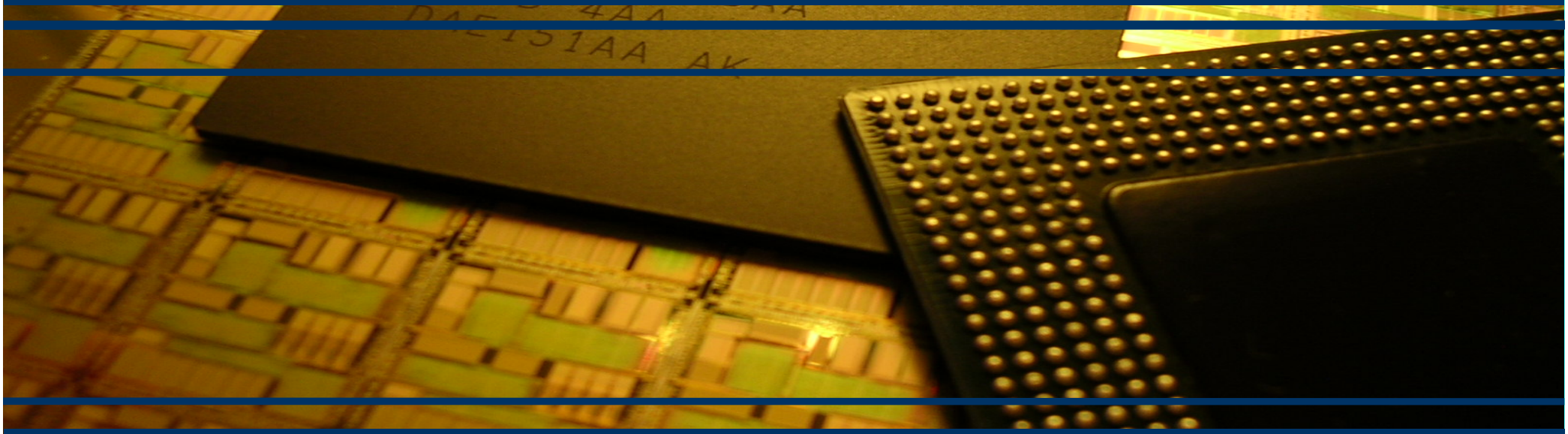
Application Specific Integrated Circuits
Design and Implementation



Dott. Ing. Maurizio Skerlj
mailto: maurizio.skerlj@ieee.org

AGENDA

- **Efficient RTL;**
- **Synthesis;**
- **Static Timing Analysis;**
- **Design for Testability and Fault Coverage.**

References

M. J. S. Smith: *Application-Specific Integrated Circuits*, **Addison-Wesley (1997).**

K. K. Parhi: *VLSI Digital Signal Processing Systems,* **John Wiley & Sons, Inc. (1999).**

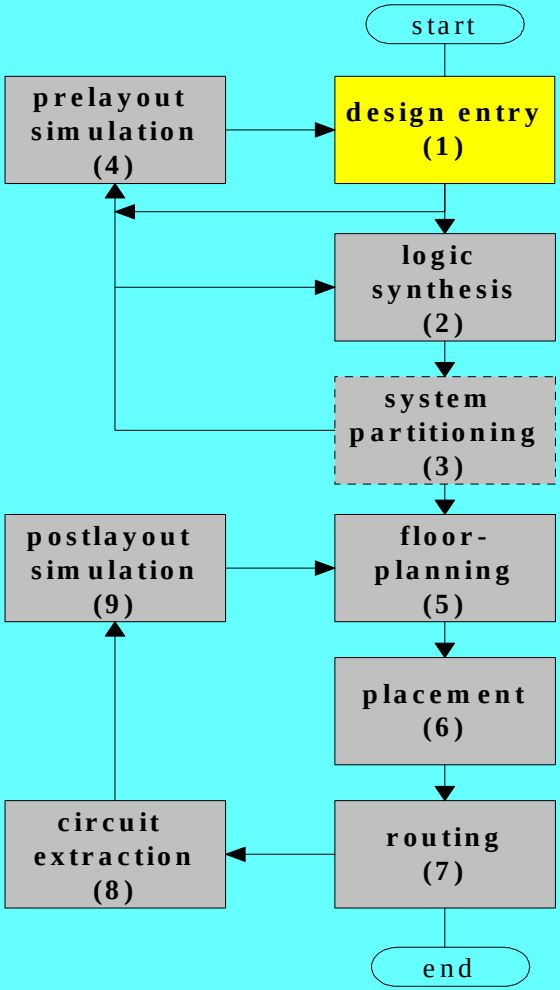
R. Airiau, J. Berge, V. Olive: *Circuit Synthesis with VHDL*, **Kluwer Academic Publishers (1991).**

D. L. Perry: *VHDL (Computer Hardware Description Language)*, **McGraw Hill (1991).**

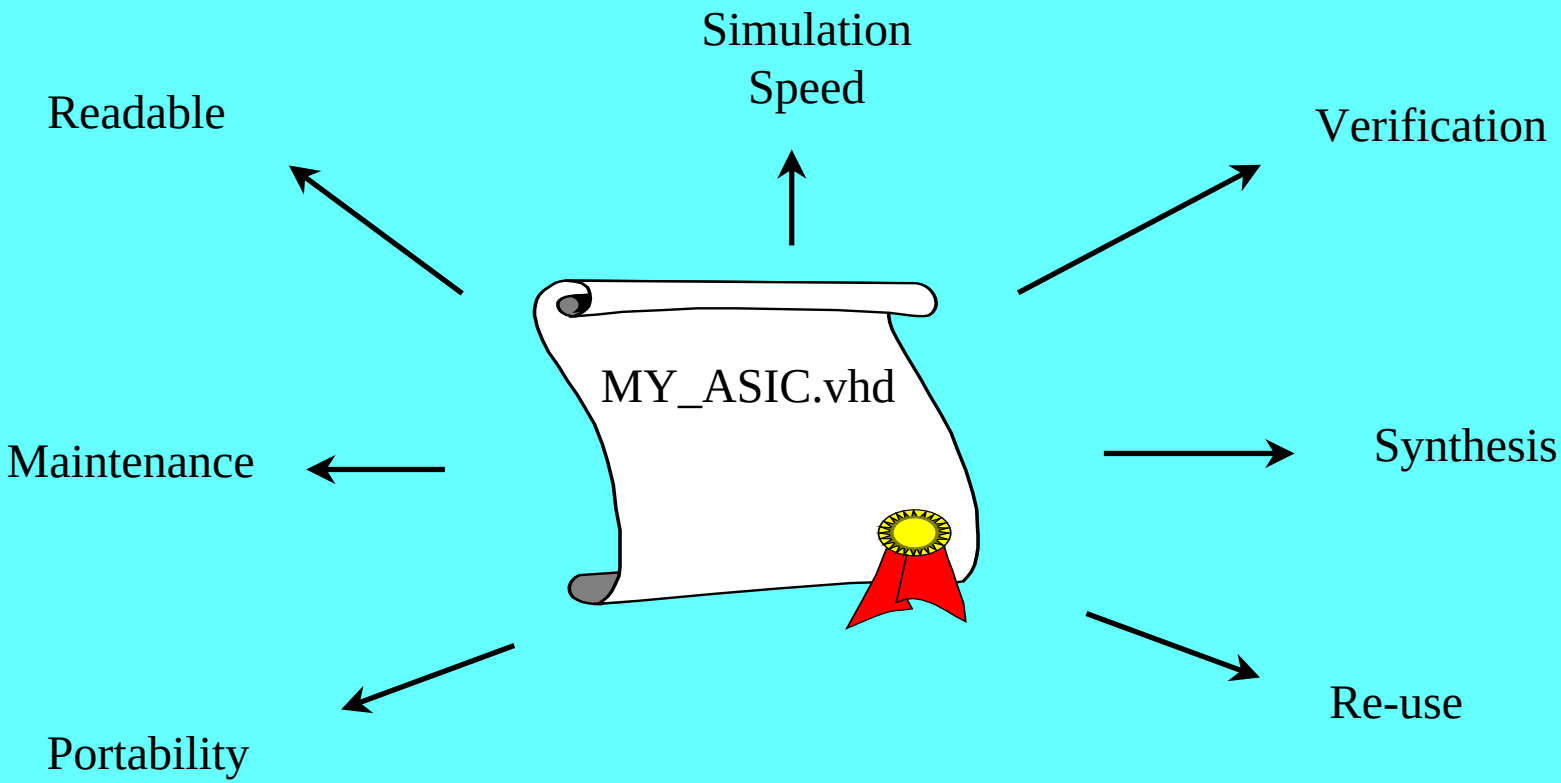
P. Kurup, T. Abbasi: *Logic Synthesis Using Synopsis*, **Kluwer Academic Publishers (1997).**

K. Rabia: *HDL coding tips for multi-million gate ICs,* **EEdesign (December 2002).**

Design Entry



The Coding Style



Readable VHDL

- use consistent style for signal names, variables, functions, processes, etc. (e.g. DT_name_77, CK_125, START_L, I_instance_name, P_process_name,...)
- use the same name or similar names for ports and signal that are connected to.
- use a consistent ordering of bits (e.g. MSB downto LSB).
- use indentation.
- use comments.
- use functions instead of repeating same sections of code.
- use loops and arrays.
- don’t mix component instantiation and RTL code.

Simulation speed

It takes days to simulate few milliseconds of circuit real life!

Therefore it is very important to write HDL code that doesn’t slow down the verification process.

- use arrays as much as possible instead of loops.
- priority on low frequency control signals.
- avoid process with heavy sensitivity lists (each signal in the sensitivity list will trig the process).

Verification

The VHDL and the consequent inferred circuit architecture must be thought for a exhaustive verification.

•Avoid architectures for which is not clear what is the worst case or will create difficult-to-predict problems (e.g. asynchronous clocking and latches).

•Poor practices on clock generations (gated clocks, using both falling and rising clock edges in the design, etc.)

•Never use clocks where generated.

•Always double-check your design with a logic synthesis tool as early as possible. (VHDL compilers don’t check the sensitivity lists and don’t warn you about latches)

The VHDL - RTL Subset

NO

- Timing delays
- Multidimensional arrays (latest l.s. tools allows it)
- Implicit finite state machines

YES

- Combinatorial circuits
- Registers

Recommended Types

YES

- `std_ulogic` for signals;
- `std_ulogic_vector` for buses;
- `unsigned` for buses used in circuits implementing arithmetic functions.

NO

- `bit` and `bit_vector`: some simulators don't provide built-in arithmetic functions for these types and, however, is only a two states signal ('X' state is not foreseen);
- `std_logic(_vector)`: multiple drivers will be resolved for simulation (lack of precise synthesis semantics).

Design re-use

Nowadays, designs costs too much to use them for only one project. Every design or larger building block must be thought of as **intellectual property (IP).**

Reuse means:
- use of the design with multiple purposes;
- design used by other designers;
- design implemented in other technologies;

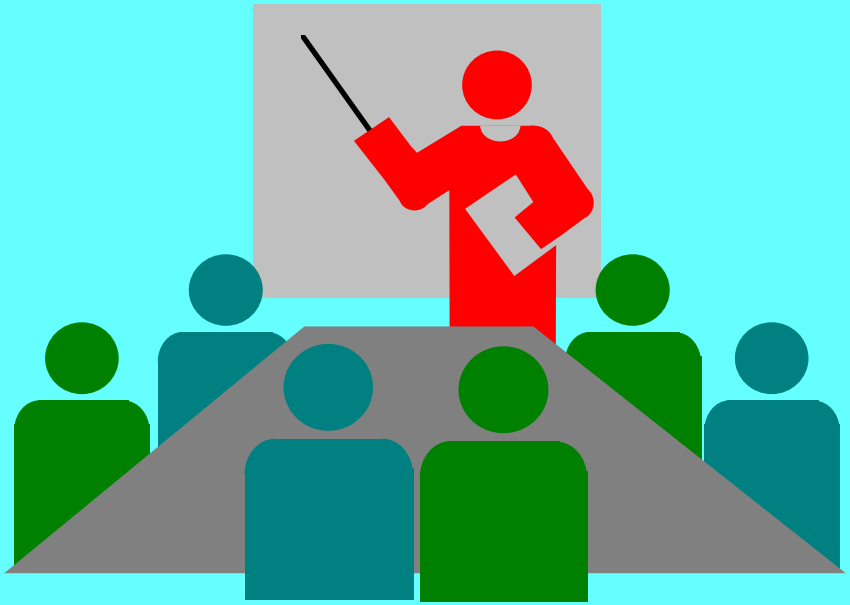
Therefore, it is necessary to have strong coding style rules, coded best practices, architectural rules and templates.

Maintenance

Design that are implemented following rules and coding styles shared by the design community are easy to understand and to upgrade, prolonging its life cycle.

For the same purposes a **good documentation** is a must. On the other hand, the documentation itself should be shorter, dealing only with the general description of the block, since most of the details will be clear from the design practices and guidelines.

Some HDL guidelines and examples



Combinatorial Processes

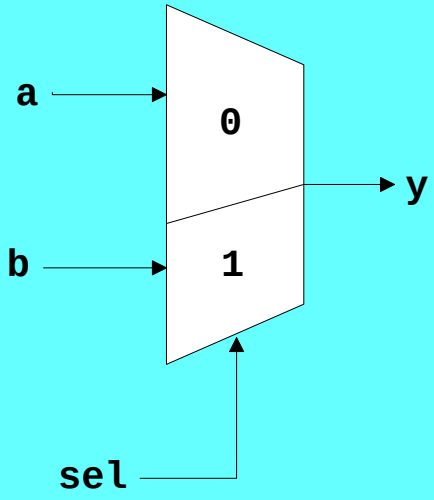

```
process(sensitivity list)
   begin
      statement_1;
         …
      statement_n;
end process;
```

! Only signals in the sensitivity list activate the process. If the list is not complete, the simulation will show poor results;

! Not assigning signals in every branch of the concurrent statements will lead to inferred latches.

Concurrent Assignments Inside Processes


```
P_MUX1: process(sel,a,b)
begin
    case sel is
        when '0' =>
            y <= a;
        when others =>
            y < b;
      end case;
end process;

P_MUX2: process(sel,a,b)
begin
    if SEL = '0' then
        y <= a;
    else
        y <= b;
    end if;
 end process;
```



Tips on Conditional Statements

EASY TO WRITE, DIFFICULT TO VERIFY AND MAINTAIN:

```
if cond1 then
    …
elsif
    …
else
    …
end if;
```

DIFFICULT TO WRITE, EASY TO VERIFY AND MAINTAIN:

```
case sel is
    when choice_1 => …
    when choice_2 => …
    when others => …
end if;
```

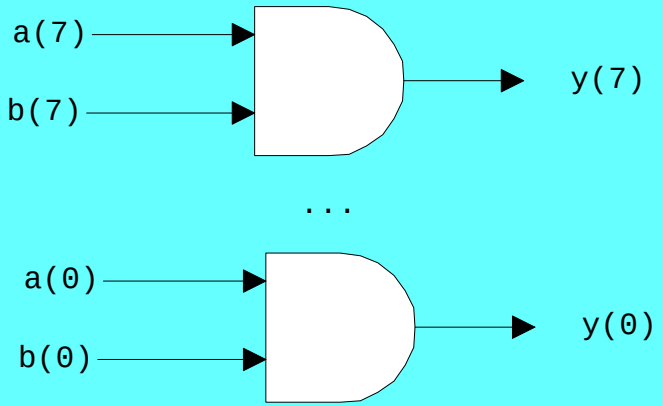
Frequent Errors


```
if a="00" then
    y0 <= '0';
elsif a="11" then
    y0 <= '1';
    y1 <= '0';
else
    y0 <= '0';
    y1 <= '1';
end if;
```

y1 not always assigned => **INFERRED LATCH**

The Use of ‘for loops’


```
signal a,b,y: std_ulogic_vector(7 downto 0);

for I in y’range loop
    y(I)<= a(I) and b(I);
end loop;
```



Tips for ‘for loop’ Implementation

The `for loop` statement is supported by synthesis tools when the range bounds in the loop are globally static. When the range is not static (e.g. when one of the bounds is a signal value), the synthesis result is not a simple hardware duplication.

In other words, the `for loop` must be un-foldable.

! Often the use of `for loop` can be avoided using vectors.

Avoiding for loops

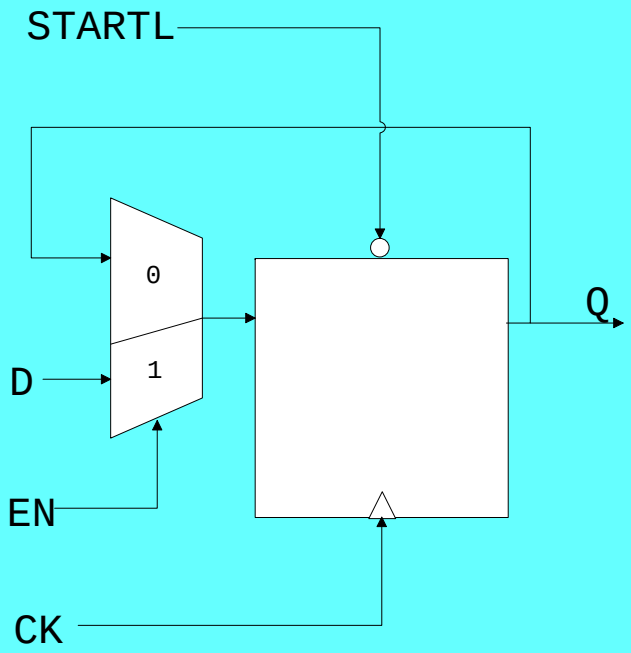

```
signal a: std_ulogic_vector(15 downto 0);


P_SHIFT: process(a)
begin
    for I in 0 to 14 loop
        a(I) <= a(I+1);
    end loop;
    a(15) <= a(0);
end process;
```

OR

```
a(14 downto 0) <= a(15 downto 1);
a(15) <= a(0);
```

The Edge Triggered Enabled flip flop


```
process(CK,STARTL)
  begin
    if STARTL = '0' then
      Q <= '0';
    elsif CK='1' and CK'event then
      if EN = '1' then
          Q <= D;
        end if;
    end if;
end process;
```



Finite State Machines

Only synchronous finite state machines are the only ones accepted by synthesis tools.

Basics:

• The automaton is always in one of its possible sates: the **current state** (stored in the state register).

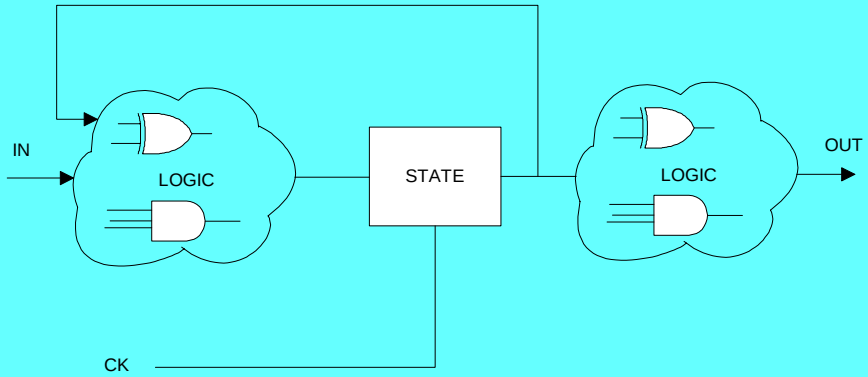
• The next state may be computed using the current state and the input values.

• Output values are computed depending on either the current state or the transition between two states **(Moore** or **Mealy**).

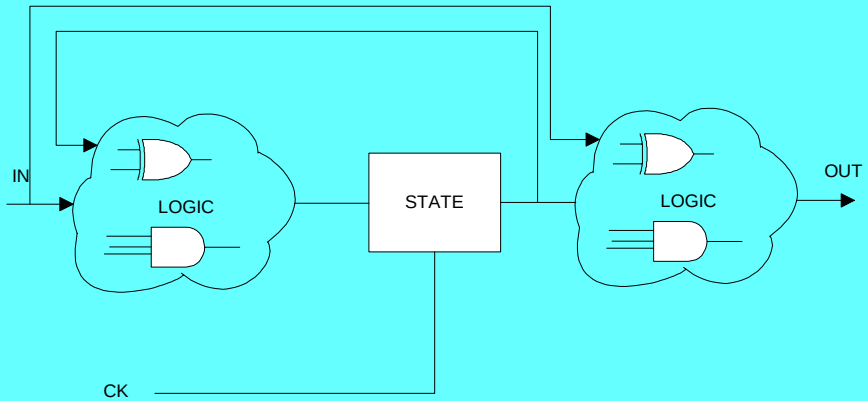
• During each clock period, the state register is updated with the previously computed state (next state)

Moore and Mealy

Moore



Mealy



FSM template


```
P_NXT_STATE: process(STATE, IN)
begin
    STATE_NXT <= … logic …
end process;

P_STORE: process(CK, STARTL)
begin
    if STARTL = '0' then
        STATE <= (others => '0');
    elsif CK'event and CK='1' then
      STATE <= STATE_NXT;
    end if;
end process;

P_OUT: process(STATE, IN)
begin
    OUT <= logic(STATE, IN)
end process;
```

PRESENT ONLY
IF *MEALY*

One-hot vs. binary coded

One-hot encoding sets one bit in the state register for each state. This seems wasteful (a FSM with N states requires exactly N flip-flops instead of $\lceil log_2 N \rceil$ with binary encoding).

One-hot encoding simplifies the logic and the interconnect between the logic resulting often in smaller and faster FSMs.

Especially in FPGAs, where routing resources are limited, one-hot encoding is sometimes the best choice.

The Meta-stability Phenomenon

Meta-stability may occur when the data input changes too close to the clock edges (setup or hold violation). In such cases the flip-flop cannot decide whether its output should be a ‘1’ or a ‘0’ for a long time. This situation is called an **upset**.

This cannot occur in fully synchronous design if timing constraints were met. However it may rise the opportunity to register signals that come from the outside world (or form another clock domain).

Metastability Theory

Experimentally was found that the **probability of upset** is:

$$p = T_0 e^{\frac{-t_r}{\tau_c}}$$

where t_r is the time the flip-flop has to resolve the output, T_0 and τ_c are flip-flop constant.
The **M**ean **T**ime **B**etween **U**psets **(MTBU)** is

$$MTBU = \frac{1}{p \cdot f_{CK} \cdot f_{DT}}$$

Therefore even if the data is changing slowly, simple over-sampling is not an error-free technique.

Input Synchronisation

Using two flip-flops in cascade **greatly reduces the overall value of τ_c and T_0** and as a consequence the probability of upset, *p*.

When the first flip-flop capture an intermediate voltage level (‘X’) the flip-flop takes some time to resolve in a ‘0’ or ‘1’ level. The resolution time is usually **several times** longer than the **clock-to-out** time of the flip-flop, but less than the clock period. However the second flip-flop is always capturing stable data.

The penalty is an **extra clock cycle latency**.

Pipelining and Parallel Processing

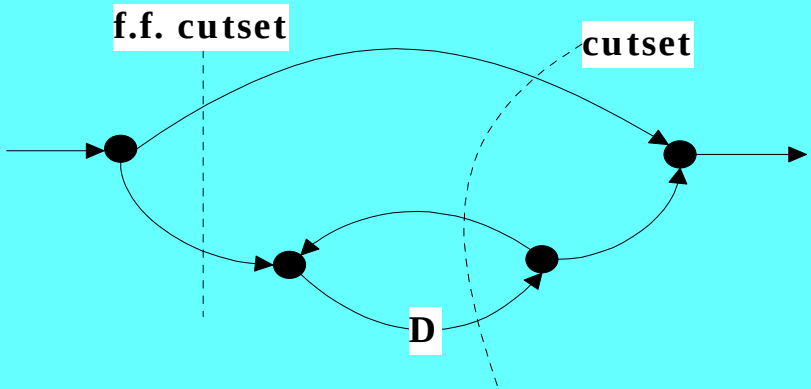
Pipelining transformation leads to a **reduction in the critical path**, which can be exploited to either **increase the sample speed** or to **reduce power consumption** at same speed.
Pipelining reduces the effective critical path by **introducing pipelining delays** along the datapath.

In **parallel processing**, multiple outputs are computed in a clock period. Parallel processing increase the sampling rate by **replicating the hardware** so that several inputs can be processed a the same time.

Therefore, the effective **sampling speed is increased** by the level of parallelism.

Pipelining

The pipelining latches can only be placed across any **feed-forward cutset** of the graph. We can arbitrarily place latches on a feed-forward cutset without affecting the functionality of the algorithm.



In an ***M*-level** pipelined system, the number of delay elements in any path from input to output is ***(M-1)*** greater than that in the same path in the original system.

The two main drawbacks of the pipelining are increase in the number of latches (**area**) and in **system latency**.

Parallel Processing

To obtain a parallel processing system, the **SISO** (single input – single output) system must be converted into a **MIMO** (multiple input – multiple output).

In a parallelized system the critical path remain the same. It is important to understand that in a parallel system the clock period T_{ck} and the sample period T_S are different. In an M-level parallelized system holds

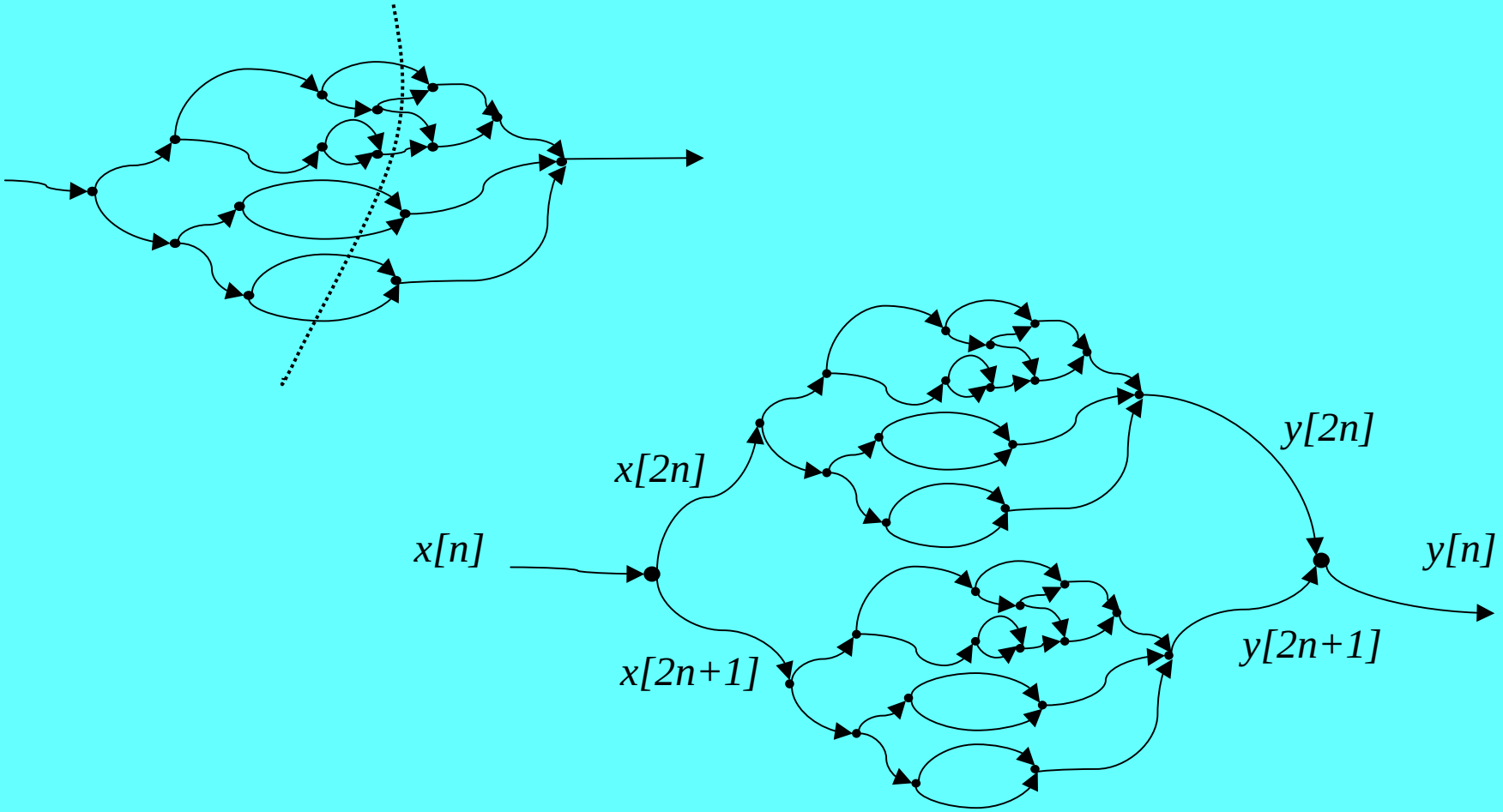
$$T_{ck}=M\,T_S$$

Pipelining and Parallel Processing
Dualism (1)

Parallel processing and pipelining techniques are **duals** of each other. **If a computation can be pipelined, it can also be processed in parallel and vice versa**.

While independent sets of computation are performed in an interleaved manner in a pipelined system, they are computed in parallel processing mode by means of duplicate hardware.

Pipelining and Parallel Processing
Dualism (2)



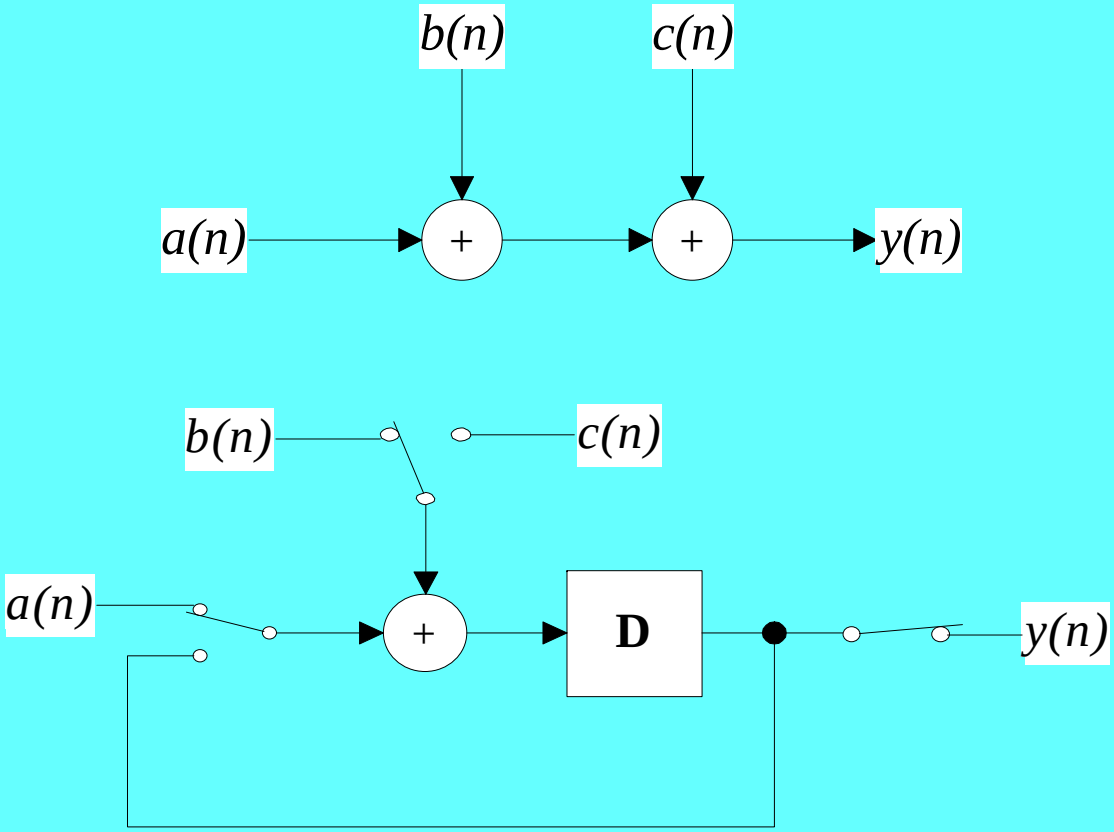
Folding

It is important to minimize the silicon area of the integrated circuits, which is achieved by reducing the number of functional units, registers, multiplexors, and interconnection wires.

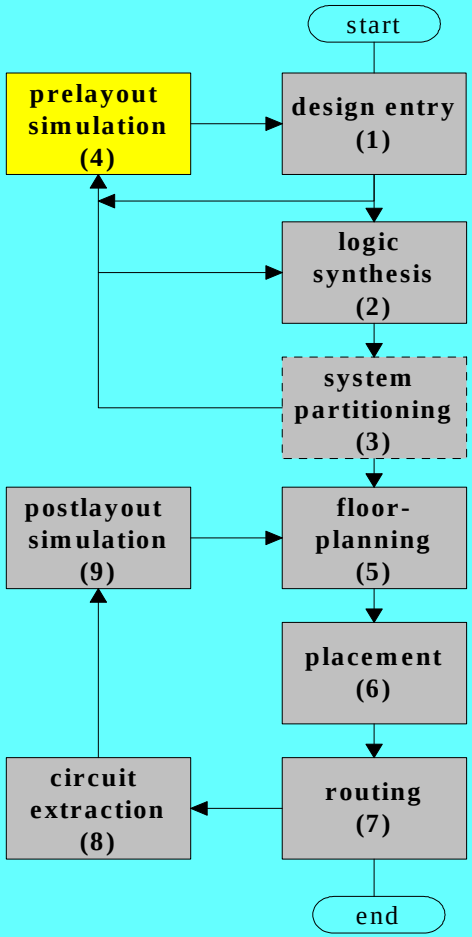
By executing multiple algorithm operations on a single functional unit, the number of functional units in the implementation is reduced, resulting in a smaller silicon area.

Folding provides a means for trading area for time in a DSP architecture. In general, folding can be used to reduce the number of hardware functional units by a factor N at the expense of increasing the computation time by a factor of N.

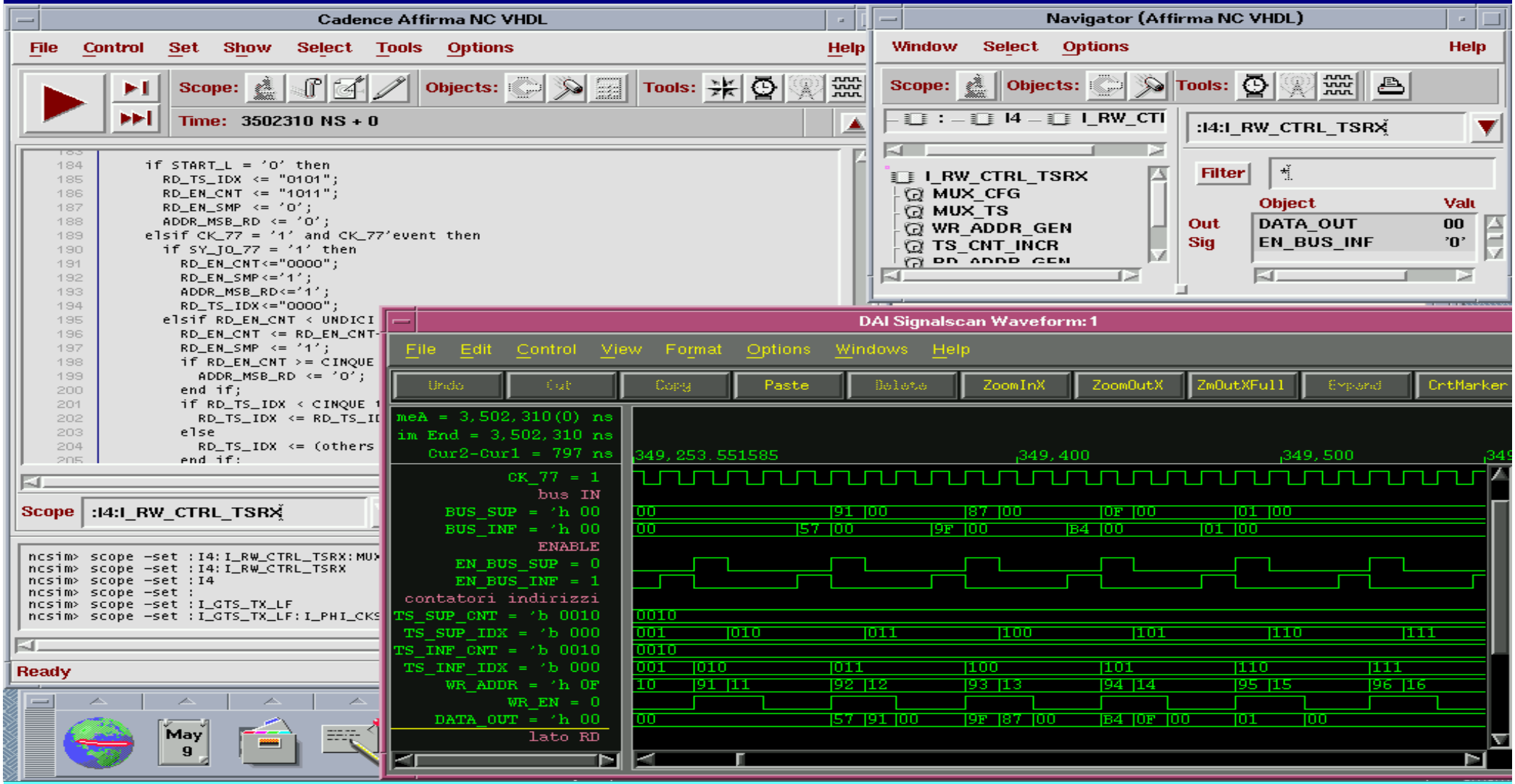
Folding Example



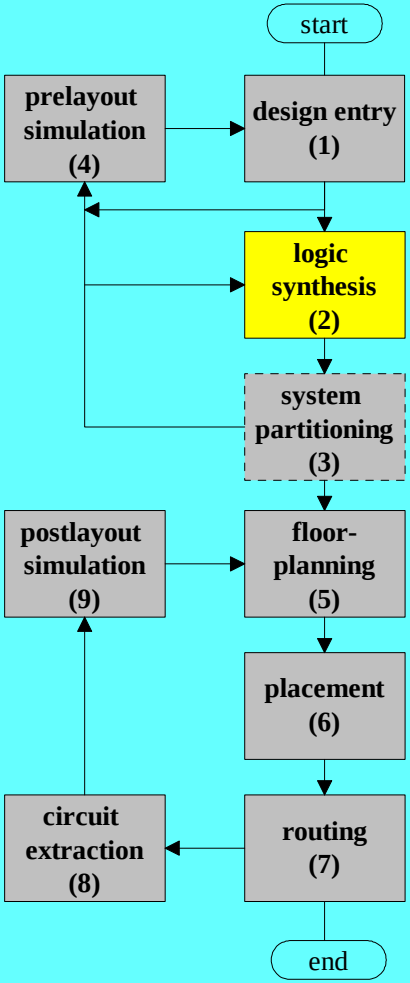
Functional Verification



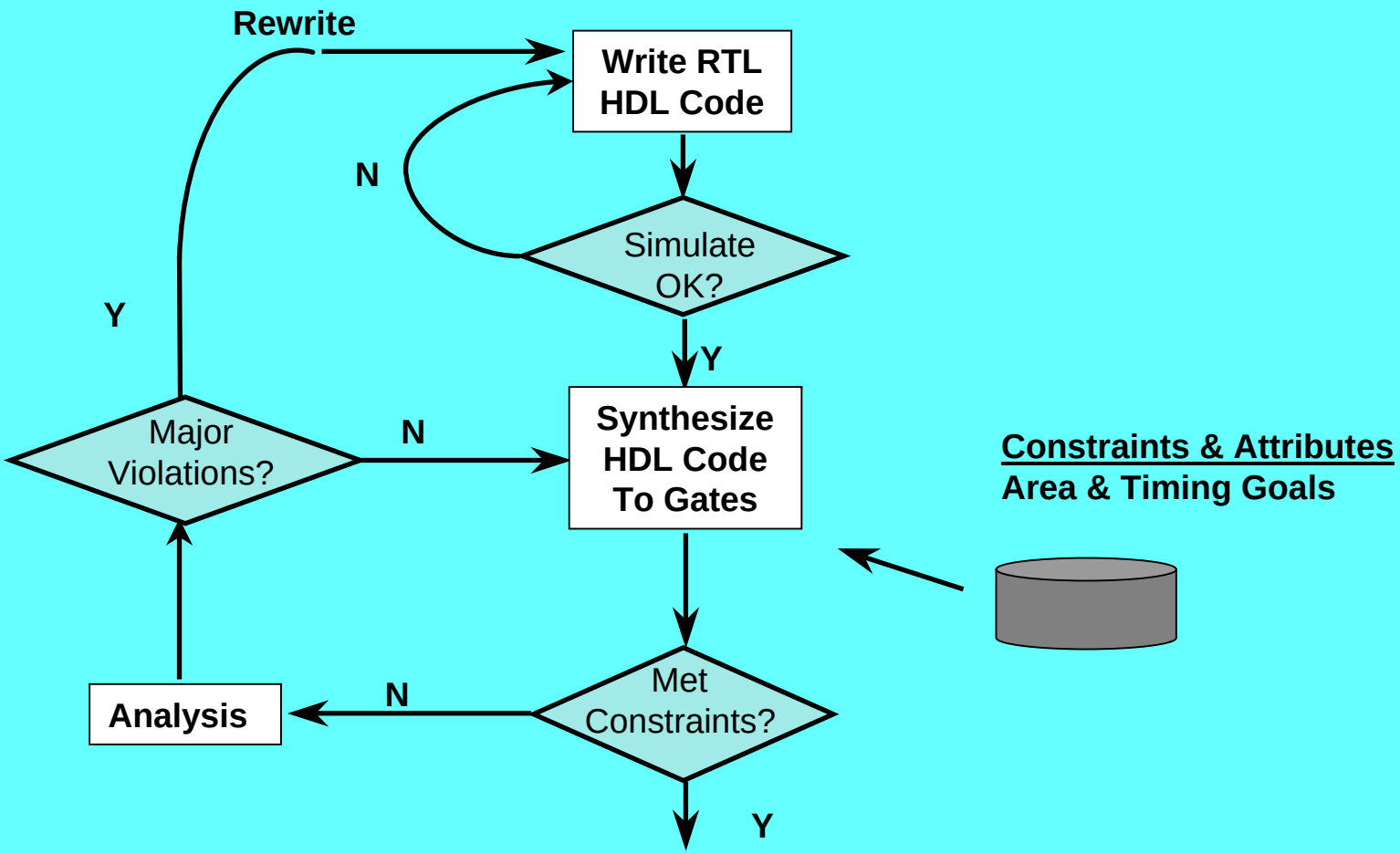
Pre- and Post Layout Simulation



Logic Synthesis



RTL Block Synthesis

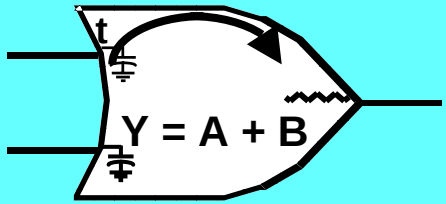


Technology Library


```
cell ( AND2_3 ) {
    area : 8.000 ;
    pin ( Y ) {
        direction : output;
        timing ( ) {
            related_pin : "A" ;
            timing_sense : positive_unate ;
            rise_propagation (drive_3_table_1) {
                 values ("0.2616, 0.2608, 0.2831,..)
            }
            rise_transition (drive_3_table_2) {
              values ("0.0223, 0.0254, ...)
                    . . . .
          }
        function : "(A & B)";
        max_capacitance :  1.14810 ;
        min_capacitance :  0.00220 ;
    }
    pin ( A ) {
        direction : input;
        capacitance : 0.012000;
        }
                    . . . .
```

Cell Name

Cell Area



Nominal Delays

Cell Functionality

Design Rules for Output Pin

Electrical Characteristics of Input Pins

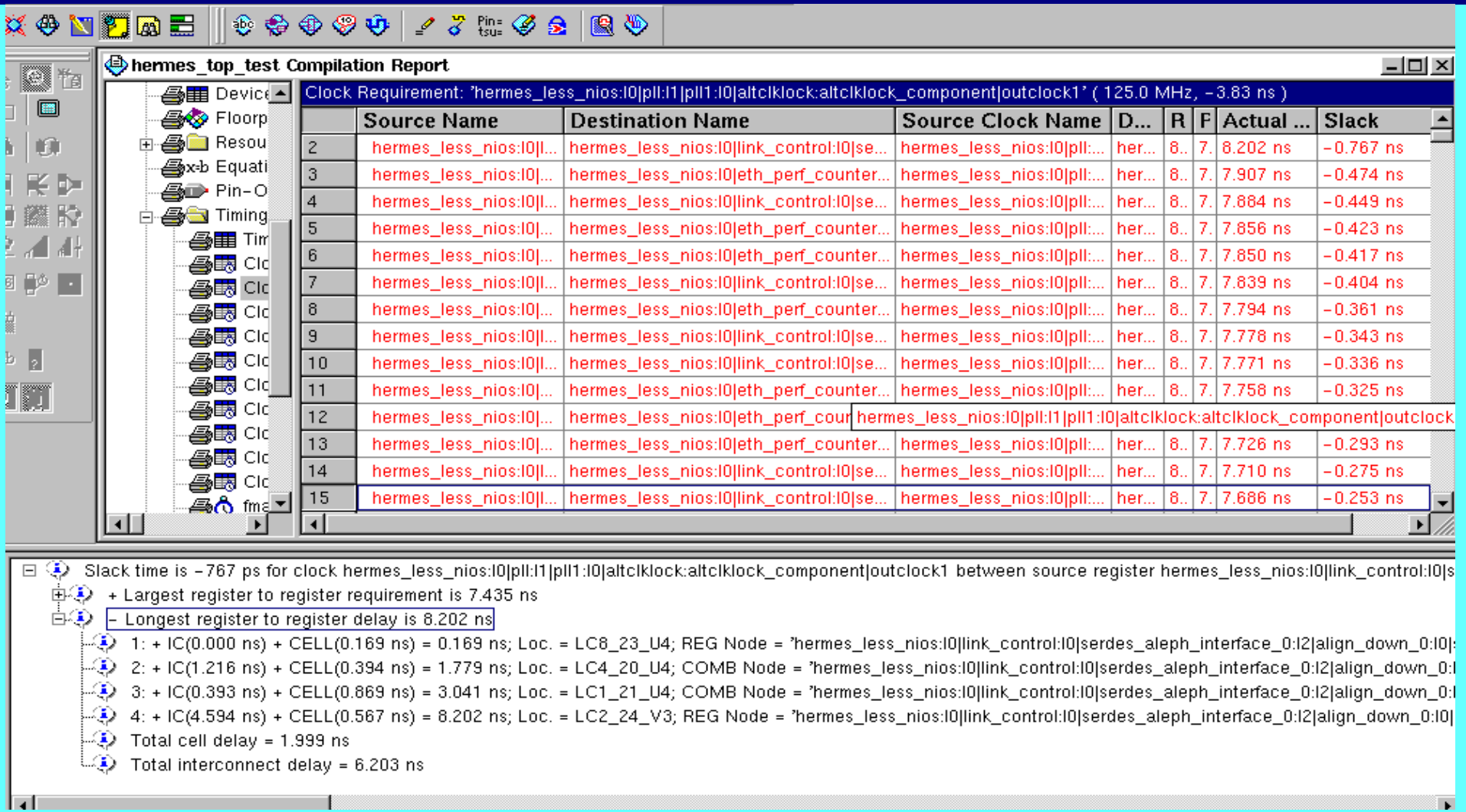
Wire-load Model

With the shrinking of process geometries, the delays incurred by the switching of transistors become smaller. On the other hand, delays due to physical characteristics (R, C) connecting the transistors become larger.

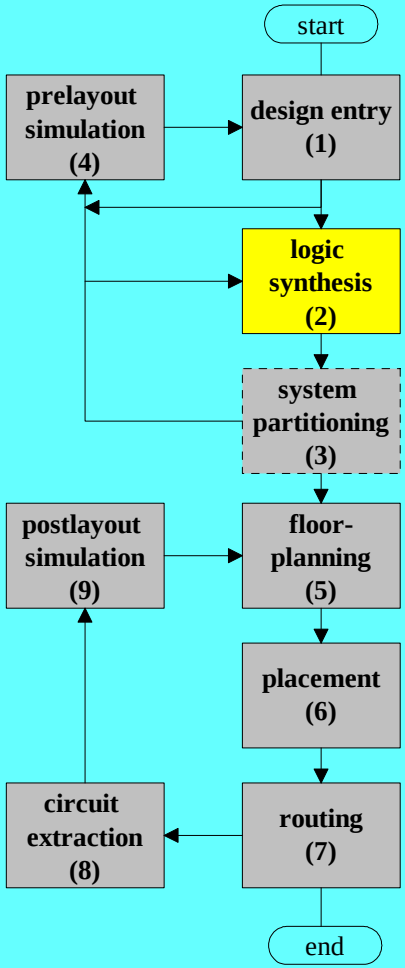
Logical synthesis tools do not take into consideration “physical” information like placement when optimizing the design. Further the wire load models specified in the technology library are based on statistical estimations.

In-accuracies in wire-load models and the actual placement and routing can lead to synthesized designs which are un-routable or don’t meet timing requirements after routing.

Post layout timing analysis (FPGA)



Timing Analysis

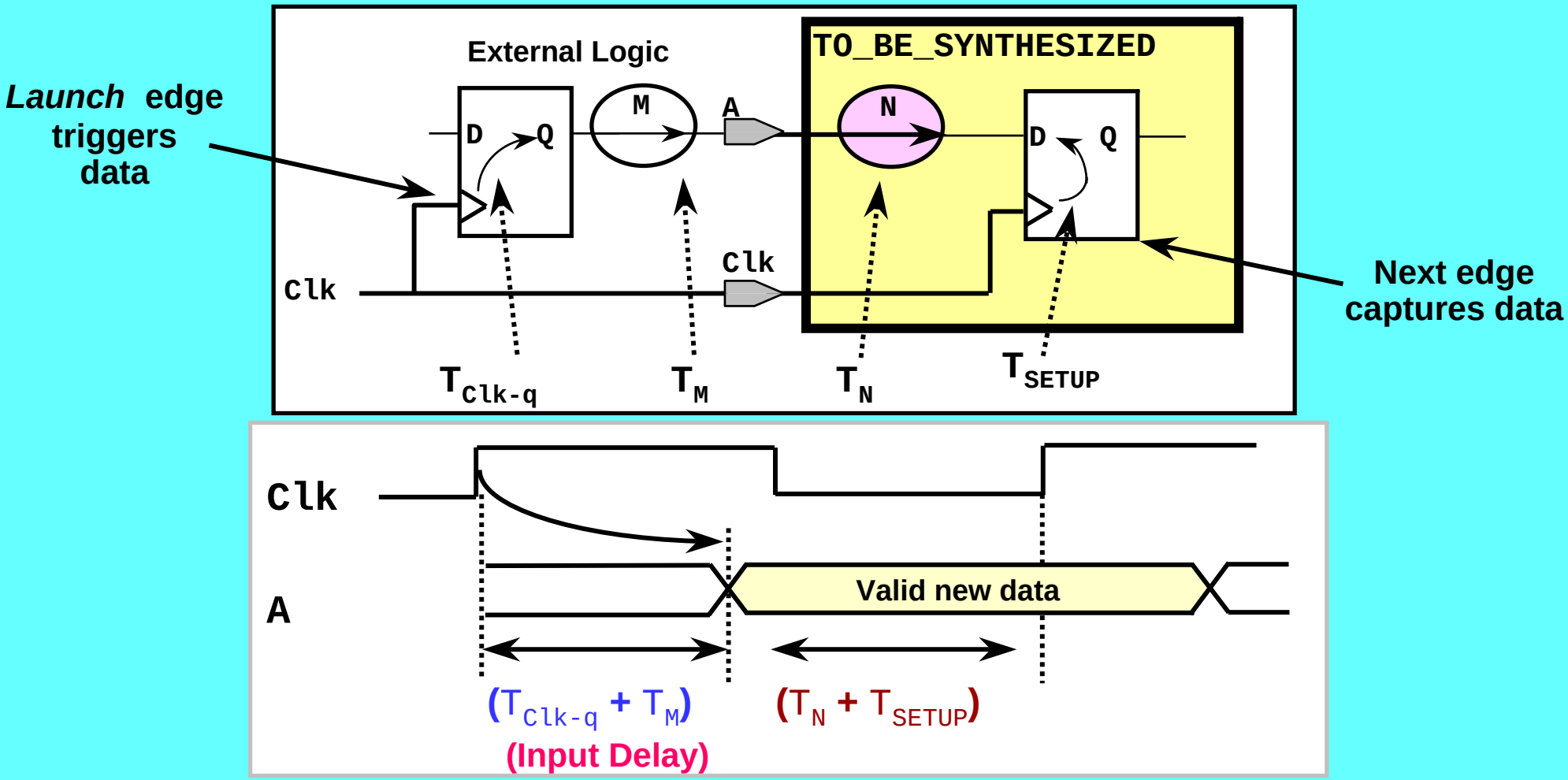


Timing Goals: Synchronous Designs

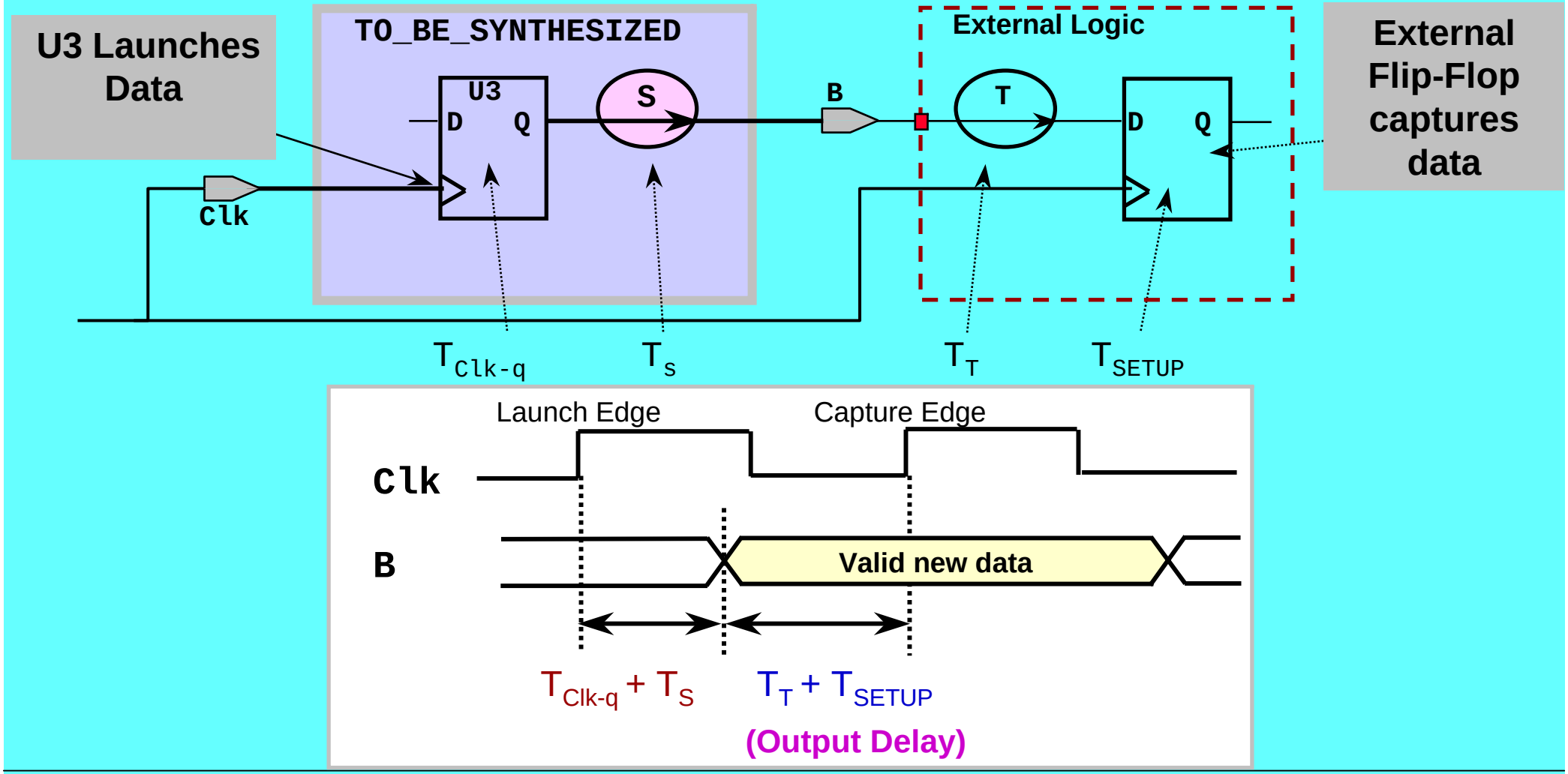
- **Synchronous Designs:**
  - Data arrives from a clocked device
  - Data goes to a clocked device

- **Objective:**
  - Define the timing constraints for all paths within a design:
    - all input logic paths
    - the internal (register to register) paths, and
    - all output paths

Constraining the Input Paths

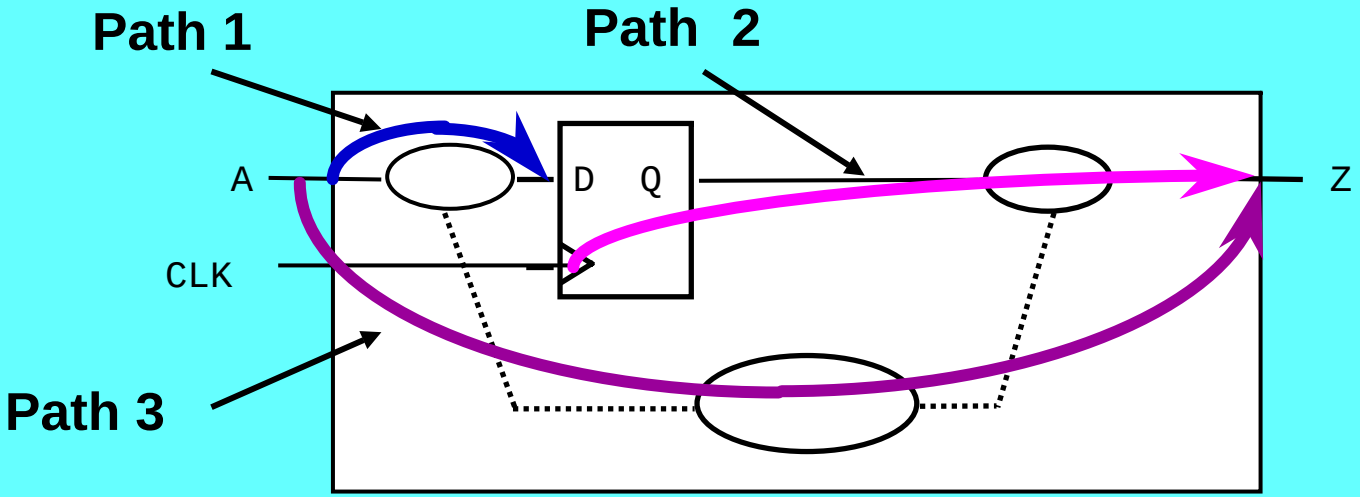


Constraining Output Paths of a Design



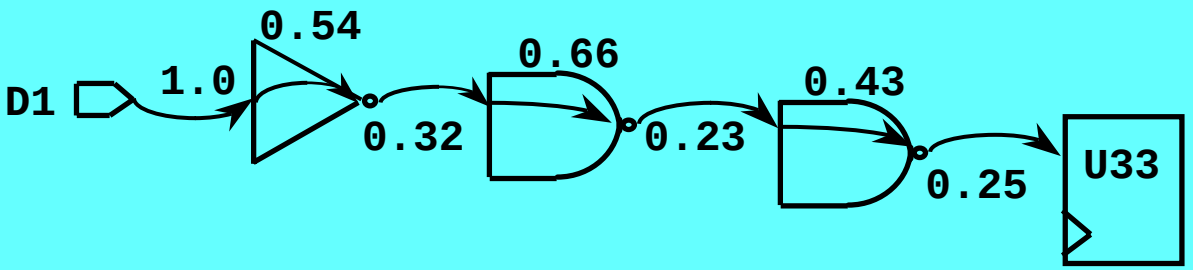
Static Timing Analysis

- Static Timing Analysis can determine if a circuit meets timing constraints without dynamic simulation
- This involves three main steps:
  - Design is broken down into sets of timing paths
  - The delay of each path is calculated
  - All path delays are checked to see if timing constraints have been met



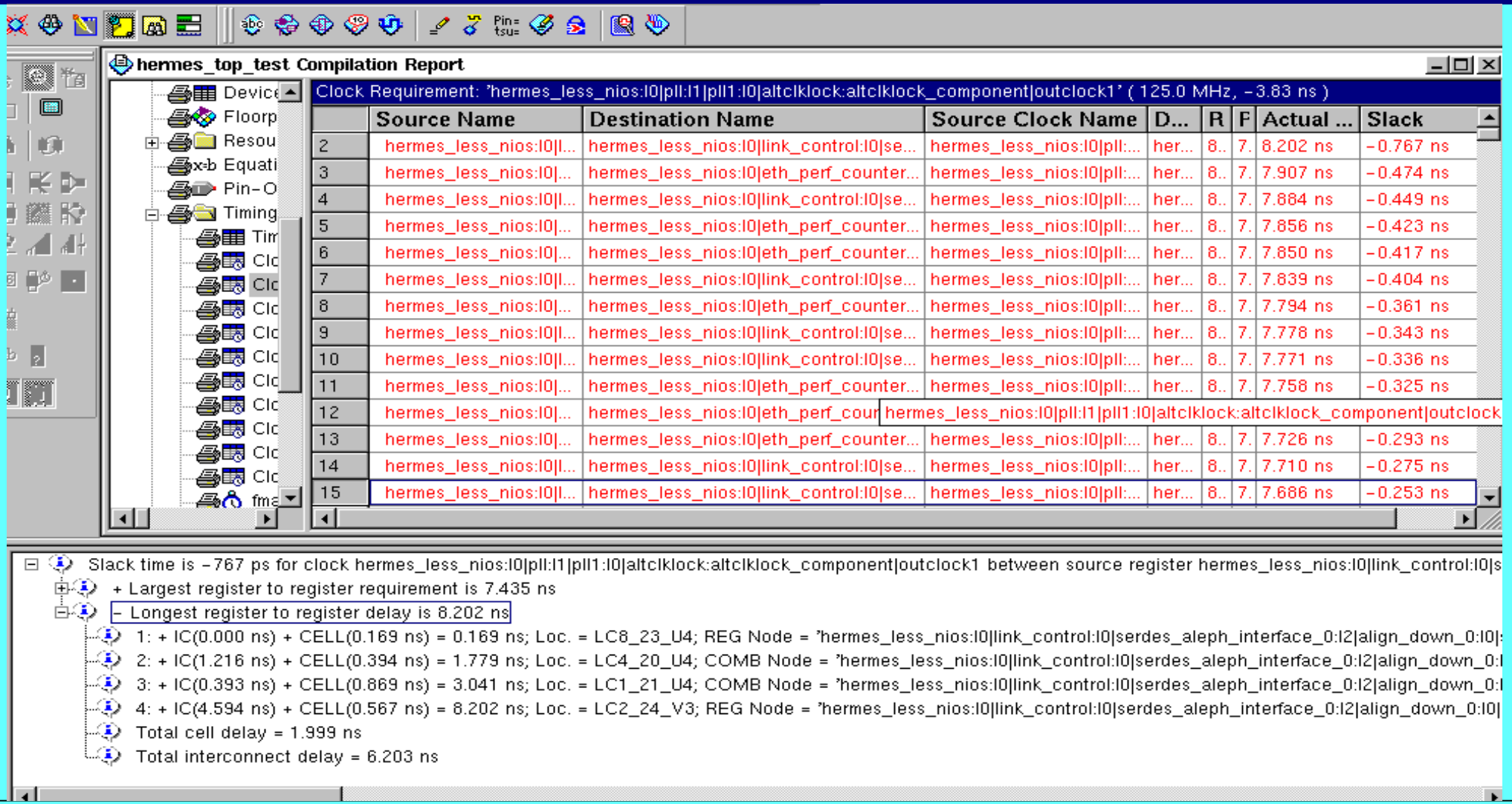
Calculating a Path Delay

To obtain the **path delay** you have to add all the net and cell timing arcs along the path.

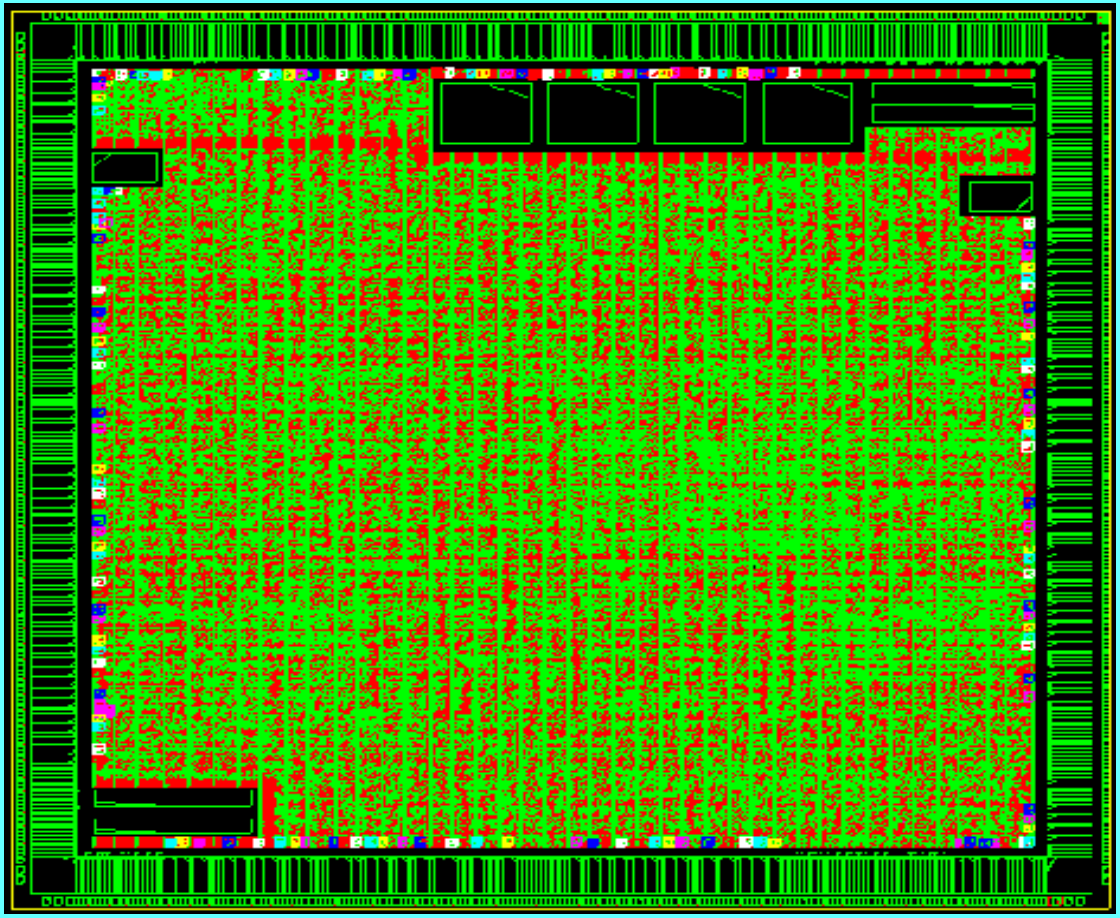


path_delay = (1.0 + 0.54 + 0.32 + 0.66 + 0.23 + 0.43 + 0.25) **= 3.43 ns**

Post synthesis timing (FPGA)

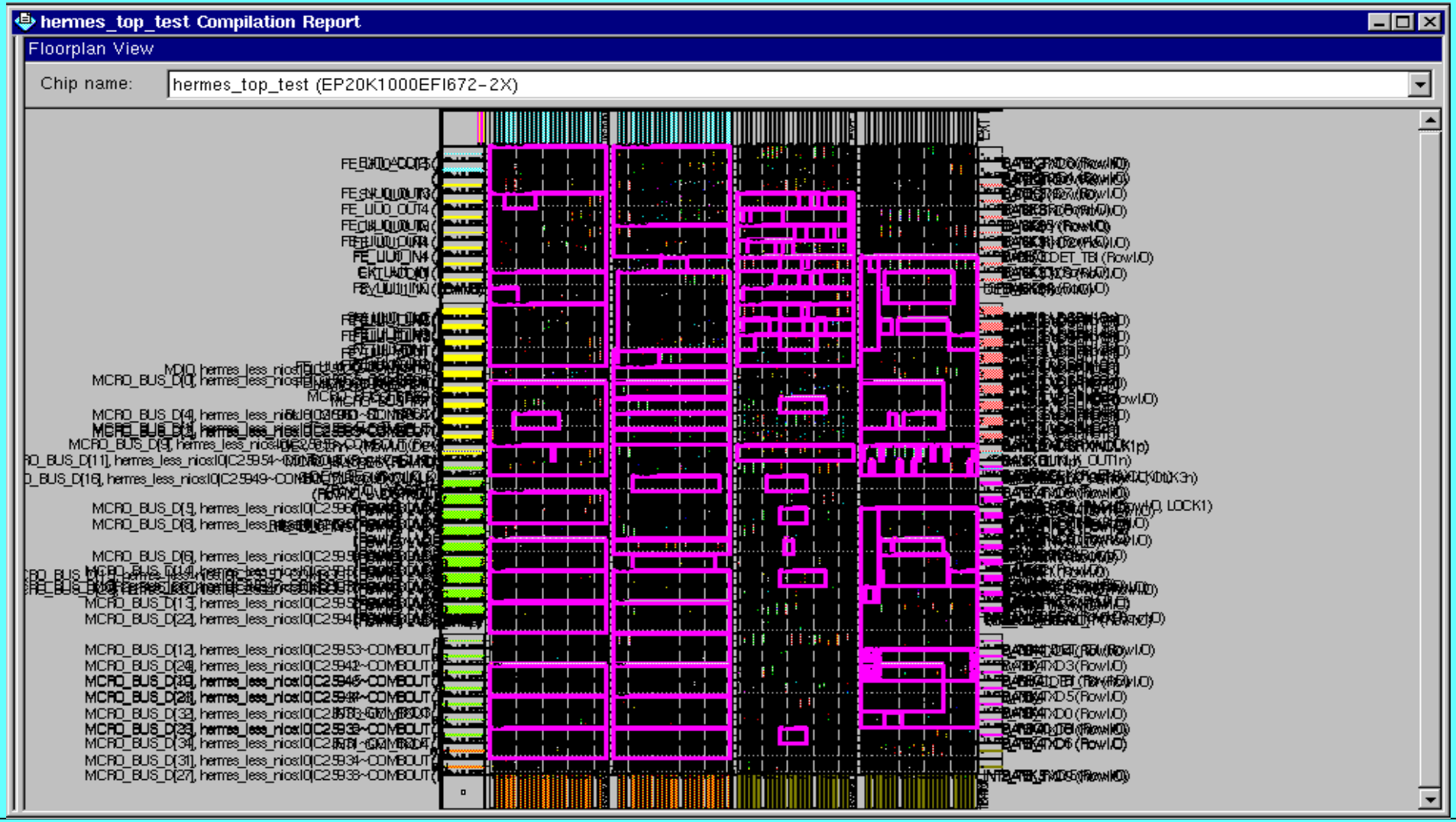


Layout (ASICs)



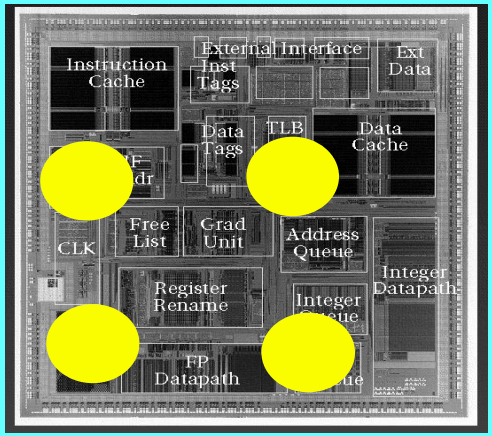
- Placement
- Clock Tree Synthesis
- Routing
- Netlist Optimisation
- Physical Analysis
  - Cross Talk
  - Signal Integrity
  - Electro Migration
  - IR Drop

Layout & Floorplan (FPGAs)



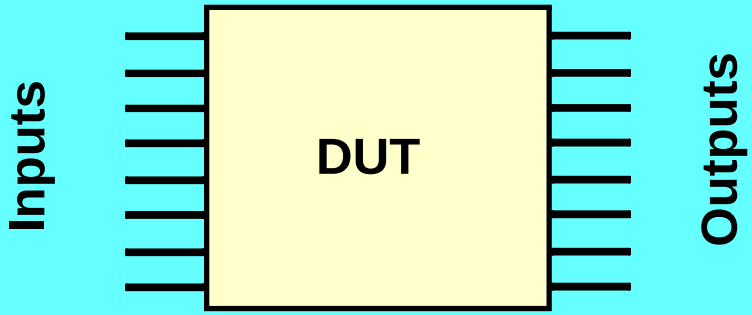
Test for Manufacturing Defects

- The manufacturing test is created to detect manufacturing defects and reject those parts before shipment
- Debug manufacturing process
- Improve process yield



How is Manufacturing Test Performed?

- Automatic Test Equipment (ATE) applies input stimulus to the Device Under Test (DUT) and measures the output response



- If the ATE observes a response different from the expected response, the DUT fails manufacturing test
- The process of generating the input stimulus and corresponding output response is known as Test Generation

ATE

The importance of Test

Cost	Operation description
$1	to fix an IC (throw it away)
$10	to find and replace a bad IC on a board
$100	to find a bad board in a system
$1000	to find a bad component in a fielded system

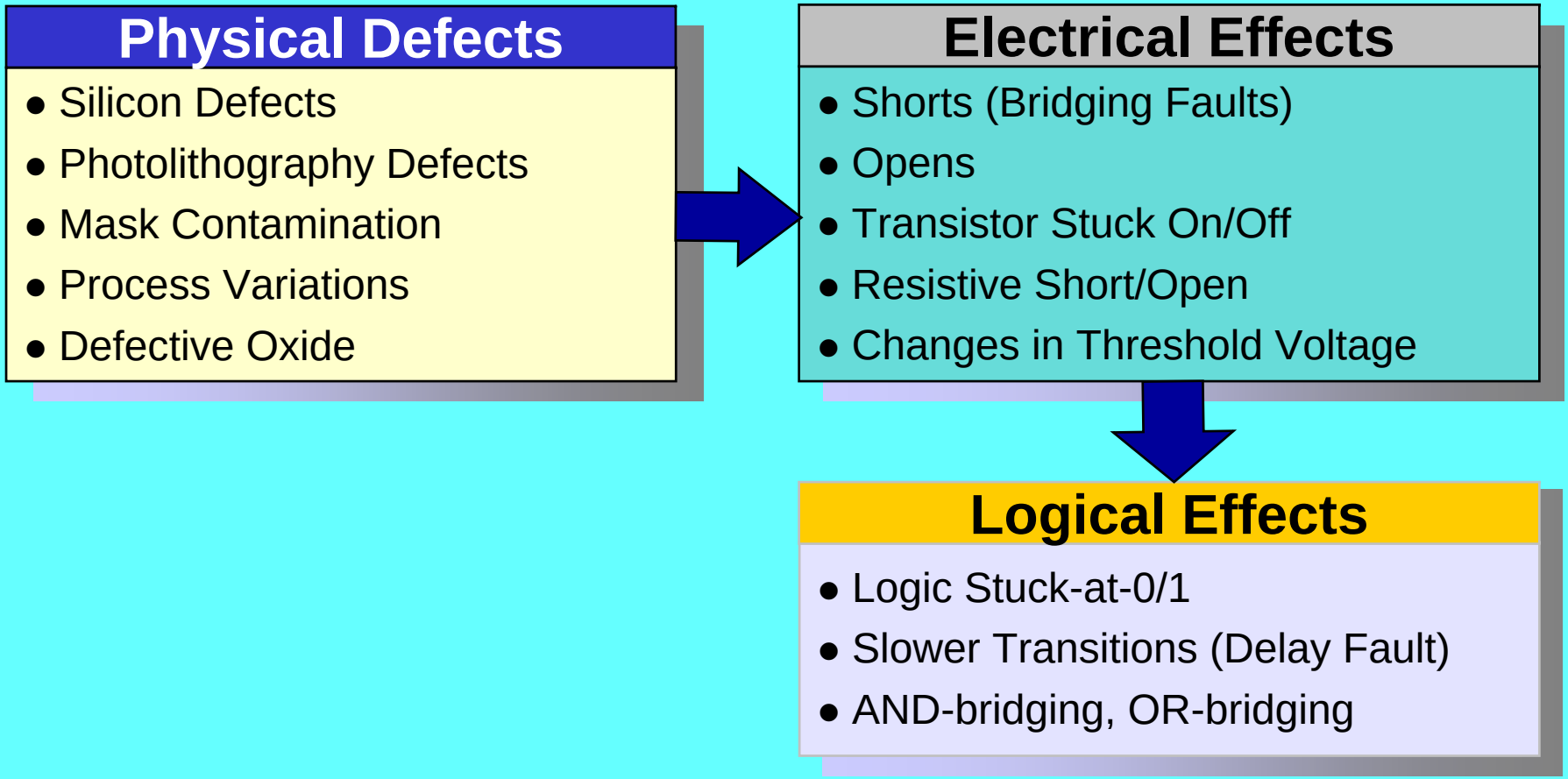
Test Case Study

ASIC defect level	Defective ASICs	Total PCB repair cost	Defective boards	Total repair cost at a system level
5%	5000	$1million	500	$5million
1%	1000	$200,000	100	$1milion
0.1%	100	$20,000	10	$100,000
0.01%	10	$2,000	1	$10,000

Assumptions

- *the number of part shipped is 100,000;*
- *part price is $10;*
- *total part cost is $1million;*
- *the cost of a fault in an assembled PCB is $200;*
- *system cost is $5000;*
- *the cost of repairing or replacing a system due to failure is $10,000.*

Manufacturing Defects



Fault models

Fault level	Physical fault	Logical fault		
		Degradation fault	Open-circuit fault	Short-circuit fault
Chip	Leakage or short between package leads	*		*
	Broken, misaligned, or poor wire bonding		*	
	Surface contamination	*		
	Metal migration, stress, peeling		*	*
	Metallization (open/short)		*	*
Gate	Contact opens		*	
	Gate to S/D junction short	*		*
	Field-oxide parasitic device	*		*
	Gate-oxide imperfection, spiking	*		*
	Mask misalignement	*		*

Stuck-at Fault Model

The **single stuck-at fault (SSF)** model assumes that there is just one fault in the logic we are testing.

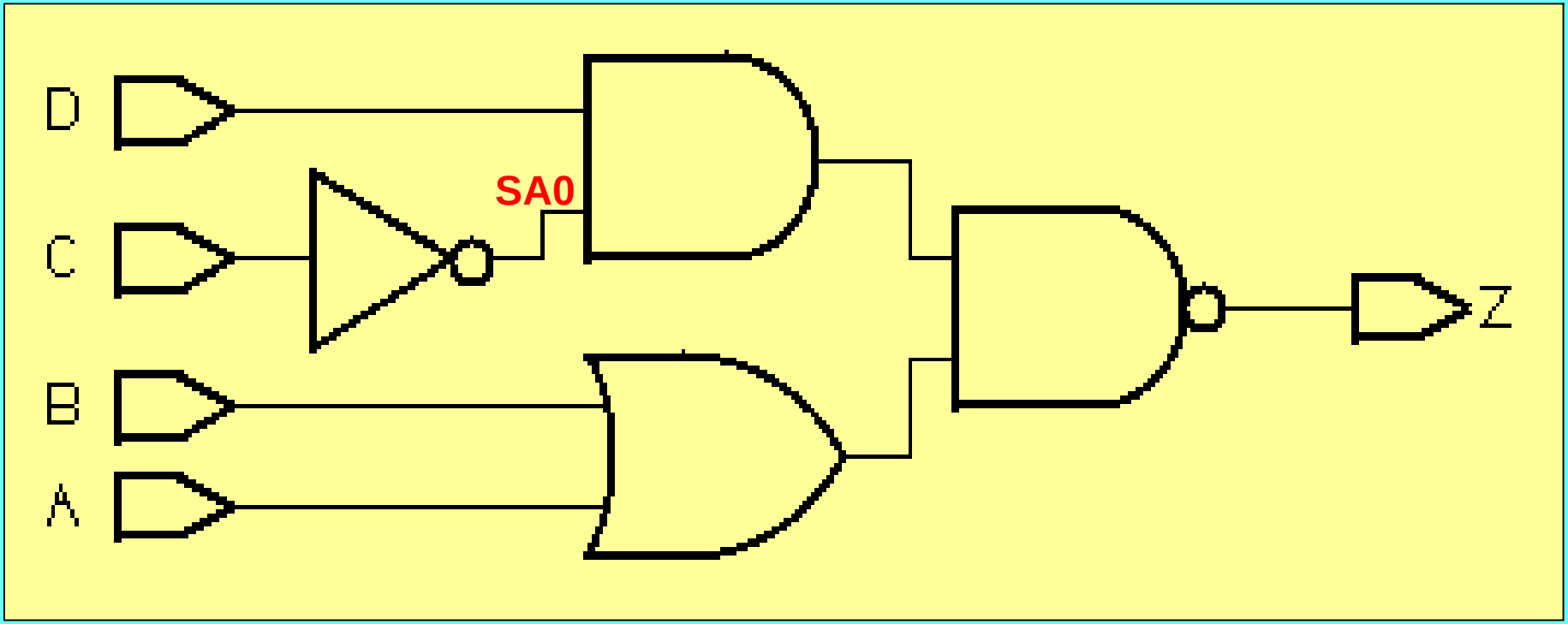
We use a SSF model because a **multiple stuck-at fault model** is too complicated to implement.

In the SSF model we further assume that the effect of the physical fault is to create only two kinds of logical fault (**SA1** and **SA0**). The place where we inject faults is called the **fault origin (net/input/output faults)**.

When a fault changes the circuit behaviour, the change is called the **fault effect**. Fault effects travel through the circuit to other logic cells causing other fault effects (**fault propagation**).

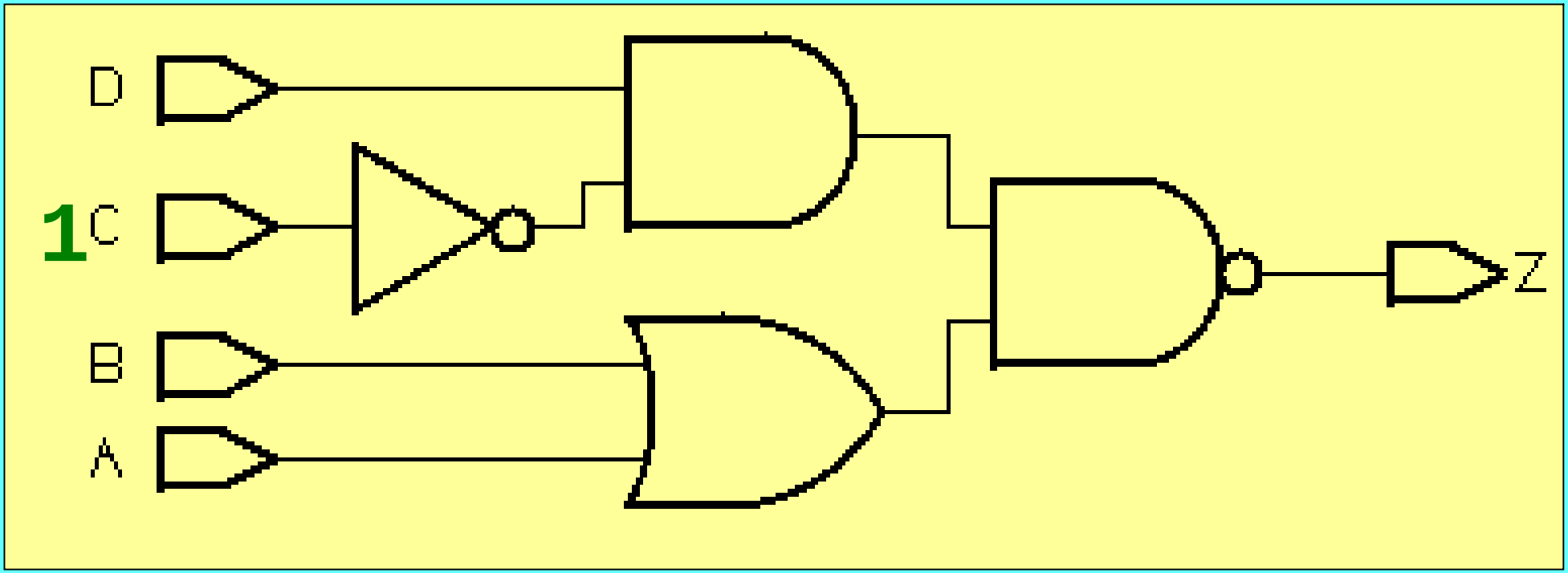
Single “Stuck-at” Fault Model: Example

Model manufacturing defects with a “Stuck-at” Fault



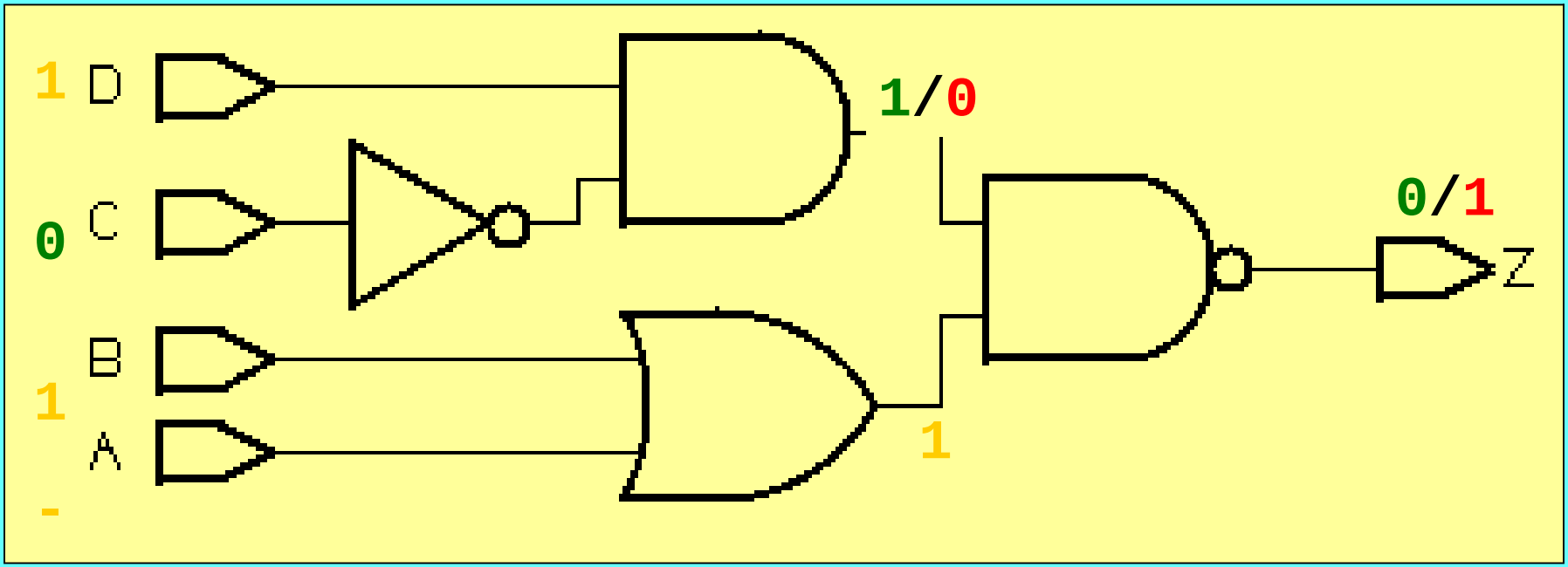
Controllability

Ability to set internal nodes to a specific value



Observability

Ability to propagate the fault effect from an internal node to a primary output port



Fault collapsing

Stuck-at faults attached to different points may produce identical fault effects.
Using **fault collapsing** we can group these equivalent faults into a **fault-equivalent class (representative fault).**

If any of the test that detect a fault B also detects fault A, but only some of the the test for fault A also detect fault B, we say that A is a **dominant fault** (some texts uses the opposite definition)**.** To reduce the number of tests we will pick the test for the dominated fault B (**dominant fault collapsing).**

Example: output SA0 for a two-input NAND dominates either input SA1 faults.

Fault Simulation and Fault Coverage

We use **fault simulation** to see what happens in a design when deliberately introduce faults. In a production test we only have access to the package pins: **primary inputs/outputs (PI/PO)**.

To test an ASIC we must devise a series of sets of input **patterns** that will detect **any** faults.

If the simulation shows that the POs of the faulty circuit are different than the POs of the good circuit at any strobe time, then we have a **detected fault**; otherwise we have an **undetected fault**. At the end of the simulation we can find the **fault coverage**

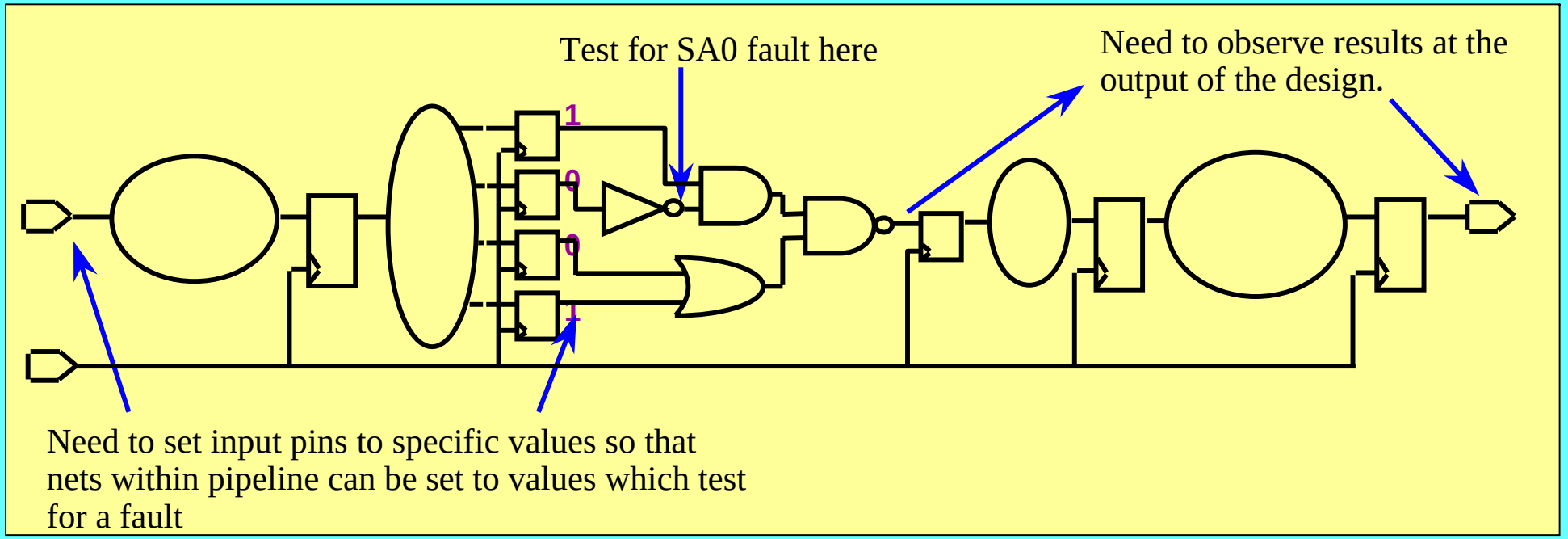
$$\text{fault coverage} = \frac{\text{detected faults}}{\text{detectable faults}}$$

Fault Coverage and Defect Coverage

Fault Coverage	*Average defect level*	*Average Quality Level (AQL)*
50%	7%	93%
90%	3%	97%
95%	1%	99%
99%	0.1%	99.9%
99.9%	0.01%	99.99%

These results are experimental and they are **the only justification** for our assumptions in adopting the SSF model.

Testing a Multistage, Pipelined Design



Each fault tested requires a **predictive** means for both controlling the **input** and **observing** the results downstream from the fault.

Scan Chains

Gains:

- Scan chain initializes nets within the design (adds controllability);
- Scan chain captures results from within the design (adds observability).

Paid price:

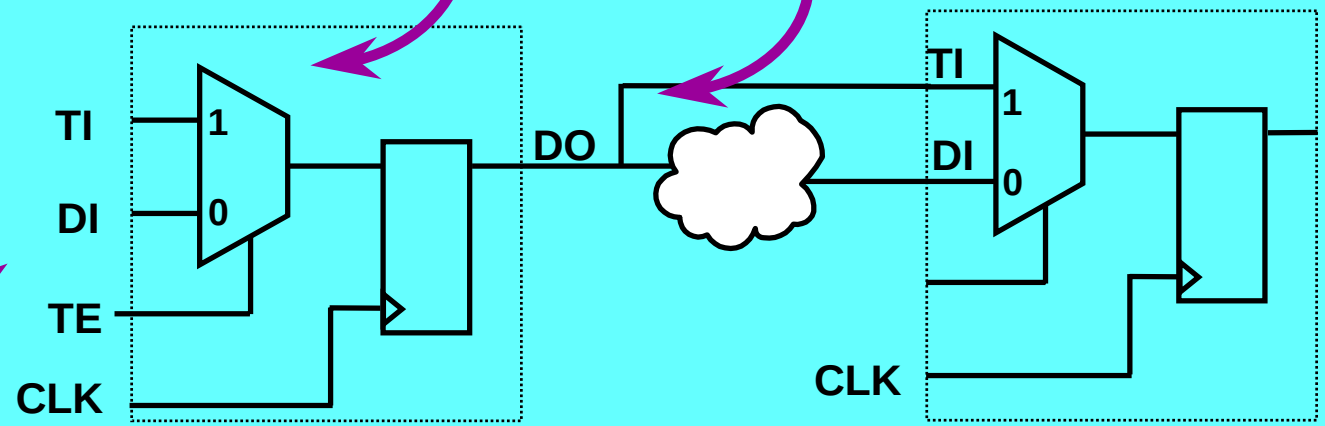
- Inserting a scan chain involves replacing all Flip-Flops with scannable Flip-Flops;
- Scan FF will affect the circuit timing and area.

Inaccuracy Due to Scan Replacements

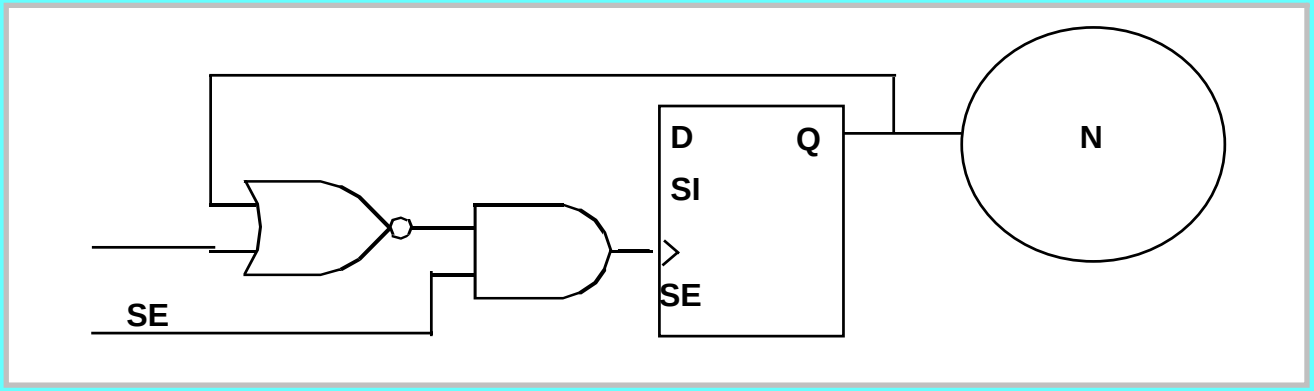
Larger area than non-scan registers;

Additional fanout and capacitive loading

Larger setup time requirement



Testability Violation: Example



What would happen if, during test, a ‘1’ is shifted into the FF? **We would never be able to “clock” the Flip-Flop!**

Therefore, the FF cannot be allowed to be part of a scan chain. Logic in the net ‘N’ cannot be tested.

The above circuit

- **violates good ‘DFT’ practices;**
- **reduces the fault coverage.**

Synthesizing for Test Summary

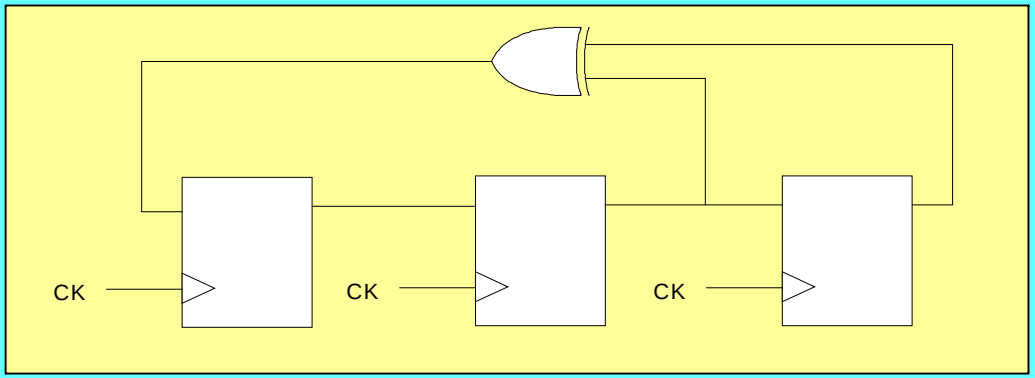
- Test is a **design methodology**: it has its own testability design rules.

- Most problems associated with test can be anticipated and corrected in advance, during the **initial compile of the HDL** code.

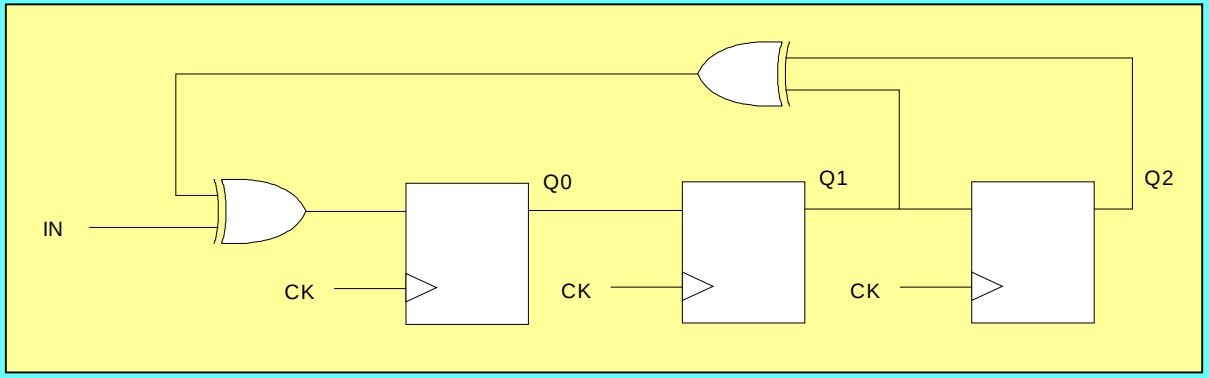
Built-in Self-test

Built-in Self-test (**BIST**) is a set of structured-test techniques for combinatorial and sequential logic, memories, etc.

The working principle is to generate test vectors, apply them to the **circuit under test** (CUT) and then check the response. In order to produce long test vectors **linear feedback shift register** (**LFSR**) are used. By correctly choosing the points at which we take the feedback form an *n-bit* SR we can produce a **pseudo random binary sequence (PRBS)** of a maximal length (2^n-1).



Signature Analysis



If we apply a binary input sequence to IN, the shift register will perform **data compression** on the input sequence. At the end of the input sequence the shift-register contents, `Q0Q1Q2`, will form a pattern called **signature**.

If the input sequence and the **serial-input signature register** are long enough, it is unlikely that two different input sequences will produce the same signature.

If the input sequence comes form logic under test, a fault in the logic will cause the input sequence to change. This causes the signature to change from a known value and we conclude that the CUT is bad.

The IEEE 1149.1 Standard

In 1985 a group of European manufacturers formed the **Joint European Test Action Group,** later renamed in **JTAG (1986).** The JTAG 2.0 test standard formed the basis of the **IEEE Standard 1149.1 Test Port and Boundary-Scan Architecture (1990).**

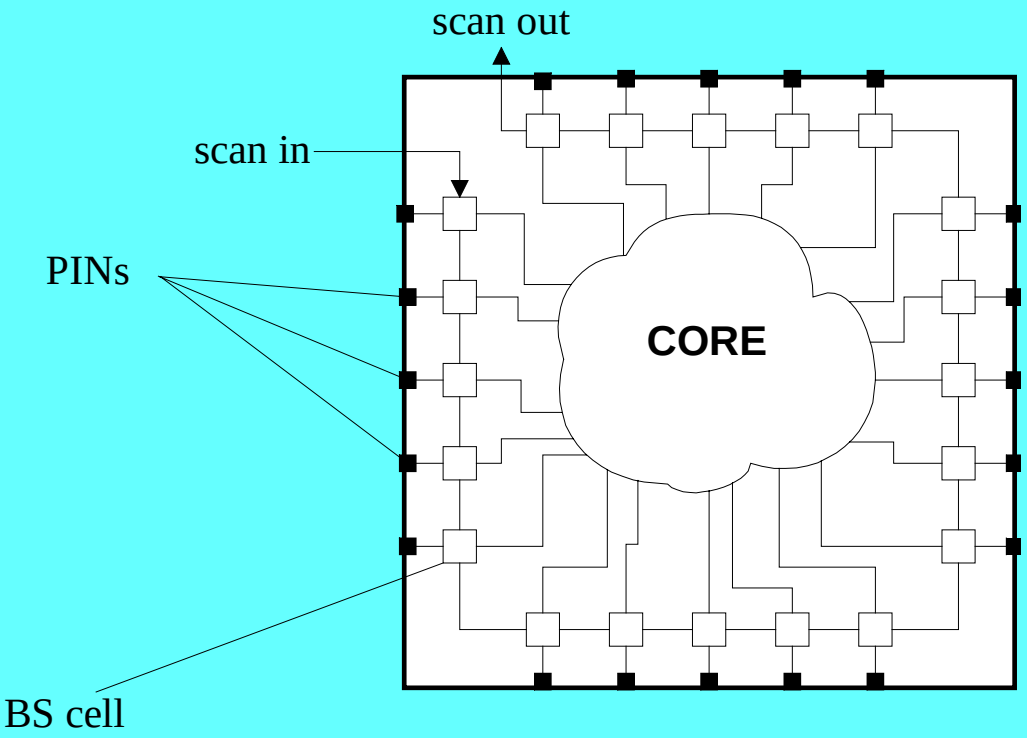
Boundary-Scan test (BST) is a method for testing boards using a four-wire interface (with a fifth optional master reset signal). The BST standard was designed to test boards , but it's also useful to test ASICs.

We can automatically generate test vectors for combinational logic, but **ATPG** (**A**utomatic **T**est **P**attern **G**eneration) is much harder to sequential logic.

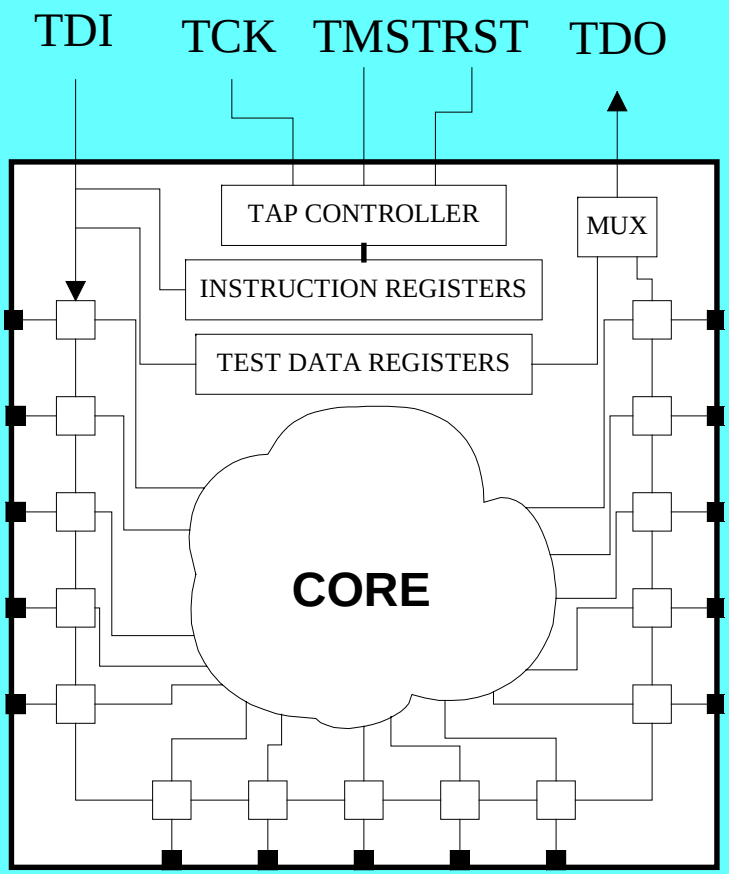
In full scan design we replace every sequential element with a scan flip-flop. The result is an internal form of boundary scan and we can use the IEEE 1149.1 TAP to access an internal scan chain.

Boundary Scan Chain

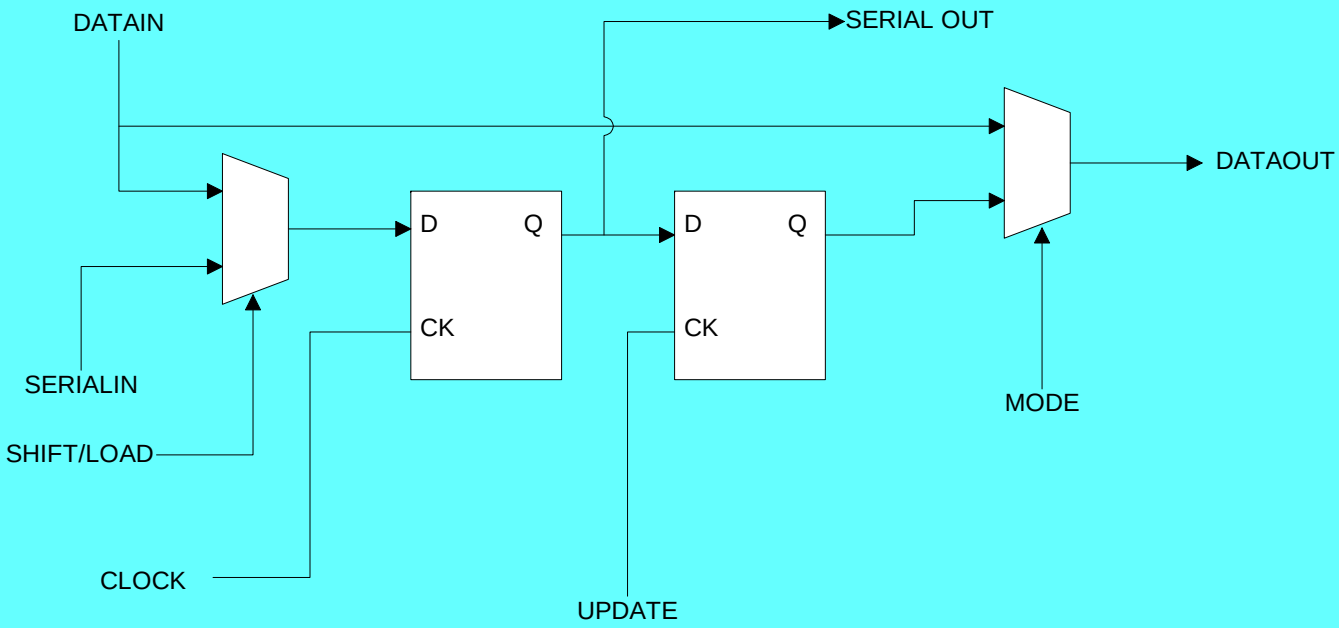
Each IO pin is replaced with a multi-purpose element called Boundary Scan cell.



The BS Architecture



The BS Cell


DATAIN
SERIAL OUT
DATAOUT
D
Q
CK
D
Q
CK
SERIALIN
SHIFT/LOAD
CLOCK
UPDATE
MODE

Quiescent Leakage Current

A CMOS transistor in never completely *off* because of **subthreshold** and **leakage current**.

• **Subthreshold current:** $V_{GS} = 0$ but trough the transistor a current of few pA/μm is flowing.

• **Leakage current:** The sources and drains of every transistor and the junctions between the wells and substrate form parasitic diodes. Reverse-biased diodes conducts a very small leakage current.

The **quiescent leakage current (I_{DDQ})** is the current measured when we test an ASIC with no signal activity and must have the same order of magnitude than the sum of the subthreshold and leakage current.

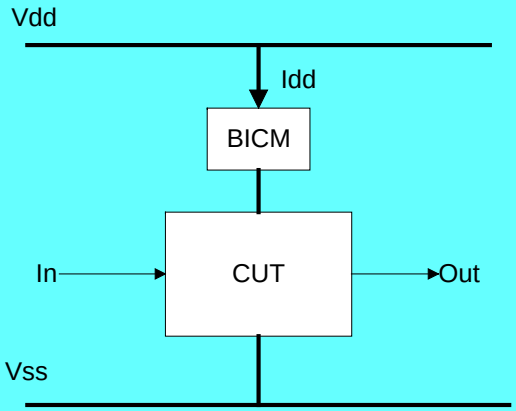
A measurement of more current than this in a non-active CMOS ASIC indicates a problem with the chip manufacture (or the design).

Measuring the I_{DDQ} Externally

- With specific ATE (Automatic Test Equipment)
  - no dedicated circuits on chip;
  - no impact on chip performance;
  - external ad-hoc ATPG;
  - time consuming.

Measuring the I_{DDQ} on Chip

- With a ***current sensor (*BICM – Built In Current Monitor)**
  - dedicated circuitry;
  - internal ATPG or scan to lead the device in the quiescent mode;
  - impact on chip due to voltage drop over the BICM;



IDDQ Test

- The IDDQ test reveals shorts but not opens;
- A 100% coverage may be expensive;
- Multiple samples of the current are necessary for a meaning test.
- Quiescent I_{DD} depends from the design but also mainly from process and package;

BISR: Increasing the Process Yield

Increasing size, density and complexity in memory technologies lead to higher **defect density** and, consequently, a decrease in process yield.

A cost- and time-effective solution is built-in self-repair (**BISR**). It consists of replacing, on silicon, the defective memory columns by **spare columns** available next to the functional memory. BISR is implemented at the column, row, block or bit level. Using non-volatile blocks to store the memory reconfiguration improves the memory production yield.

BISR: Increasing Reliability

Reliability aspect is also considered by chip manufacturers. High memory size and high-end memory technologies often lead to an increasing number of defects that happen **during the product life**.

BISR solutions allow the memory to be tested in the filed and the defective memory blocks to be replaced by redundant blocks that are not defective. If the memory contains critical contents, transparent BISR allows defective blocks to be tested and replaced without losing the original memory content.

Thank You