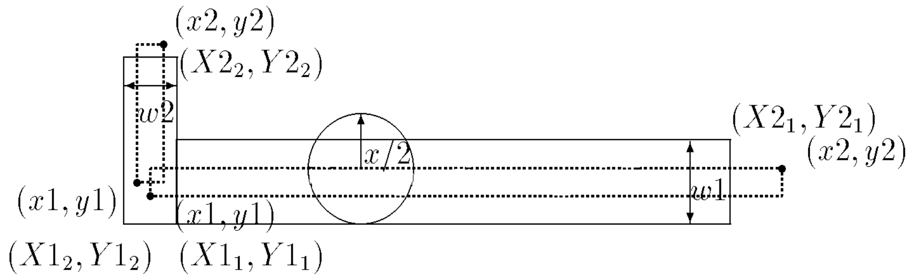


$$A_l = (x2 - x1)(y2 - y1)$$

$$A_v = 2z[(x2 - x1) + (y2 - y1)]$$

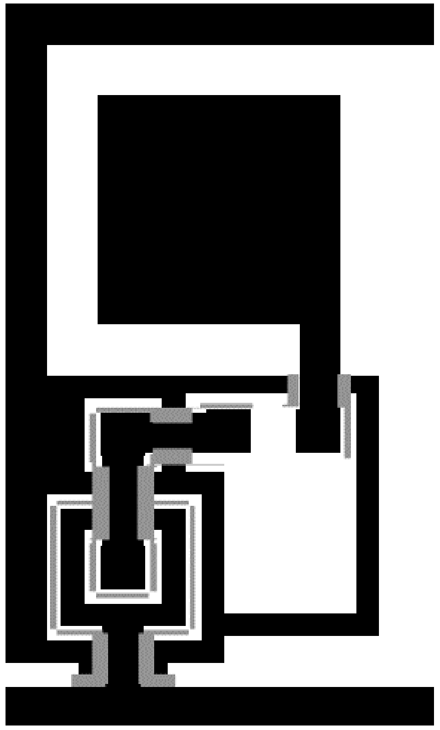
Lithographic defects



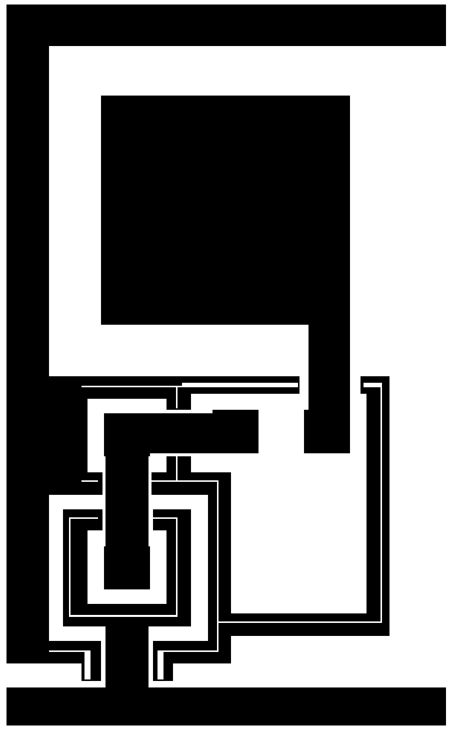


Critical Area Extraction

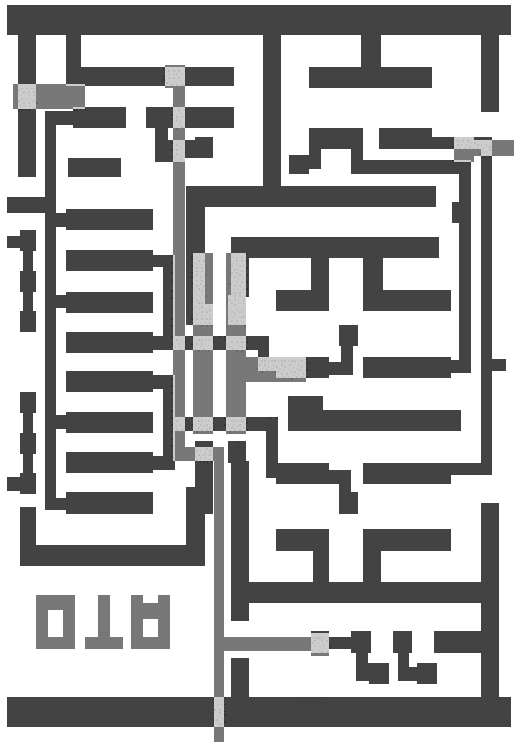
Shorts



Breaks

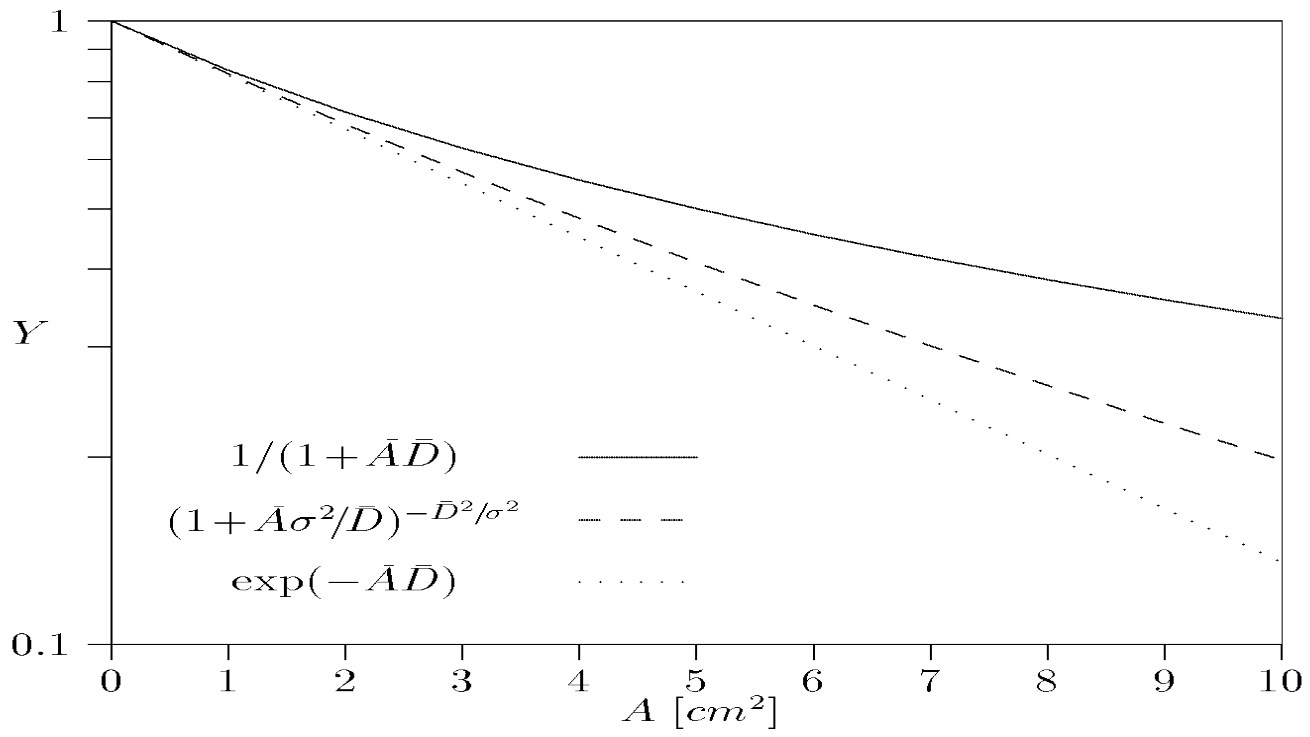


Vertical shorts

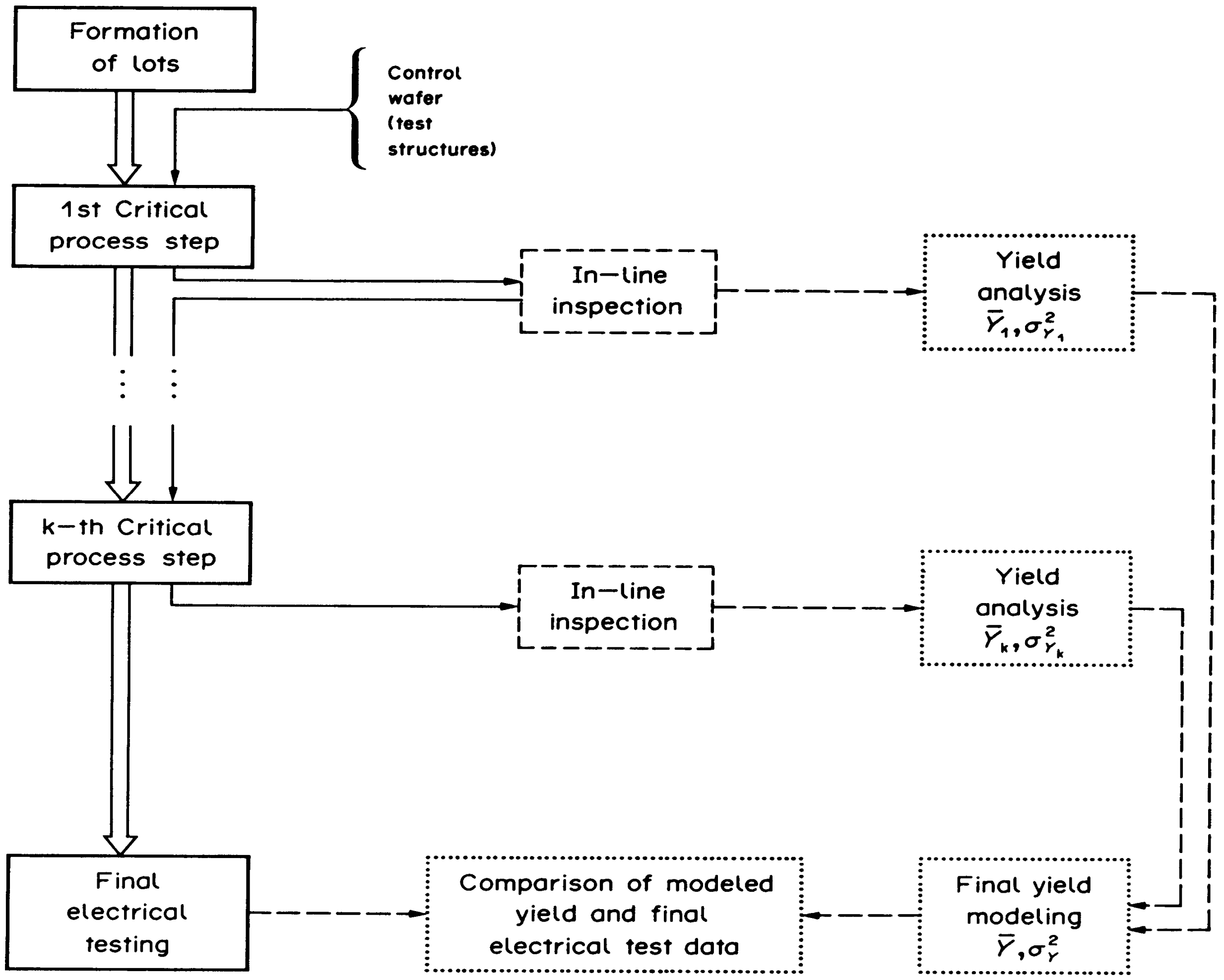


Yield Models

$$Y = \int_0^\infty \exp\left(-\bar{A}D\right) g(D)\, dD$$



Yield Control



Formation of lots
Control wafer (test structures)
1st Critical process step
In–line inspection
Yield analysis $\bar{Y}_1, \sigma^2_{Y_1}$
k–th Critical process step
In–line inspection
Yield analysis $\bar{Y}_k, \sigma^2_{Y_k}$
Final electrical testing
Comparison of modeled yield and final electrical test data
Final yield modeling $\bar{Y}, \sigma^2_Y$