

Testarea și Fiabilitatea Sistemelor de Calcul
Curs 1

Introducere - partea I

Ce conține acest capitol?

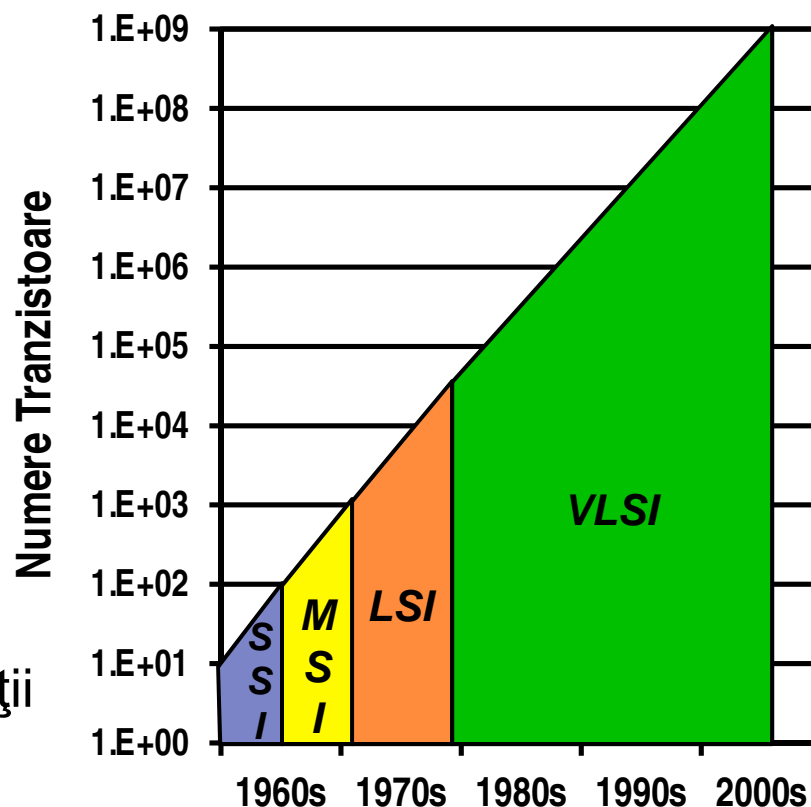
- Diferențierea noțiunilor de testare - verificare
- Analiză a următoarelor aspecte:
 - Importanța testării în procesul de proiectare și manufactură a circuitelor
 - Probleme actuale în generarea de teste și modelarea defectelor
 - Nivele de abstractizare ale testării VLSI
- Imagine de ansamblu asupra testării VLSI

Introducere în Testarea Circuitelor Integrate

- ❑ **Introducere**
- ❑ **Testarea pe durata de viață a unui circuit**
- ❑ **Generarea testelor**
- ❑ **Modelarea defectelor**
- ❑ **Nivele de abstractizare**
- ❑ **Imagine de ansamblu asupra tehnologiei de testare**
- ❑ **Concluzii**

Introducere

- ❑ Circuitele Integrate (ICs) au crescut în mărime și complexitate după anii 1950
 - Integrare pe scală mică (SSI)
 - Integrare pe scală medie (MSI)
 - Integrare pe scală largă (LSI)
 - Integrare pe scală foarte largă (VLSI)
- ❑ *Legea lui Moore*: densitatea CI se dublează la fiecare 18 luni
 - Creșterea densității și complexității aduce noi probleme în testare



Importanța Testării

- ❑ Legea lui Moore pornește de la scăderea lărgimii canalului unui tranzistor
 - Scăderea a avut loc de la zecimi de μm la zecimi de nm pentru tranzistori și interconexiuni
- ❑ Frecvențele de lucru au crescut de la sute de KHz la GHz
- ❑ Scăderea dimensiunilor tranzistoarelor au făcut să crească probabilitatea generării de defecte în timpul manufacturii
 - Este suficient un tranzistor cu defect pentru a fi defect circuitul integrat -> testarea este esențială

Importanța Testării

- *Creștere exponențială*: costul detectării unui CI defect crește exponențial în domeniile:
 - dispozitiv → PCB → sistem → sistem în funcțiune
 - Testarea este necesară la fiecare din aceste nivele
- Testarea are loc și în timpul:
 - Manufacturării pentru creșterea productivității
 - Metodă folosită: Failure mode analysis (FMA)
 - Funcționării în timp real, pentru a reduce rata de defecte:
 - În această fază defectele se repară

Necesitatea Testării

□ Tendințele tehnologice VLSI și impactul asupra testării [“Design Verification & Testing” UMBC, CMPE 418]

Perioada	1997 - 2001	2003 - 2006	2009 - 2012
Dimensiunea tranzistorului (μm)	0.25 – 0.15	0.13 – 0.10	0.25 – 0.15
Milioane tranzistori / cm^2	4 - 10	18 - 39	84 - 180
Numărul de straturi tehnologice	6 - 7	7 - 8	8 - 9
Mărimea <i>dye</i> -ului, în mm^2	50 - 385	60 - 520	70 - 750
Numărul de pini	100 - 900	160 - 1475	260 - 2690
Frecvența folosită, în MHz	200 - 730	530 – 1100	840 - 1830
Tensiune, în V	1,2 – 2,5	0,9 – 1,5	0,5 – 0,9
Putere, în W	1,2 - 61	2 - 96	2,8 - 109

Necesitatea Testării

□ Exemplu practic [“Design Verification & Testing” UMBC, CMPE 418]

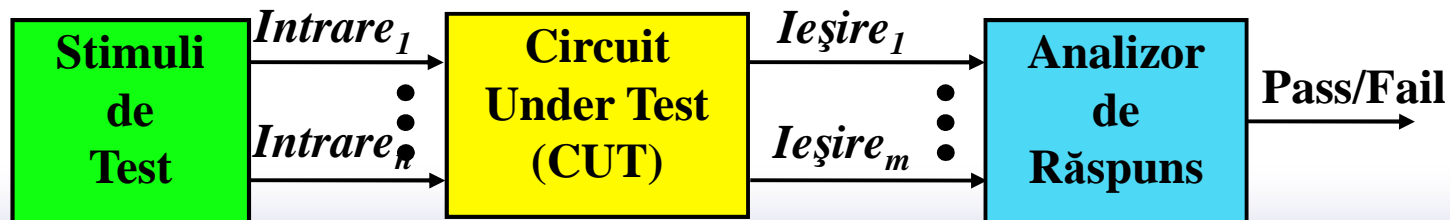
Costul echipamentului automatizat (ATE) estimat pentru un CI cu frecvență > 250 MHz:

- Necesari un tester pentru 500 MHz, al cărui cost este:
 $1.2 \text{ M \$} + (1024 \text{ pini} \times 3000 \text{ \$ / pin}) = 4272 \text{ M \$}$
- Costul de întreținere: 1439 M\$/an (4.5 cenți/s)
- Timpul de testare a unui ASIC: 6 s -> 27 cenți
- Pentru un randament de 65%, prețul de testare: $27 / 0.65 = 41.5 \text{ cenți}$

Testarea în Cursul Duratei de Viață a unui CI

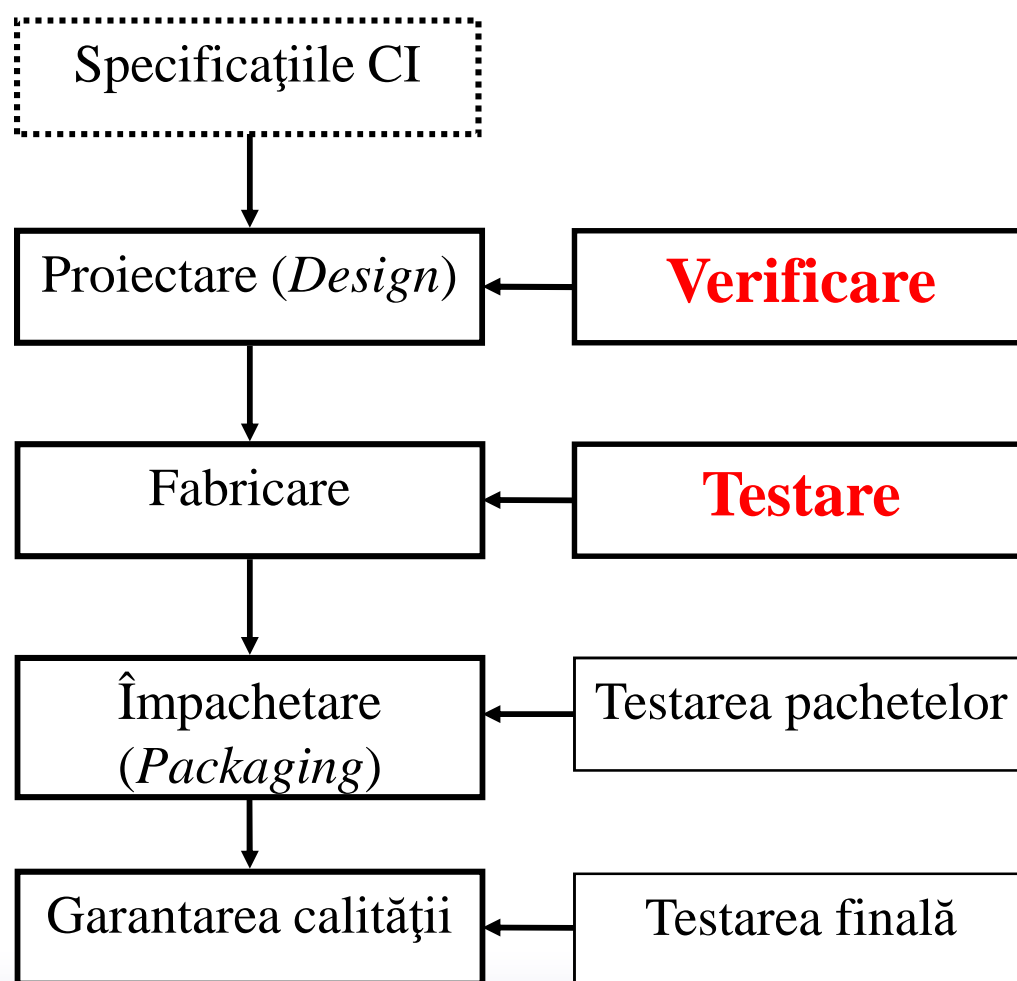
□ Testarea constă în:

- Aplicarea unui set de stimuli de intrare asupra
- Intrărilor IC numit *circuit under test* (CUT) și
- Analiza răspunsurilor circuitului testat
 - Dacă este incorect (eroare), CUT este defect
 - Dacă este corect (fără eroare), CUT nu are defecte



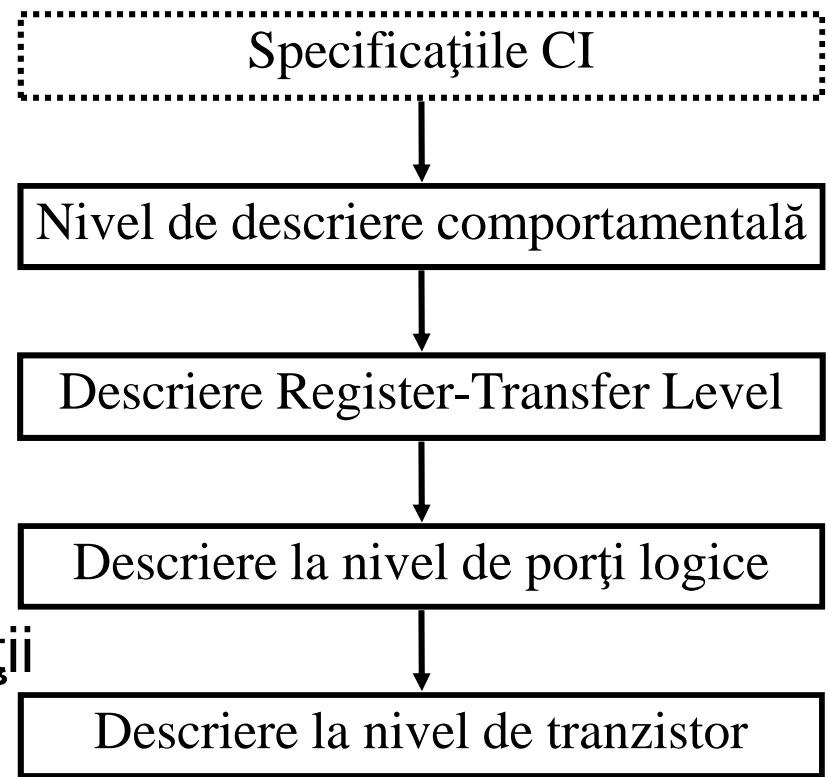
Testarea în Contextul Realizării unui CI

- ❑ Verificarea circuitului detectează erorile de proiectare
 - Corecțiile sunt făcute înainte de fabricare
- ❑ Testarea circuitului detectează erorile de fabricație
 - Un defect reprezintă o imperfecțiune fizică care generează funcționare incorectă



Verificarea unui Circuit Integrat CI

- ❑ CI deține diferite nivele de abstractizare
 - Aplicațiile CAD realizează sinteza circuitului de la nivelul RTL la nivel fizic
- ❑ Simularea folosită în diverse faze pentru:
 - Detectarea erorilor de proiectare din RTL/specificații
 - Detectarea performanțelor circuitului după sinteză



Verificarea și Testarea unui CI

□ Verificare

- Garantează corectitudinea CI
- Realizată prin simulare, analiză formală, emulare
- Realizată *înainte* de fabricare

□ Testare

- Garantează corectitudinea CI
- Realizată prin:
 - Generare de teste, în timpul dezvoltării CI
 - Aplicarea testelor, după obținerea CI pe wafer
- Realizată pe *fiecare* circuit integrat în parte, *după* fabricare

Randament și Rata Pierderilor

□ Defectele de fabricație pot genera CI defecte

- Randament (*yield*) = $\frac{\text{număr_de_componente_acceptate}}{\text{număr_total_de_componente}}$

□ 2 tipuri de pierderi de randament

- Catastrofic – datorat defectelor aleatoare
- Parametric – datorat variațiilor de proces

□ Rezultate nedorite în timpul testării datorate:

- Circuite defecte nedetectate de teste
 - Numită rată a pierderilor = $\frac{\text{număr_de_defecte_acceptate}}{\text{număr_total_de_componente}}$
- Circuite bune nu trec testul
 - Datorită unor teste scrise incorect

Manufacturea sistemelor electronice

□ Un sistem electronic conține

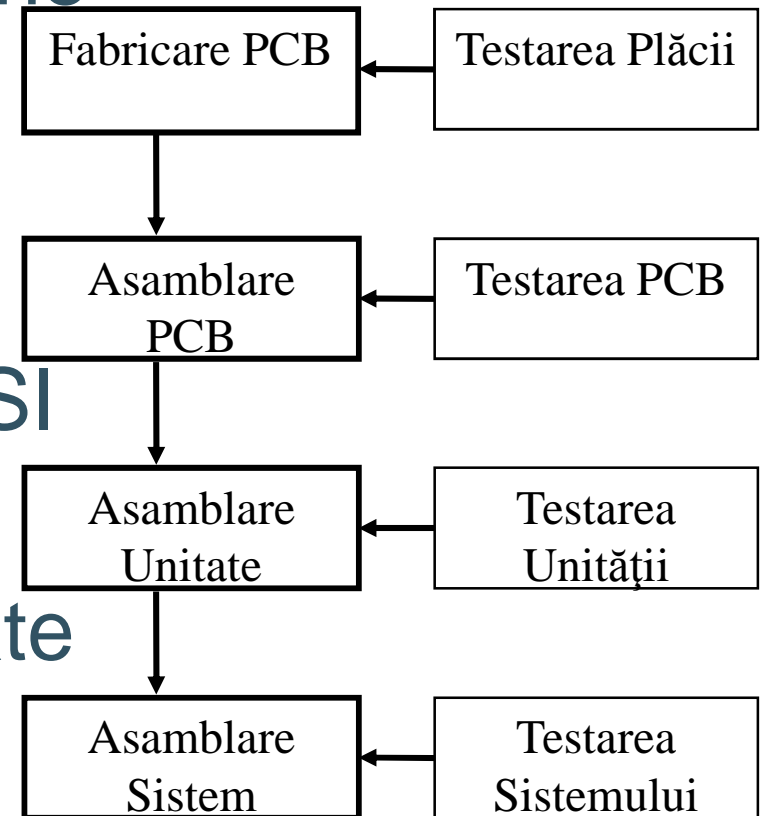
- PCB-uri compuse din
 - Dispozitive VLSI

□ Fabricarea PCB-ului este similară fabricării unui VLSI

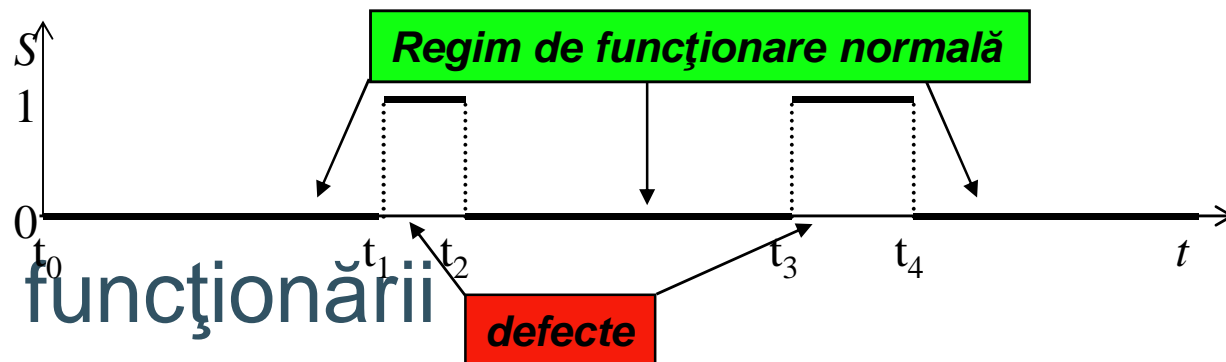
- Ambele pot avea defecte

□ Asamblarea PCB-ului poate conține defecte

- Testarea este necesară în toate etapele manufacturii



Funcționarea la nivel de sistem



□ Defectele pot apărea în cursul funcționării

□ Legea exponențială de defectare

- Intervalul de funcționare normală a sistemului este dat de o distribuție aleatorie exponențială

□ Fiabilitate

- Reprezintă probabilitatea ca un sistem să funcționeze până la momentul t : $P(T_n > t) = e^{-\lambda t}$
- Rata de defectare λ este suma ratelor de defectare ale componentelor, λ_i : $\lambda = \sum_{i=0}^k \lambda_i$

Funcționarea la nivel de sistem

- Timpul mediu între defectări este dat de: Mean Time Between Failures: $MTBF = \int_0^{\infty} e^{-\lambda t} dt = \frac{1}{\lambda}$
- Timpul de Reparare (R) are de asemenea o distribuție exponențială: $P(R > t) = e^{-\mu t}$
 - μ este rata de reparare
- Timpul mediu de reparare este dat de: Mean Time To Repair (MTTR): $MTTR = \frac{1}{\mu}$
- Rata de timp în care circuitul funcționează normal se numește **Disponibilitate** și este dată de: $\frac{MTBF}{MTBF + MTTR}$

Testarea la nivel de sistem

- Testarea se face pentru a asigura o Disponibilitate mare a sistemului
- Tipuri de testare la nivel de sistem:
 - Testare On-line – are loc în timpul funcționării
 - Testare Off-line – are loc atunci când sistemul (sau doar o parte a lui) nu funcționează
 - Se face periodic, dar fără a întrerupe fluxul de operații
 - Folosit la diagnosticarea (identificare și localizare) componentelor care se pot înlocui, pentru a micșora timpul de reparare

Generarea Testelor

- Un test este reprezentat de o secvență de valori de testare, numite vectori de test, aplicate circuitului CUT, ale cărui ieșiri sunt monitorizate și analizate
 - Testare exhaustivă – când sunt aplicate toate secvențele posibile de testare
 - Testare funcțională – când sunt aplicate toate intrările tabelului de adevăr (CUT combinațional)
- Rata de Acoperire a Defectelor (Fault Coverage) este o măsură cantitativă a calității unui set de vectori de test

Generarea Testelor

- Rata de acoperire a defectelor pentru un set de vectori de test = $\frac{\text{număr_defecte_identificate}}{\text{număr_total_defecte}}$
- Rata de acoperire de 100% este imposibilă datorită defectelor nedetectabile
- Eficiența detectării este dată de $\frac{\text{număr_defecte_identificate}}{\text{număr_total_defecte} - \text{număr_defecte_nedetectabile}}$
- $Rata_refuz = 1 - randament^{(1 - rata_acoperire)}$
 - Exemplu: un PCB format din 40 CI, fiecare cu o rată de acoperire de 90% și un randament de 90%, are o rată de refuz de 41.9%

Generarea Testelor

- **Scop:** găsirea unui set optim de vectori de test pentru un maxim de acoperire de defecte
- Rata de acoperire de defecte se poate determina prin simularea defectelor
 - Sunt necesare modele de defecte care să emuleze efectul acestor defecte
- Modele defectelor trebuie să:
 - Asigure eficiență în simularea circuitelor afectate
 - Asigure acuratețe în privința modelării
- Nu există modele de defecte globale care să acopere un ansamblu de defecte fizice

Exerciții și Aplicații

- ❑ O unitate de măsură des întâlnită în calculele de fezabilitate este numărul de defectări pe durata a 10^9 ore, notată cu FITS. Calculați timpul mediu între defectări (MTBF) al unui sistem cu 500 componente, în care fiecare componentă are o rată de defectare de 1000 FITS.
- ❑ Care este timpul mediu de reparare al unui sistem pe an, dacă disponibilitatea acestuia este de 99.999%?
- ❑ Care este procentul componentelor defecte dintr-un lot dacă randamentul este de 50%, iar rata de acoperire a defectelor de 90%?

Testarea sistemelor de calcul

Course 2

11 Martie/ 2022

What this course is about?

- To teach necessary concepts for tools of verification
- Describe a process for carrying out effective functional verification
- Present techniques for applying stimulus and monitoring the response of a design utilizing bus functional models
- Present the importance of behavioral modeling

Prior Knowledge

- This course focuses on functional verification of hardware design using either Verilog or System Verilog
- Expect students to have a basic knowledge of one of these languages
- Expect students to have basic understanding of digital hardware design
- Course will focus more on System Verilog

Course structure

- 5/6 weeks of theoretical knowledge
- 10 weeks of practical knowledge(Lab)
- Final course grade: 50% course, 50% laboratory (**TBD**)



60% – 80% of effort in digital design is
dedicated to verification

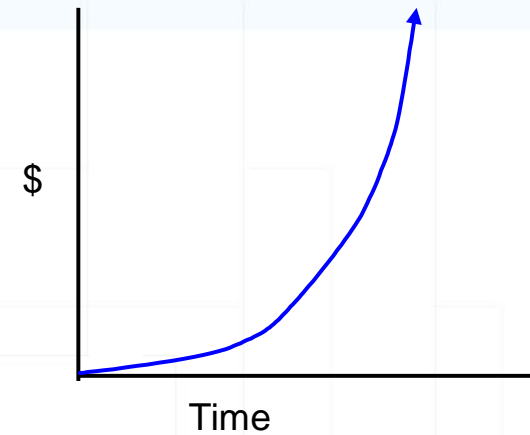
Why HDL Verification?

- I mentioned 60% - 80% time spent in verification – WHY??
 - Product time-to-market
 - hardware turn-around time
 - volume of "bugs"
 - Development costs

Why HDL Verification? (cont)

■ Cost of bugs over time

- Longer a bug goes undetected, the more expensive it is
 - Bug found early has little cost
 - Finding a bug at chip/system has low/moderate cost
 - Could require new algorithm, which could effect schedule and cause board rework
 - Finding a bug in System Test requires new 'spin' of a chip
 - Requires more debug time and isolation time
 - Finding bug in customer's environment can cost hundreds of millions and worst of all - Reputation



Why HDL Verification? (cont)

- Thomas Nicely, a professor of mathematics, had written code to enumerate primes. Nicely noticed some inconsistencies in the calculations on June 13, 1994, shortly after adding a Pentium system to his group of computers. On October 24, 1994, he reported the issue to Intel.

$$4.195.835 / 3.145.727 = 1,333820449136241002$$

$$4.195.835 / 3.145.727 = 1,333739068902037589$$



- On January 17, 1995, Intel announced "a pre-tax charge of \$475 million against earnings, ostensibly the total cost associated with replacement of the flawed processors."

What is Verification?

What is Verification?

Verification is a **process** used to demonstrate the functional correctness of a design.

Verification Challenge

- How do we know that a design is correct?
- How do we know that the design behaves as expected?
- How do we know we have checked everything?
- How do we deal with size increases of designs faster than tools performance?
- How do we get correct Hardware for the first delivery?

Importance of Verification

- Most books focus on syntax, semantics and RTL subset
 - Given the amount of literature on writing synthesizable code vs.. writing verification testbenches, one would think that the former is a more daunting task. Experience proves otherwise.
- 70% of design effort goes to verification
 - Properly staffed design teams have dedicated verification engineers.
 - Verification Engineers usually outweigh designers 2-1
- 80% of all written code is in the verification environment

Importance of Verification

Verification is on critical path

What is being verified?

- Choosing a common origin and reconvergence points determines what is being verified and what type of method to use.
- Following types of verification all have different origin and reconvergence points:
 - Formal Verification
 - Model Checking
 - Functional Verification

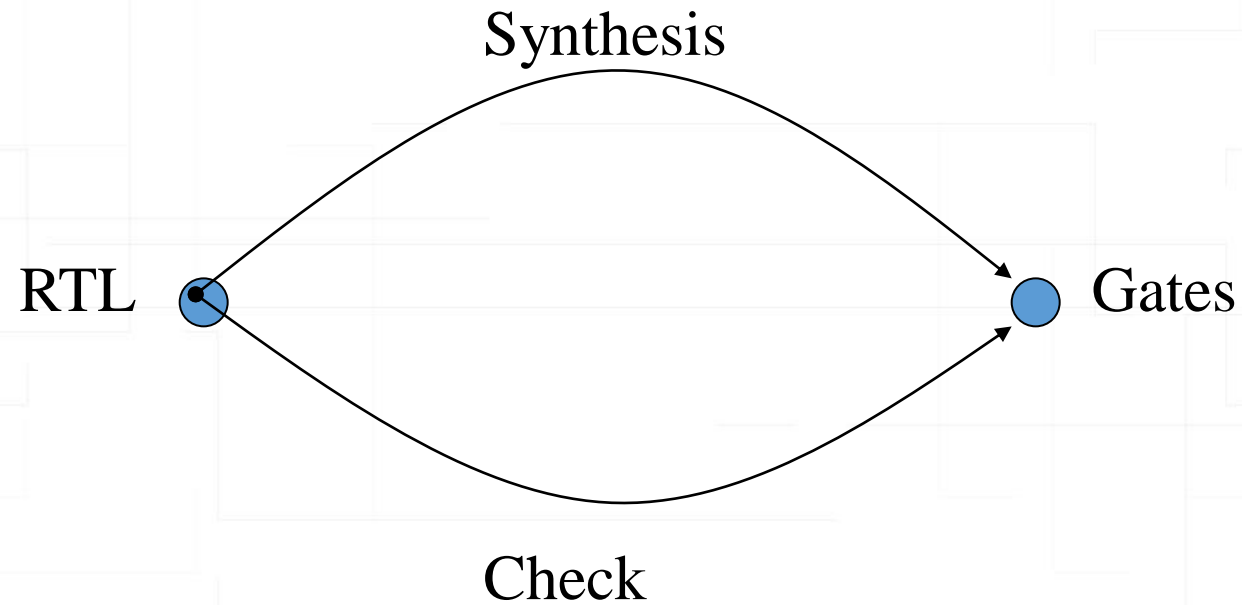
Formal Verification

- Once the end points of formal verification reconvergence paths are understood, then you know exactly what is being verified.
- 2 Types of Formal:
 - Equivalence
 - Model Checking

Equivalence Checking

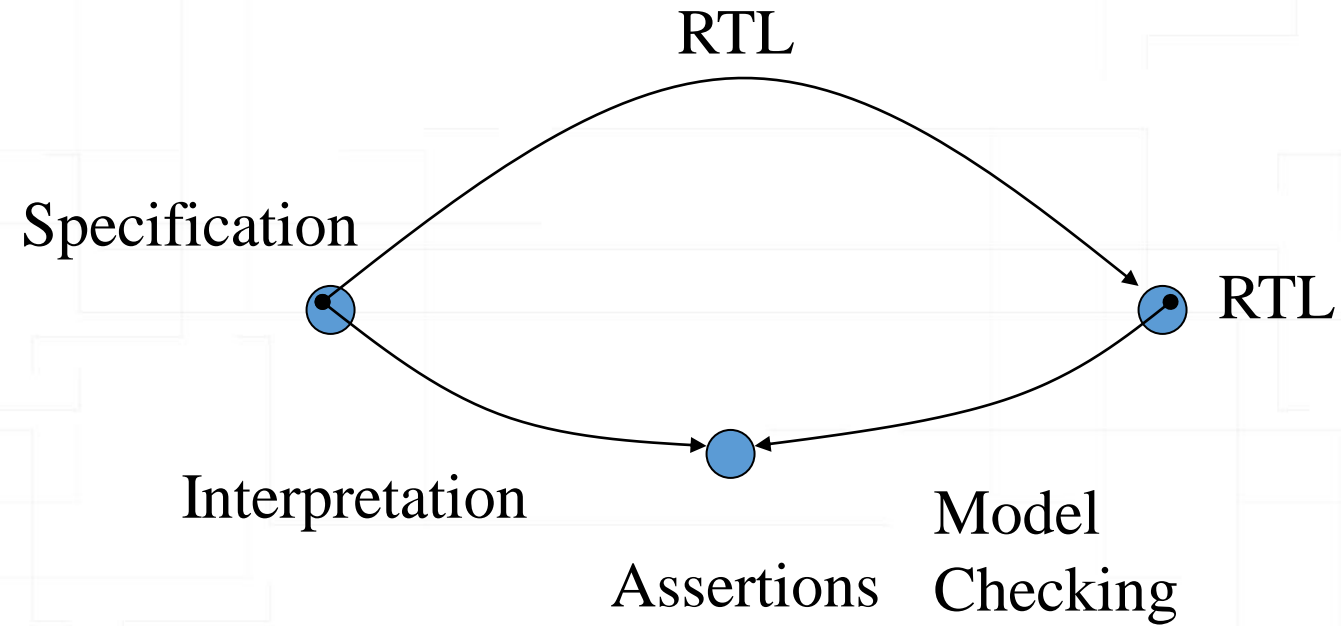
- Compares two models to see if equivalence
- Proves mathematically that the origin and output are logically equivalent
- Examples:
 - RTL to Gates (Post Synthesis)
 - Post Synthesis Gates to Post PD Gates

Equivalence Reconvergence Model



- Form of formal verification
- Characteristics of a design are formally proven or disproved
- Looks for generic problems or violations of user defined rules about the behavior of the design

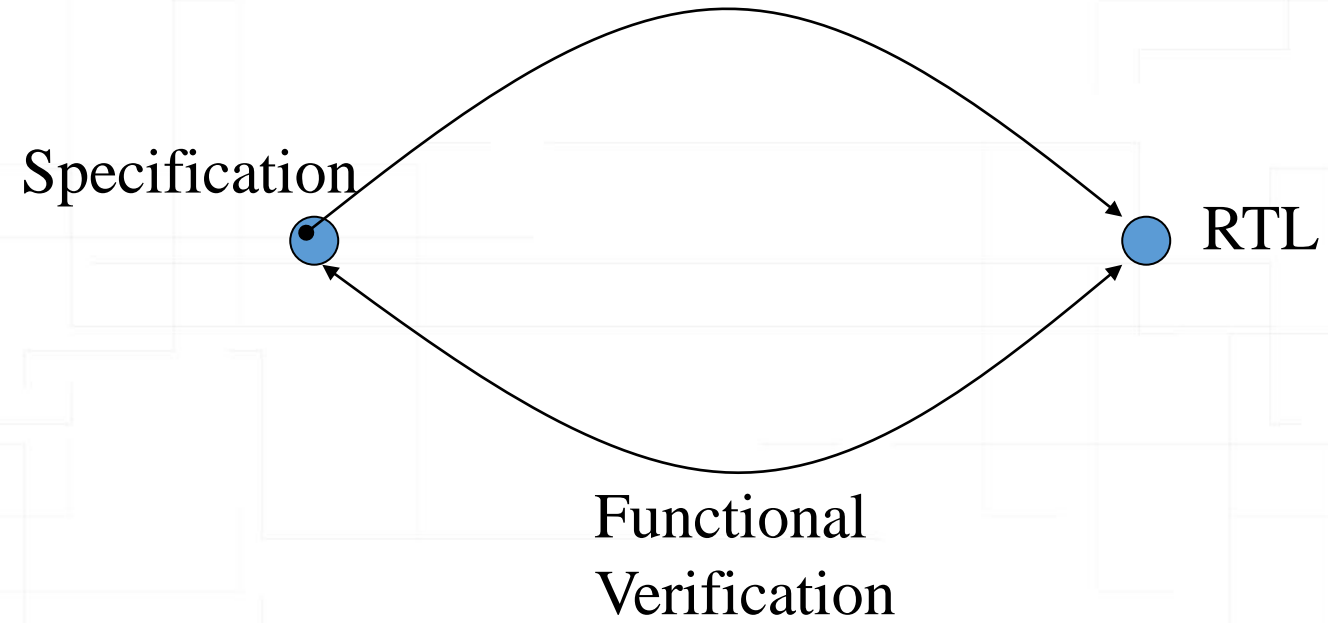
Model Checking Reconvergence Model



Functional Verification

- Verifies design intent
 - Without, one must trust that the transformation of a specification to RTL was performed correctly
- Prove presence of bugs, but cannot prove their absence

Functional Reconvergence Model



Functional Verification Approaches

- Black-Box Approach
- White-Box Approach
- Grey-Box Approach

Black-Box



- The black box has inputs, outputs, and performs some function.
- The function may be well documented...or not.
- To verify a black box, you need to understand the function and be able to predict the outputs based on the inputs.
- The black box can be a full system, a chip, a unit of a chip, or a single macro

- White box verification means that the internal facilities are visible and utilized by the testbench stimulus.
- Examples: Unit/Module level verification

Grey-Box

- Grey box verification means that a limited number of facilities are utilized in a mostly black-box environment.
- Example: Most environments! Prediction of correct results on the interface is occasionally impossible without viewing an internal signal.

Perfect Verification

- To fully verify a black-box, you must show that the logic works correctly for all combinations of inputs.

This entails:

- Driving all permutations on the input lines
- Checking for proper results in all cases
- Full verification is not practical on large pieces of designs, but the principles are valid across all verification.

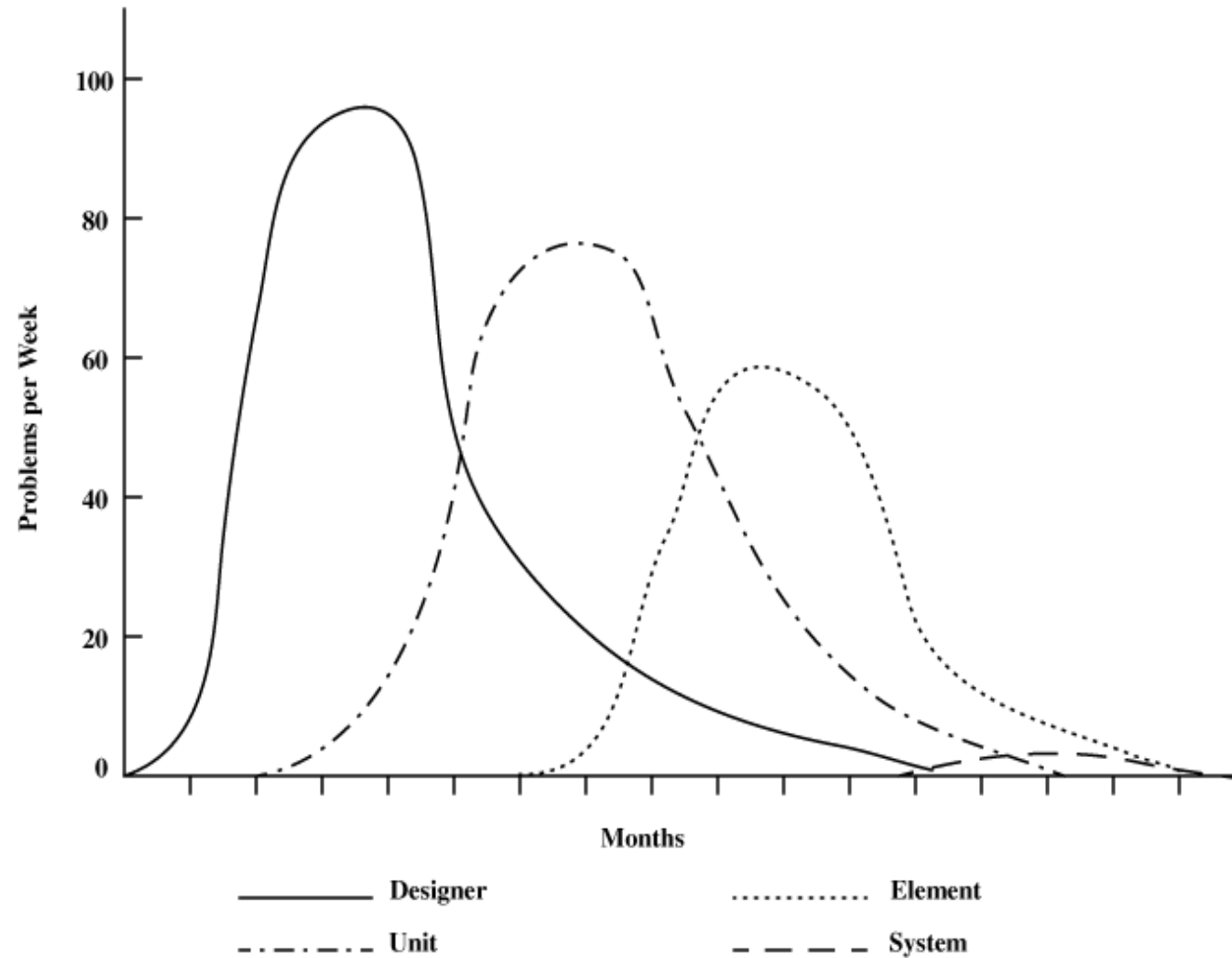
In an Ideal World....

- Every macro would have perfect verification performed
 - All permutations would be verified based on legal inputs
 - All outputs checked on the small chunks of the design
- Unit, chip, and system level would then only need to verify interconnections
 - Ensure that designers used correct Input/Output assumptions and protocols

Reality Check

- Macro verification across an entire system is not feasible for the business
 - There may be over 400 macros on a chip, which would require about 200 verification engineers!
 - That number of skilled verification engineers does not exist
 - The business can't support the development expense
- Verification Leaders must make reasonable trade-offs
 - Concentrate on Unit level
 - Designer level on riskiest macros

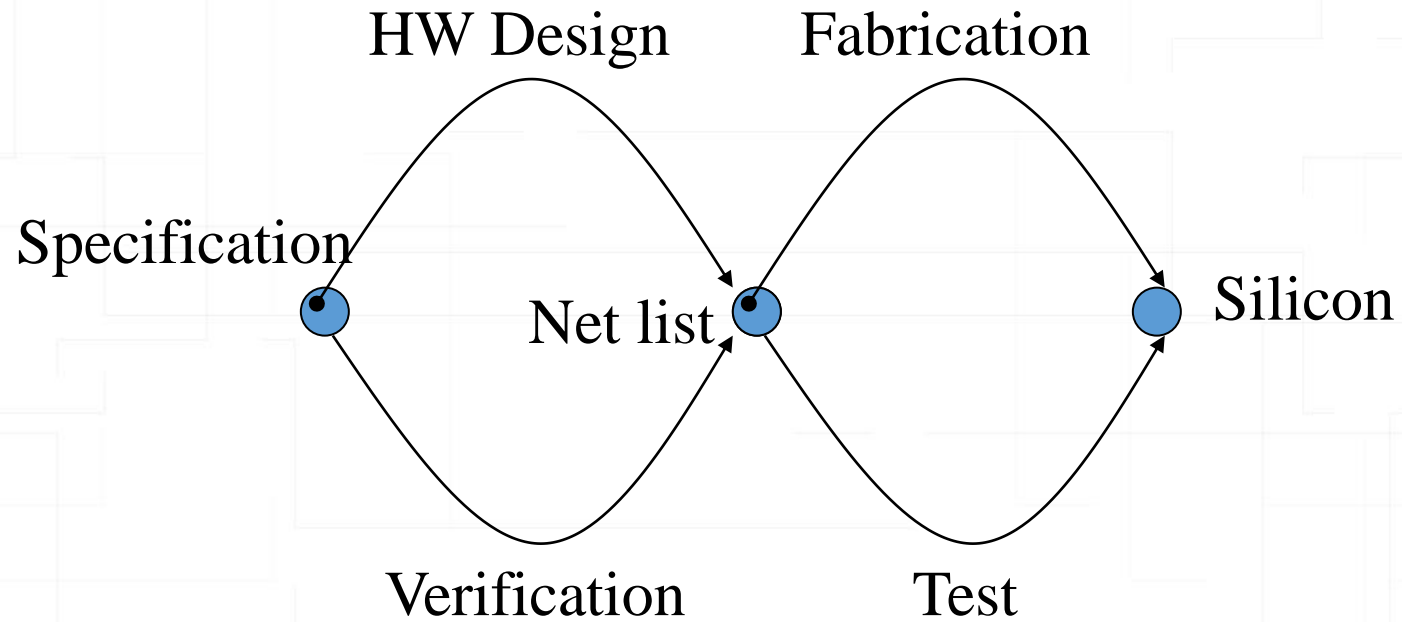
Typical Bug rates per level



Verification VS. Test

- Two often confused
- Purpose of *test* is to verify that the design was manufactured properly
- *Verification* is to ensure that the design meets the functionality **intent**

Verification and Test Reconvergence Model



Cost of Verification

- Necessary Evil

- Always takes too long and costs too much
- Verification does not generate revenue

- Yet indispensable

- To create revenue, design must be functionally correct and provide benefits to customer
- Proper functional verification demonstrates trustworthiness of the design

Ways to reduce verification time

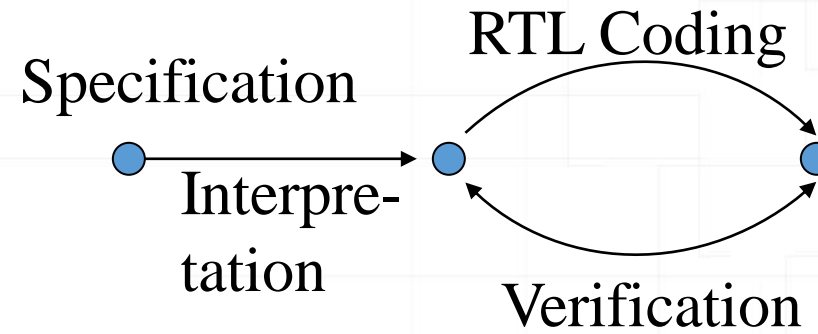
- Parallelism: Add more resources
- Abstraction: Higher level of abstraction (i.e. C vs.. Assembly)
 - Beware though – this means a reduction in control
- Automation: Tools to automate standard processes
 - Requires standard processes
 - Not all processes can be automated

When is Verification Done?

- Never truly done on complex designs
- Verification can only show presence of errors, not their absence
- Given enough time, errors will be uncovered
- Question – Is the error likely to be severe enough to warrant the effort spent to find the error?

Human Factor in Verification Process

- An individual (or group of individuals) must interpret specification and transform into correct function.



Ways to reduce human-introduced errors

- Automation
 - Take human intervention out of the process
- Poka-Yoka (TBD ????)
 - Make human intervention fool-proof
- Redundancy
 - Have two individuals (or groups) check each others work

Automation

- Obvious way to eliminate human-introduced errors – take the human out.
 - Good in concept
 - Reality dictates that this is not feasible
 - Processes are not defined well enough
 - Processes require human ingenuity and creativity

- Term coined in Total Quality Management circles
- Means to “mistake-proof” the human intervention
- Typically the last step in complete automation
- Same pitfalls as automation – verification remains an art, it does not yield itself to well-defined steps.

”

- Duplicate every transformation
 - Every transformation made by a human is either:
 - Verified by another individual
 - Two complete and separate transformations are performed with each outcome compared to verify that both produced the same or equivalent result
- Simplest
- Most costly, but still cheaper than redesign and replacement of a defective product
- Designer should NOT be in charge of verification

Testarea sistemelor de calcul

Course 3

Previous course summary

- Why HDL Verification?
- What is Verification?
- Verification Challenge
- Importance of Verification
- What is being verified?
 - Formal Verification
 - Equivalence Checking
 - Model Checking
 - Functional Verification

Previous course summary (cont)

- Verification VS. Test
- Cost of Verification
- When is Verification Done?
- Human Factor in Verification Process

Verification Tools

- Automation improves the efficiency and reliability of the verification process
- Some tools, such as a simulator, are essential. Others automate tedious tasks and increase confidence in the outcome.
- It is not necessary to use all the tools.

Verification Tools (cont)

- Linting Tools
- Simulators
- Third Party Models
- Waveform Viewers
- Verification Languages (Non-RTL)
- Revision Control
- Issue Tracking
- Metrics
- Code Coverage

Linting Tools

- Assist in finding common programming mistakes
- Only identify certain class of problems
 - Linting is a Static checker
 - Same deficiencies as in a 'C' linter
- Many false negatives are reported
- Does assist in enforcing coding guidelines

Simulators

- Simulate the design before fabrication
- Simulation requires stimulus – not static tools
- Require to reproduce environment in which design will be in – this is called a **testbench**
- Simulation outputs are validated externally against design intent (specification)
- Two types:
 - Event based
 - Cycle based

Event Based Simulators

- Event based simulators are driven based on events
- Outputs are a function of inputs
 - The outputs change only when the inputs do
 - The event is the input changing
 - This event causes simulator to re-evaluate and calculate new output

Cycle Based Simulators

- Simulation is based on clock-cycles not events
- Cycle based simulators contain no timing information
- Can handle only synchronous circuits
 - Only 'event' is active edge of clock
 - All other inputs are aligned with clock
- Much faster than event based

Co-Simulation

- Co-simulators are combination of event, cycle, and other simulators (acceleration, emulation)
- Performance is decreased due to inter tool communication.
- Ambiguities arise during translation from one simulator to the other.

Third Party Models

- Many designs use off the shelf parts
- To verify such a design, must obtain a model to these parts
- Often must get the model from a 3rd party
- Most 3rd party models are provided as compiled binary models
- Why buy 3rd party models?
 - Engineering resources
 - Quality (especially in the area of system timing)

Waveform Viewers

- Some consider this as part of the simulator
- Most common verification tools used
- Purpose is to visually inspect design/testbench/verification environment
- Things to consider:
 - Don't use viewer to determine if design passes or fails a test – use to debug
 - Recording waves causes a performance hit on the simulator

Verification Languages

- Specific to verification principles
- Deficiencies in RTL languages (Verilog and VHDL)
 - Verilog was designed with a focus on describing low-level hardware structures
 - No support for data structures (records, linked lists, etc)
 - Not object oriented
 - VHDL was designed for large design teams
 - Encapsulates all information and communicates strictly through well-defined interfaces
- These limitations get in the way of efficient implementation of a verification strategy

Verification Languages (cont)

- Some examples of verification languages
 - Specman Elite (Cadence)
 - System Verilog
 - System C

Revision Control

- Why is revision control considered a verification tool?
- Simply put – to ensure that what is being verified is actually what is being implemented
- Concepts for revision control:
 - Files must be centrally managed
 - Must be easy to get the files
 - History is kept for each file
 - Team owns the files, no one individual (makes editing files when someone is out (sick, vacation, etc) easy.
- HDL design/verification is similar to managing a software project. Let's leverage their experience

File Tags

- Each file tag has a specific meaning

Table 2-1.
Example tags
for release
management

Tag Name	Description
Submit	Ready to submit to functional verification. Author has verified syntax correctness and basic level of functionality.
Bronze	Passes a basic set of functional testcases. Release is sufficiently functional for integration.
Silver	Passes all functional testcases.
Gold	Passes all functional testcases and meets coding coverage guidelines (requires additional corner-case testcases).
To_Synthesis	Ready to submit to synthesis. Usually matches "Silver" or "Gold".
To_Layout	Ready to submit to layout. Usually matches "Gold".
Version_M.N	Version that was manufactured. Matches corresponding "To_Layout" release. Future versions of the same chip will move tags beyond this point.
ON_YYMMDD	Some meaningful release on the specified date.

Issue Tracking

- Another tool not considered a verification tool.
- Face-it – verification engineer's job is to find bugs. We enjoy pointing out a designers fault (keeps their ego's in check).
- Issue tracking is used to deal with the bugs that are found – bugs must be fixed
- Two things to consider
 - What is an issue?
 - How to track it?

What is an issue?

- First note that the cost of tracking an issue should not be greater than the cost of the issue itself!
- An issue is anything that can effect the functionality of the design!
 - Bugs found during execution of a testbench
 - Ambiguities or incompleteness of a specification
 - Architectural decisions/trade-offs
 - Errors found at all stages of the design (in the design or the verification environment)
 - New thought relevant testcases that are not part of the plan.

How to track an issue?

- **Some methods:**
 - Grapevine System
 - Procedural System
 - Computerized System

Grapevine

- This is the simplest tracking method
- What is it exactly – word of mouth! You find an issue, you walk over to your co-worker and tell him/her.
- Pros:
 - Individuals are empowered!
 - Simple issues are solved almost immediately
- Cons:
 - There is no history on the problem, history will repeat itself
 - Same issue may be revisited numerous times (no history)
 - Can lead to “finger pointing”

Procedural

- Next step in evolution – involve managers or team leaders
- Issues are formally reported via e-mail or something similar.
- Outstanding issues are reviewed and resolved during status/team meetings.
- Cons: Consumes inordinate amount of time, therefore usually only most critical issues are tracked. The smaller issues (more numerous) are resorted back to the Post-It or grapevine method.

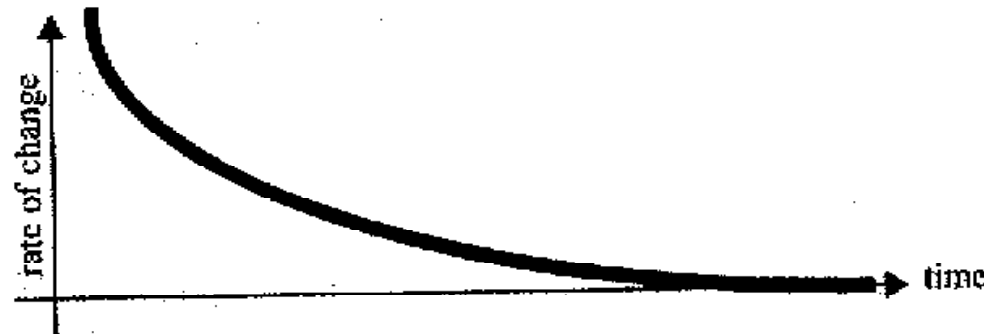
Computerized

- Top of the food chain – using a computer system (some database usually) to track issues
 - Issues are seen through to a resolution
 - Issues can be assigned and/or reassigned to individuals or small teams
 - E-mail notification can be automatic
 - More structure to the tracking of issues
 - Contains a history
 - Provides a ‘lessons-learned’ database for others
 - Can send daily/weekly status of top issues (automates the procedural system)

Metrics

- Not really a verification tool, but managers love metrics and measurements!
- They have little time to personally assess progress, so they want something measurable.
- Code coverage.

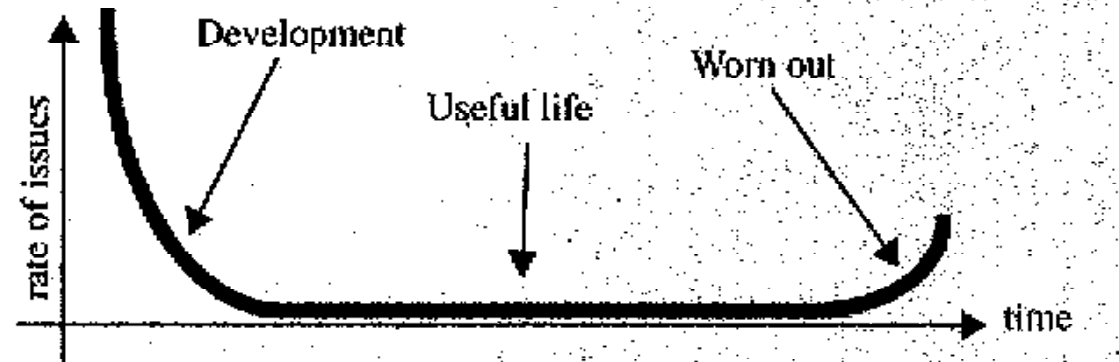
Figure 2-16.
Ideal code
change rate
metric over
time



Quality Related Metrics

- Quality is subjective
- Examples of quality metrics
 - Number of known outstanding issues
 - Number of bugs found during service life
- Must be very careful to interpret and use any metric correctly!!!

Figure 2-17.
Number of
outstanding
issues
throughout the
life cycle of a
design



Metrics (cont)

- Others can include:
 - Number of lines of code
 - Number of outstanding issues
 - Code Coverage Metrics - how thoroughly does verification suite exercise code
 - Number of Lines of Code Needed for Verification Suite - a measure of the level of effort needed
 - Ratio of Lines of Verification Code to Lines of Code in the Model - measure of design complexity
 - Number of source code changes over time

Code Coverage

- Method used by software for some time.
- Main problem is that false positives answers can look identical to true positive answers
- Code coverage can answer the question:
 - Is there a function or combination of functions that have not been verified?
- Can run coverage on the testbench to indicate what areas of the models are executing most. This is known as a profiler – It gives insight on what to optimize
- Many types of report metrics

Report Metrics for Code Coverage

- **Statement (block):**
 - Measures which lines (statements have been executed) by the verification suite
- **Branch (path):**
 - Measures all possible ways to execute a sequence of instructions
- **Expression Coverage:**
 - Measures the various ways paths through the code are executed

Report Metrics for Code Coverage(cont)

- **Toggle Coverage:**

- Toggle coverage gives a report that how many times signals and ports are toggled during a simulation run. It also measures activity in the design, such as unused signals or signals that remain constant or less value changes.

- **State/FSM Coverage:**

- FSM coverage reports, whether the simulation run could reach all of the states and cover all possible transitions in a given state machine.

Statements(Blocks)

- Statement coverage can also be called block coverage
- The EDA simulators can show how many times a statement was executed
- Also need to insure that executed statements are simulated with different values
- And there is code that was not meant to be simulated (code specifically for synthesis for example)

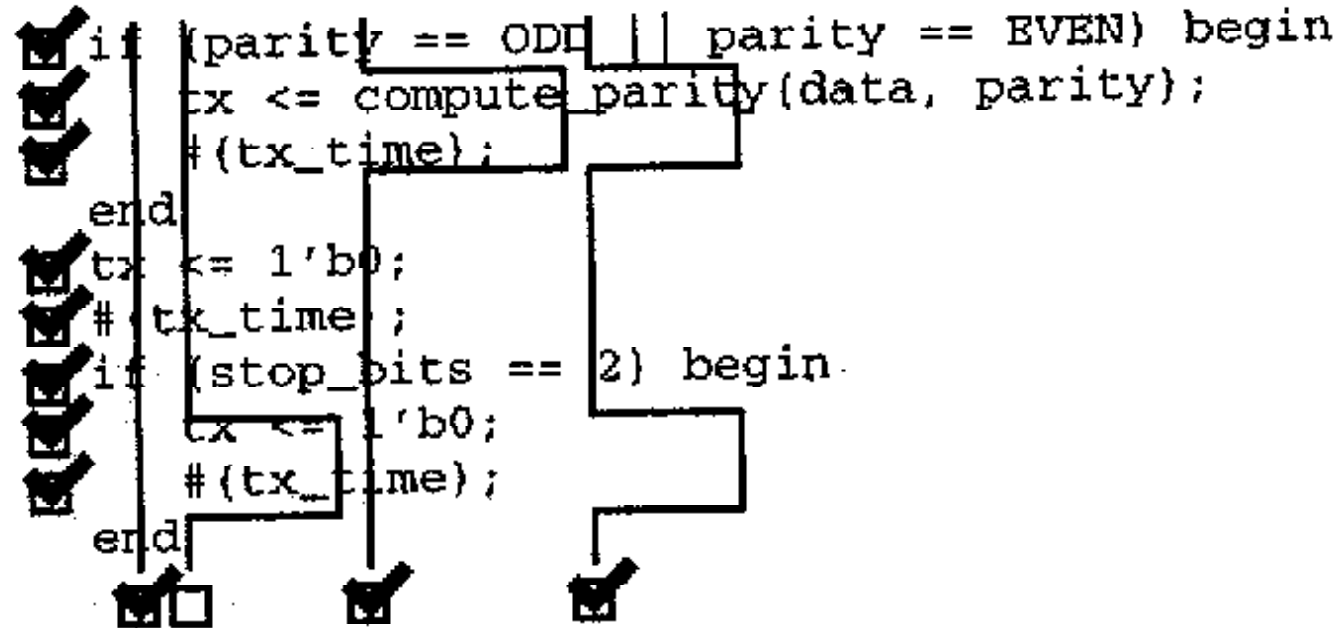
Sample 2-10.
Block vs.
statement exe-
cution

```
if (dtack == 1'b1) begin: acked
    as      <= 1'b0;
    data    <= 16'hZZZZ;
    bus_rq  <= 1'b0;
    state   <= IDLE;
end
```

Branch(Path) Coverage

- Measures all possible ways you can execute a sequence of statements
- Example has four possible paths

Sample 2-13.
Example of
statement and
path coverage



Branch Coverage Goal

- Desire is to take all possible paths through code
- It is possible to have 100% statement coverage but less than 100% path coverage
- Number of possible paths can be very, very large => keep number of paths as small as possible
- Obtaining 100% path coverage for a model of even moderate complexity is very difficult (but not impossible)

Expression Coverage

- A measure of the various ways paths through code are taken
- Example has 100% statement coverage but only 50% expression coverage

Sample 2-14.
Example of
statement and
expression
coverage

```
☒ if (parity == ODD || parity == EVEN) begin
☒   tx <= compute_parity(data, parity);
☒   #(tx_time);
☒ end
☒ tx <= 1'b0;
☒ #(tx_time);
☒ if (stop_bits == 2) begin
☒   tx <= 1'b0;
☒   #(tx_time);
☒ end
☐
```

100% Code Coverage

- What do 100% path and 100% expression coverage mean
 - Not much!! Just indicates how thoroughly verification suite exercises code. Does not indicate the quality of the verification suite.
 - Does not provide an indication about correctness of code
- Results from coverage can help identify corner cases not exercised
- Is an additional indicator for completeness of job
 - Code coverage value can indicate if job is **not** complete

Testarea sistemelor de calcul

Course 6

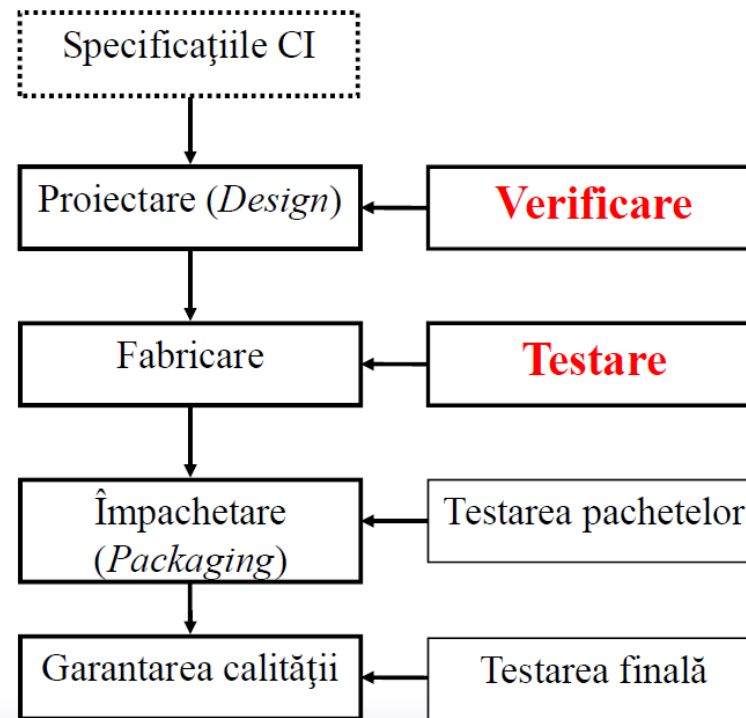
Testarea în Contextul Realizării unui CI

- ❑ Verificarea circuitului detectează erorile de proiectare

- Corecțiile sunt făcute înainte de fabricare

- ❑ Testarea circuitului detectează erorile de fabricație

- Un defect reprezintă o imperfecțiune fizică care generează funcționare incorectă



- Vezi curs 1



JTAG

JTAG stands from: Joint (European) Test
Access Group



JTAG

- Imagine you are handed this device and asked to get root on it as quickly as possible.
- No further information is given. Where would you begin?



Intro

Since you have the device in your hands, you might try directly attacking the hardware. However, if you've never done any kind of hardware hacking, getting started can be intimidating. In this course, we are going to talk about the fundamental information you need to know to use JTAG. We'll also go over a quick example to illustrate the power of direct hardware access.

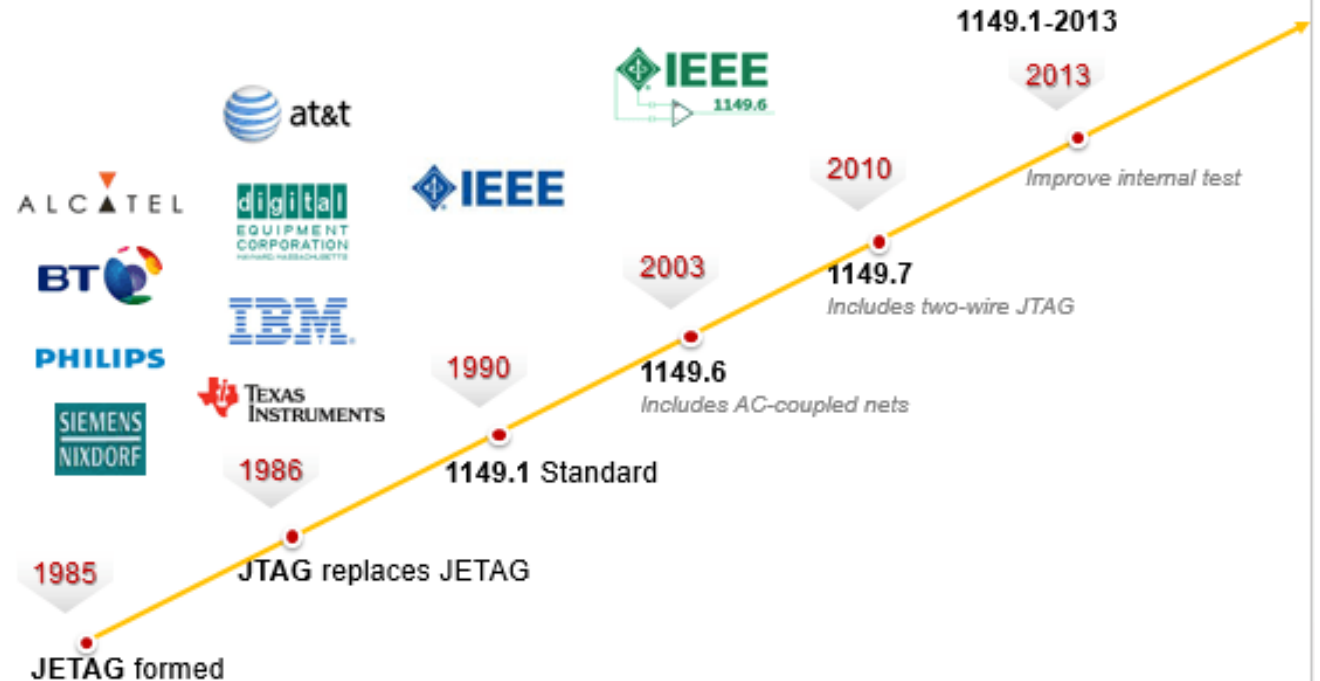
Why Do Manufacturers Use JTAG?

- JTAG is a common hardware interface that provides your computer with a way to communicate directly with the chips on a board.
- It was originally developed by a consortium, the Joint (European) Test Access Group, in the mid-80s to address the increasing difficulty of testing printed circuit boards (PCBs).

History of JTAG

- JTAG has been in widespread use ever since it was included in the Intel 80486 processor in 1990 and codified as IEEE 1491 that same year.
- Today JTAG is used for debugging, programming and testing on virtually ALL embedded devices.

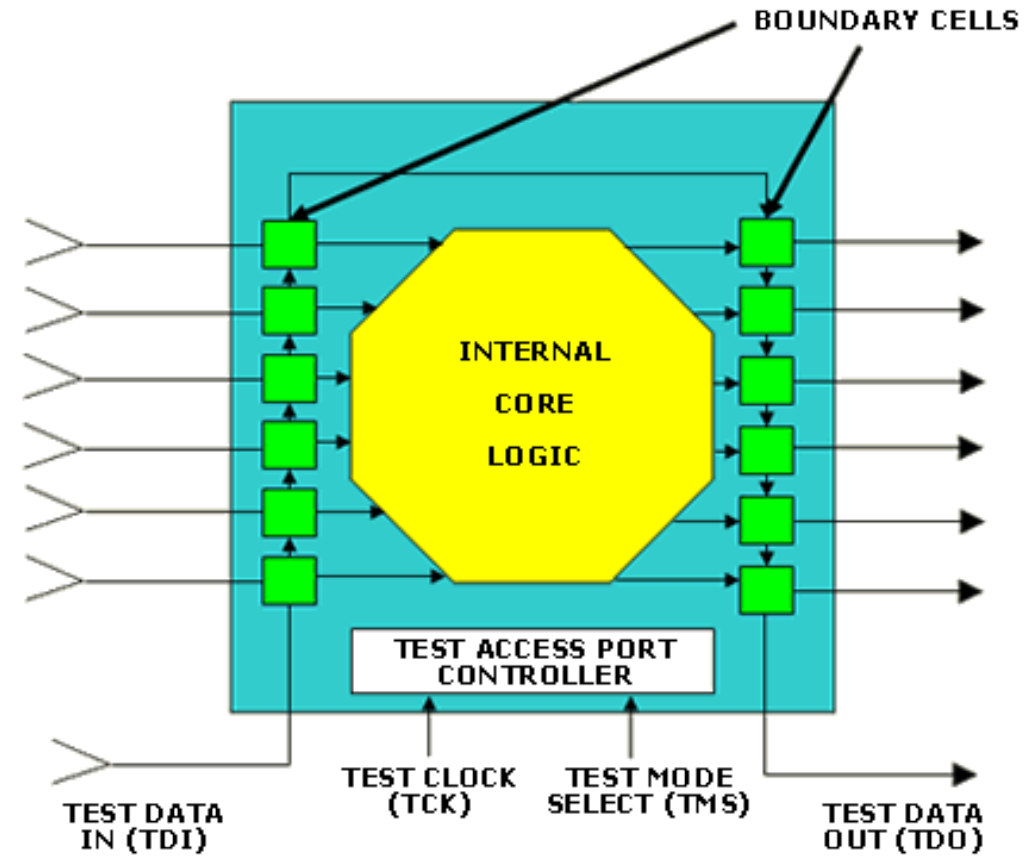
History of JTAG / Boundary Scan (IEEE1149.x)




JTAG/Boundary Scanning


With the invention of integrated circuits came the need to test physical interconnects. That is: test whether "Point A" on a circuit board correctly connects to "Point C".

As industry drove toward higher density interconnects and finer pitch (i.e. smaller) components, a standardized way of testing integrated circuits (or "chips") was needed.



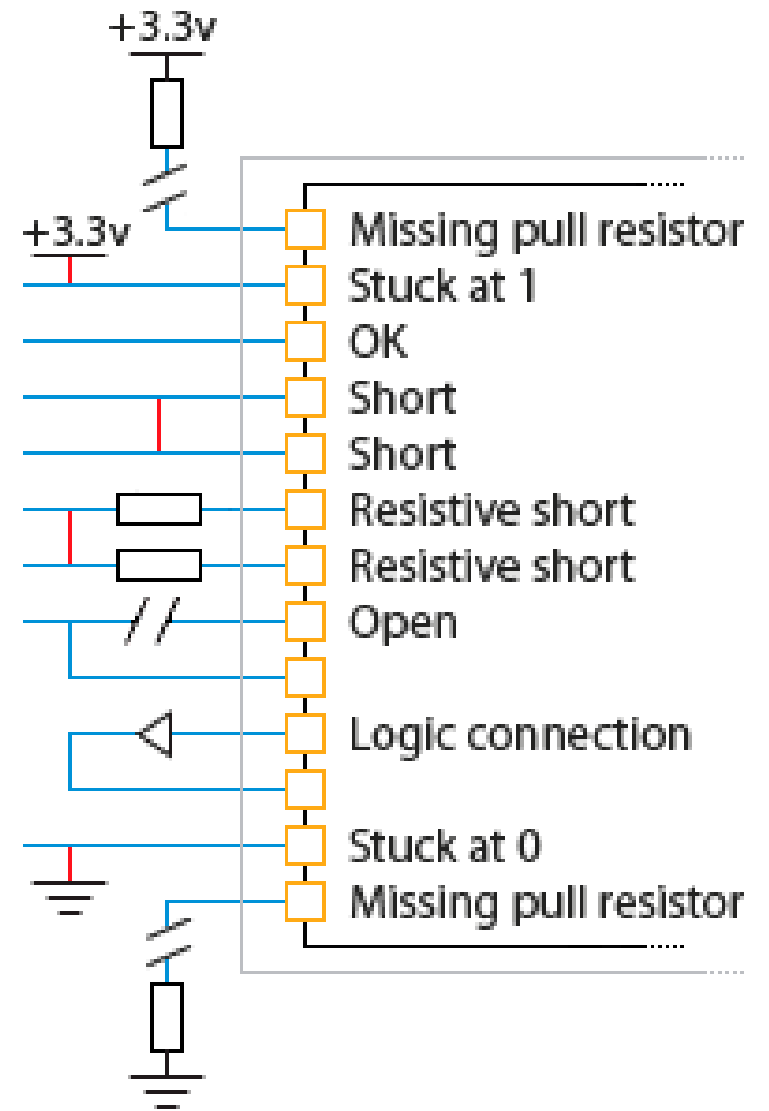


JTAG/Boundary Scanning

- By embedding the test logic within the IC itself and limiting the physical interface to just a few signals, JTAG/boundary-scan presented an elegant solution to testing, debugging, and diagnosing modern electronic systems.
 - Simply stated, boundary Scanning allows you to read and write individual bits, on individual pins, of the individual chips on a board. This ability to read and write 1's and 0's directly onto the pins is extremely powerful, but widely misunderstood.
- 

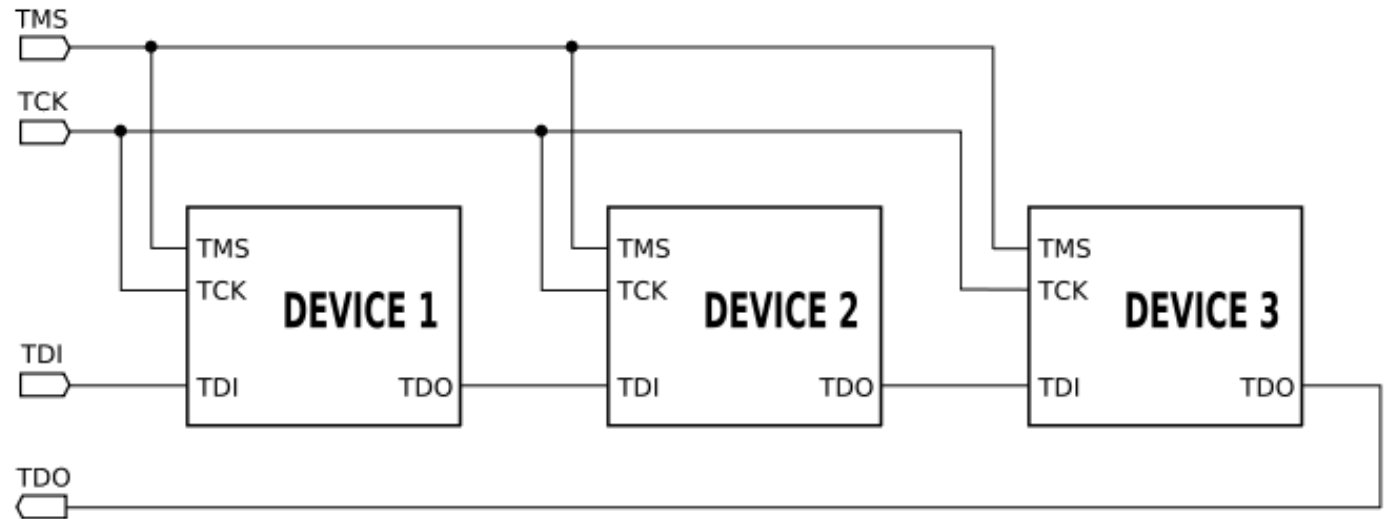
JTAG/Boundary Scanning

- The JTAG interface gives manufacturers a way to test the physical connections between pins on a chip.
- When electrical engineers talk about using JTAG to "debug" a chip, they are talking about something very different than traditional software debugging. They are talking about making sure pin A on chip A is physically connected to pin B on chip B, and that all those pins are functioning correctly.
- Since JTAG gives you direct hardware access to a device, it's also a fantastic tool for security research.



JTAG Fundamentals

- Before we go into using JTAG to actually hack something, let's review the JTAG fundamentals.
- 4 Pins, Plus 1. The official JTAG standard requires 4 standard pins (or signals), and defines an optional 5th.
- These signals, and the small bit of silicon logic that connects and controls them, are collectively referred to as the Test Access Port, or TAP controller.



TCK: Test Clock

The drummer, that dictates the speed of the TAP controller.

Voltage on this pin simply pulses up and down in a rhythmic, steady beat.

On every “beat” of the clock, the TAP controller takes a single action.

The actual clock speed is not specified in the JTAG standard.

The TAP controller accepts its speed from the outside device controlling JTAG

TMS: Test Mode Select

Voltages on the Mode Select pin control what action JTAG takes.



By manipulating the voltage 1 (voltage) or a 0 (no voltage) on this pin, you tell JTAG what you want it to do.

TDI: Test Data-In

The pin that feeds data into the chip.



The JTAG standard does not define protocols for communication over this pin. That is left up to the manufacturer.

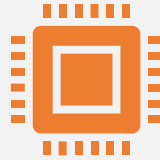


As far as JTAG is concerned, this pin is simply an ingress method for 1s and 0s to get into the chip.



What the chip does with them is irrelevant to JTAG.

TDO: Test Data-Out



The pin for data coming out of the chip.



Like the Data-In pin, communication protocols are not defined by JTAG.

TRST: Test Reset (Optional)

This optional signal is used to reset JTAG to a known good state



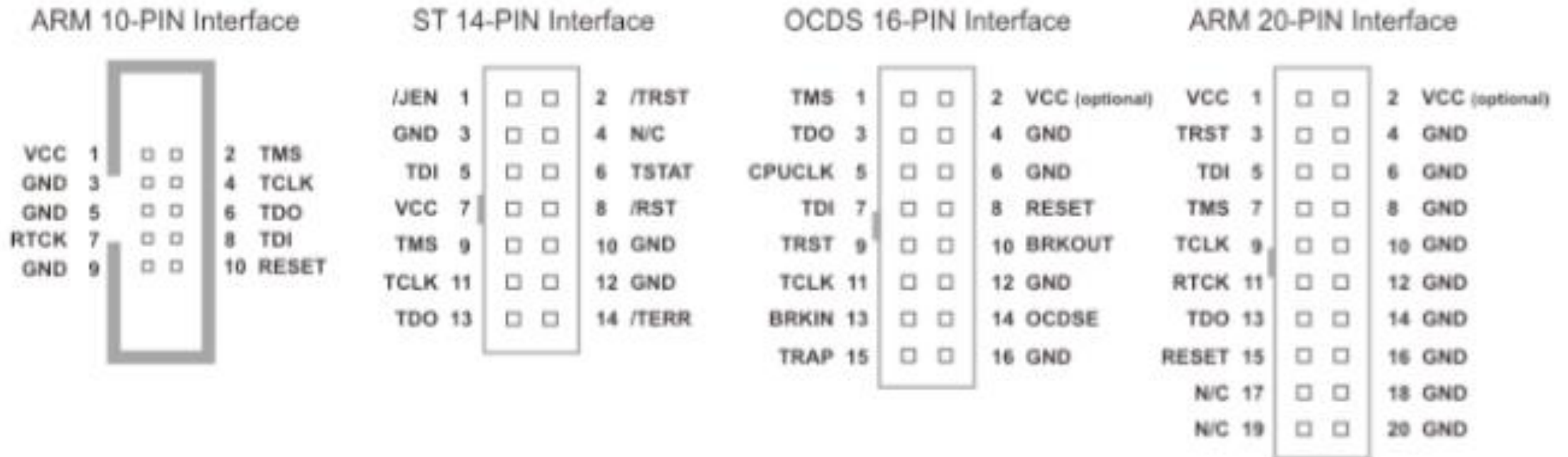
TMS is held at a 1 for five clock cycles in a row, you will wind up in the "Test Logic - Reset" state. If the TRST pin is omitted, this method can be used to reset the TAP controller to a known good state. Hence why TRST is "optional"

What JTAG ISN'T!

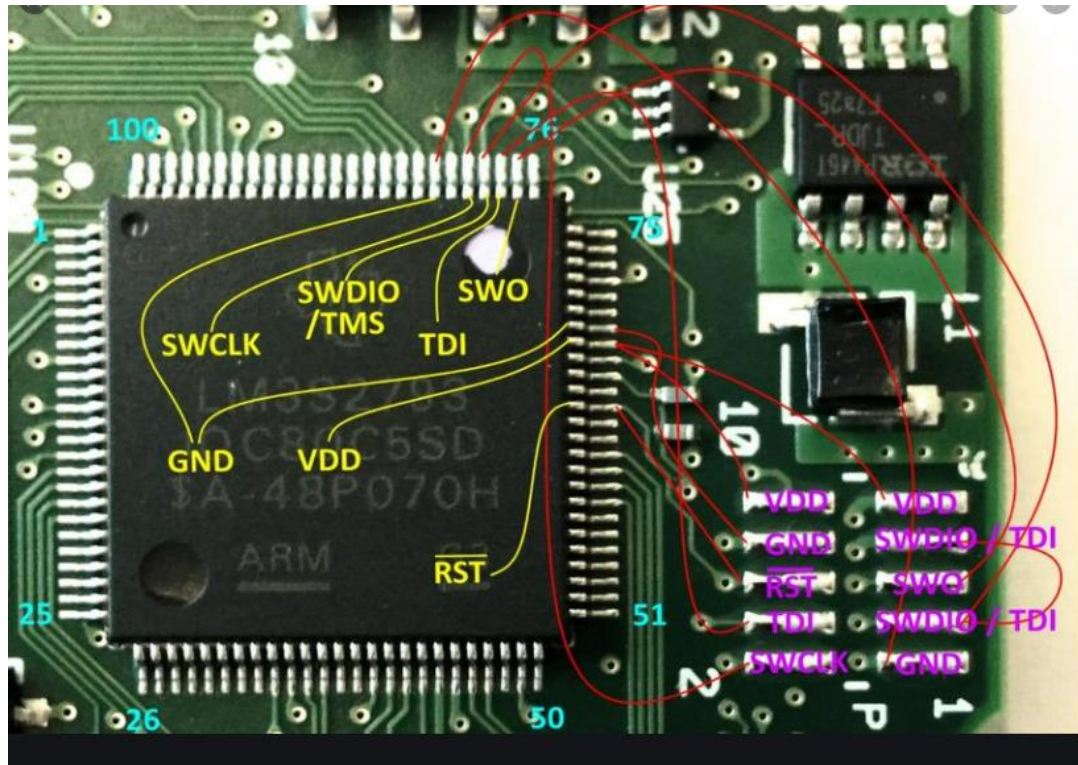
In an effort to dispel disinformation (which is so rampant) here is what JTAG isn't:

- The name of a specific device for debugging firmware or hardware
- The name of a piece of software used for debugging software code running on an embedded device.
- The name of a protocol: The JTAG specification doesn't even define the protocol for how data is passed between debugger and debugee (i.e. there is no .h file you can reference defining a "JTAG packet" to be passed over a serial, parallel, or USB cable to a target)

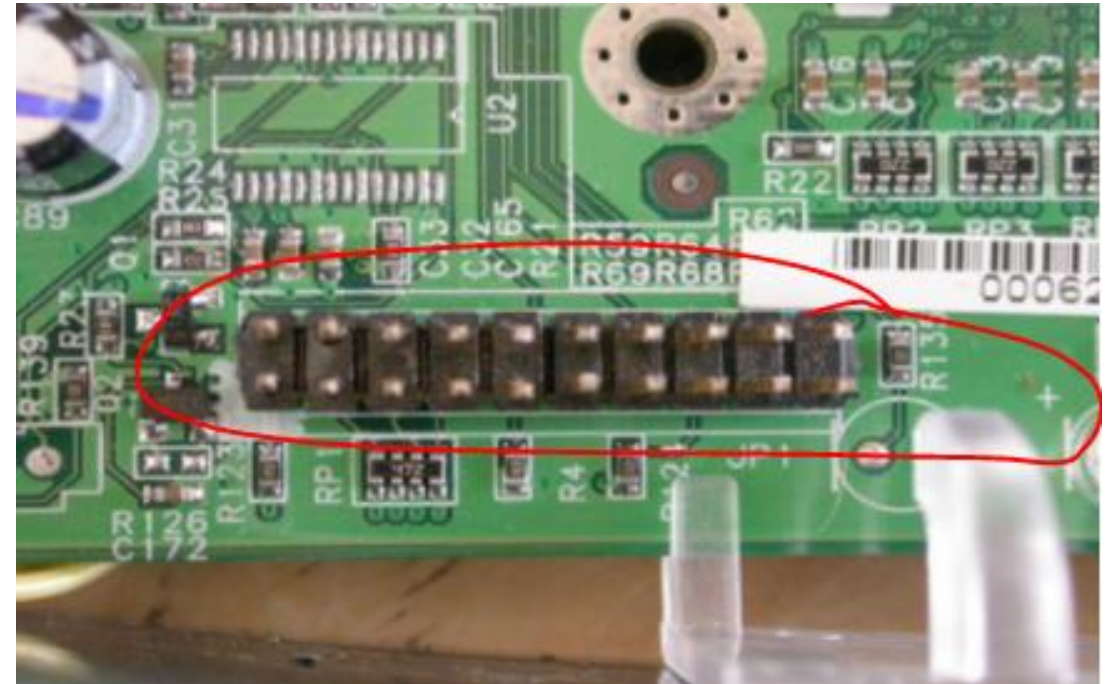
- The name of a kind of connector: Unlike a specification like USB or Ethernet, the JTAG specification doesn't even define what connectors should look like. This leads to a VERY confusing diversity of "JTAG" connectors. This is often a huge source of confusion to a software person. Imagine if every USB cable was different!



LM3S2793 JTAG Pins and Their Connections to the 10-pin ARM-JTAG Connector



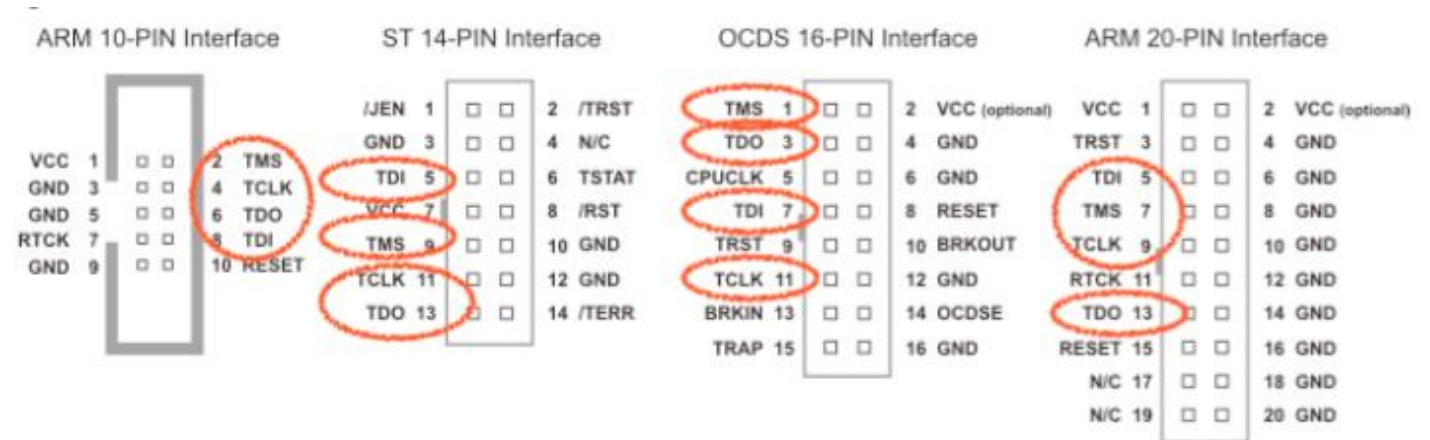
JTAG connector for a Point-Of-Sale device

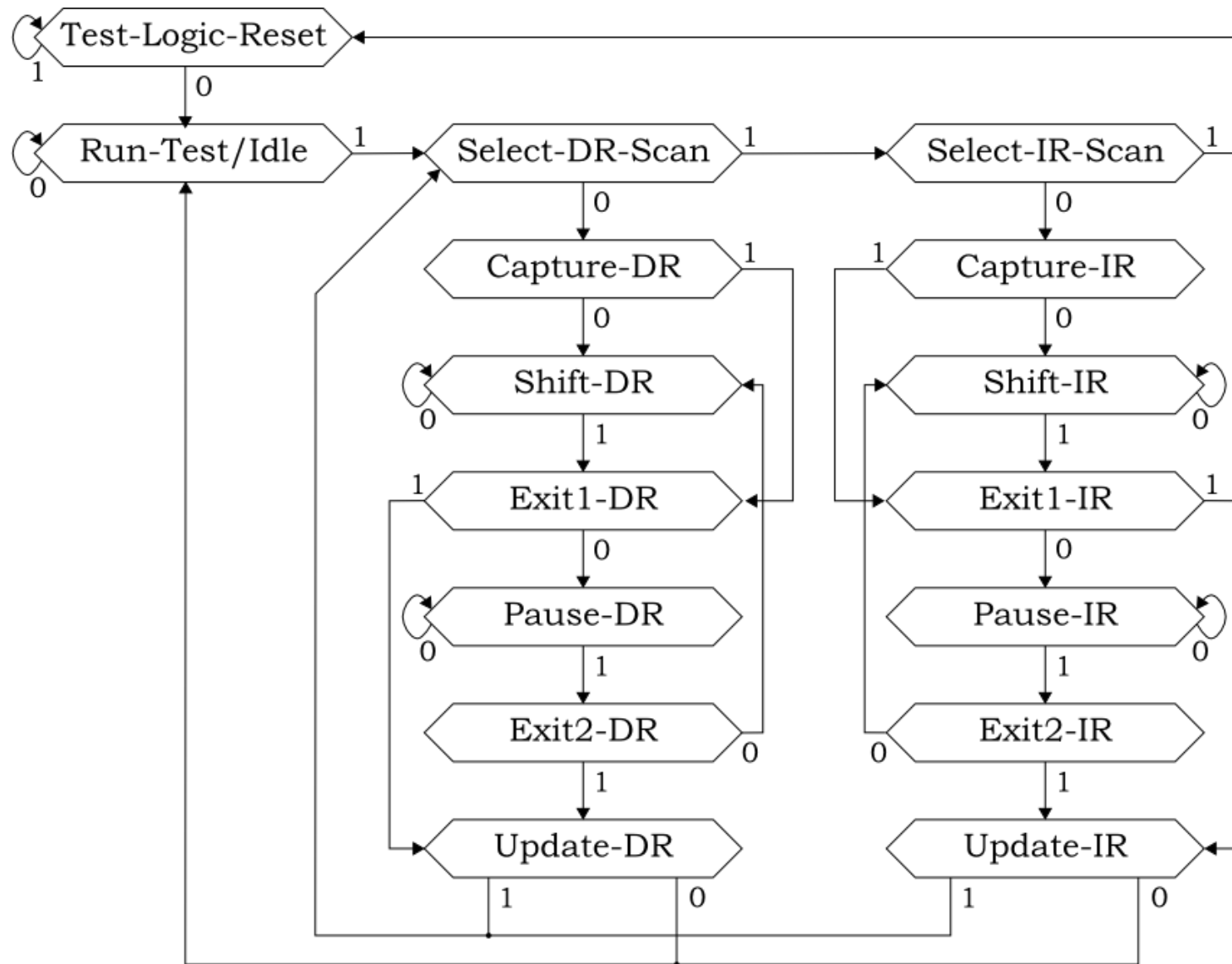


These
connectors
have JTAG
related
signals!

We now see that what **makes these connectors "JTAG" is not their pin configurations**, but rather the mere presence of these JTAG signals.

It is with these signals that we can manipulate and communicate with the State Machine of a target device via it's "Test Access Port" (or TAP) controller embedded in its silicon.



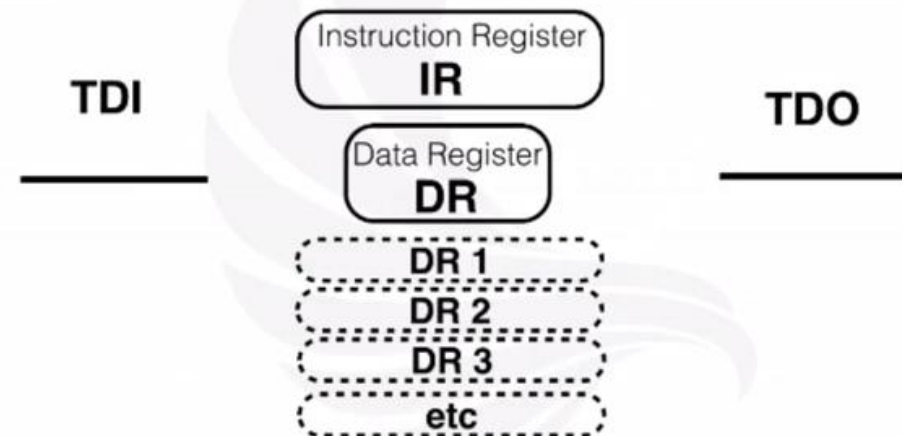
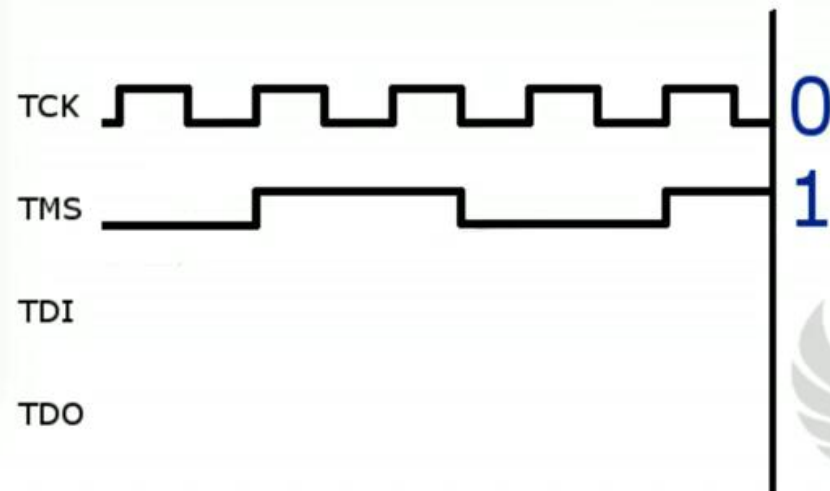
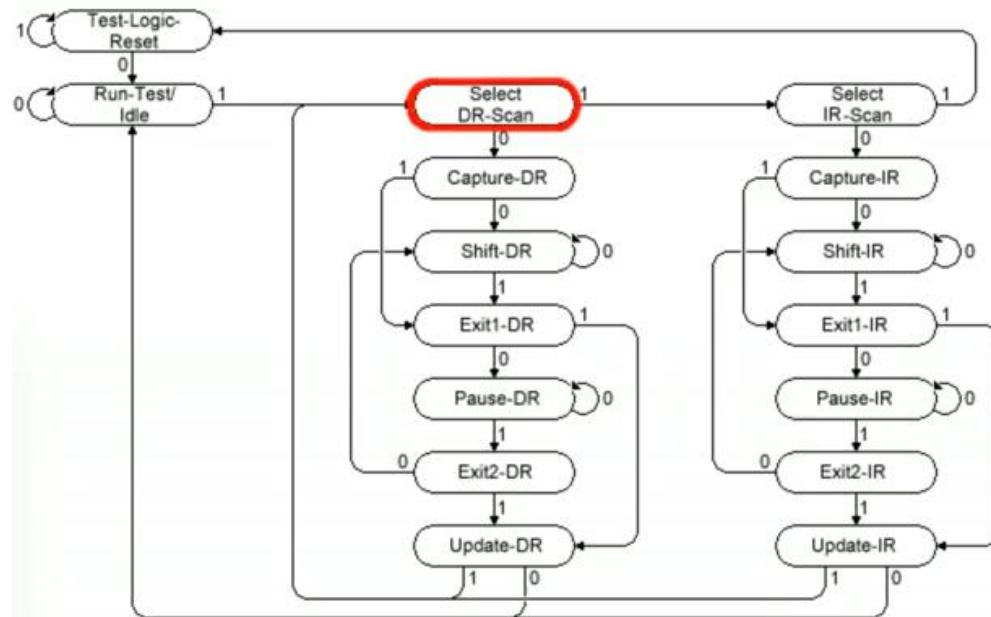


The JTAG State Machine



JTAG State Machine

- The JTAG specification IEEE 1149.1 specifies not only the signals but how the "State Machine" embedded in the TAP Controller of a device must behave
- By manipulating the voltage on the TMS pin of the TAP controller and the TCK pin, you control the way the State Machine changes state.
- On each beat of the clock, the JTAG TAP controller in your targeted device checks if there is voltage on the TMS pin, and interprets that as either a 1 (voltage) or a 0 (no voltage).
- This series of 1's and 0's moves JTAG through a relatively simple State Machine.
- The value of the TMS pin needed to change state is denoted with a tiny "1" or "0" in the state machine diagram. You can see the operation of the state machine in the animation below

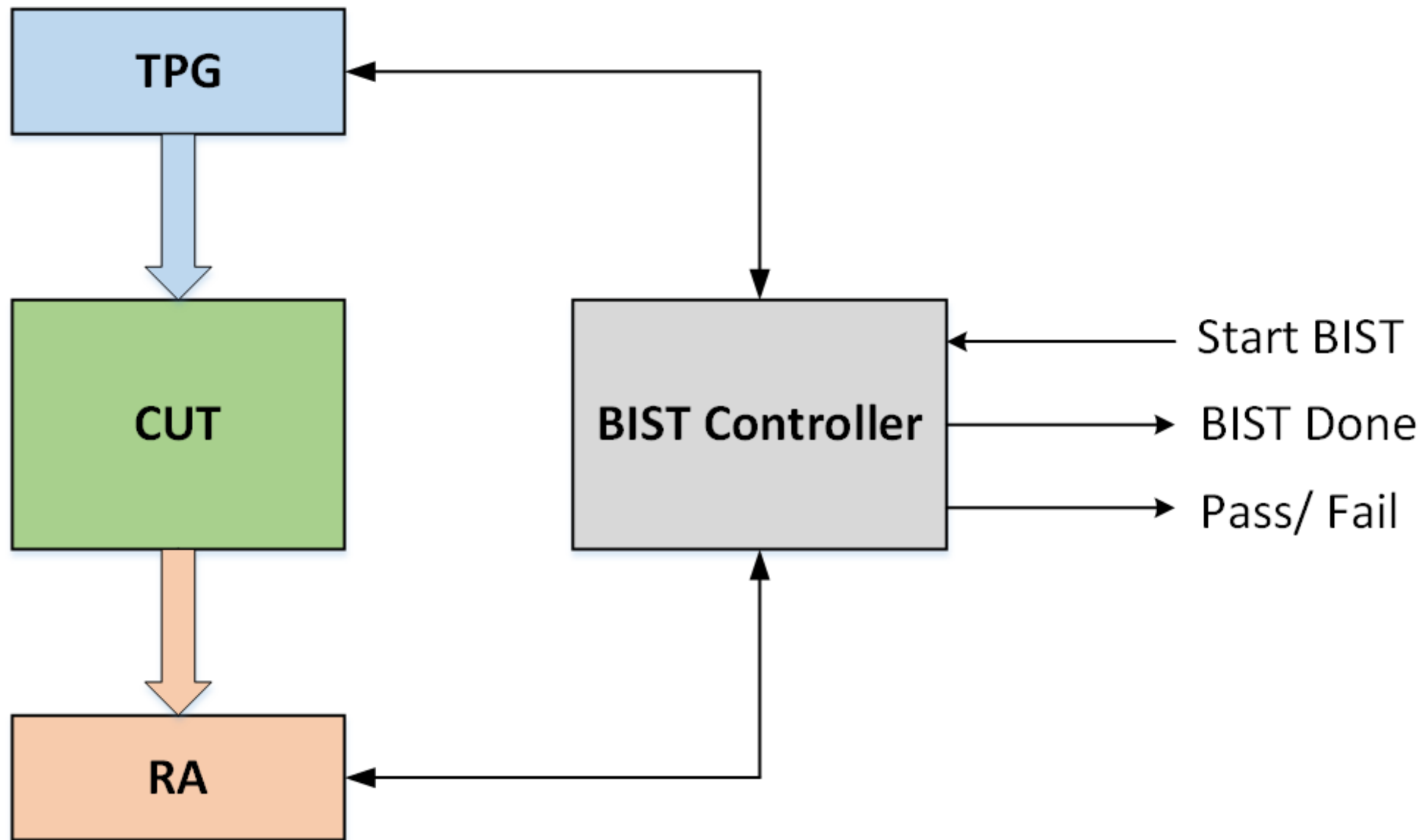


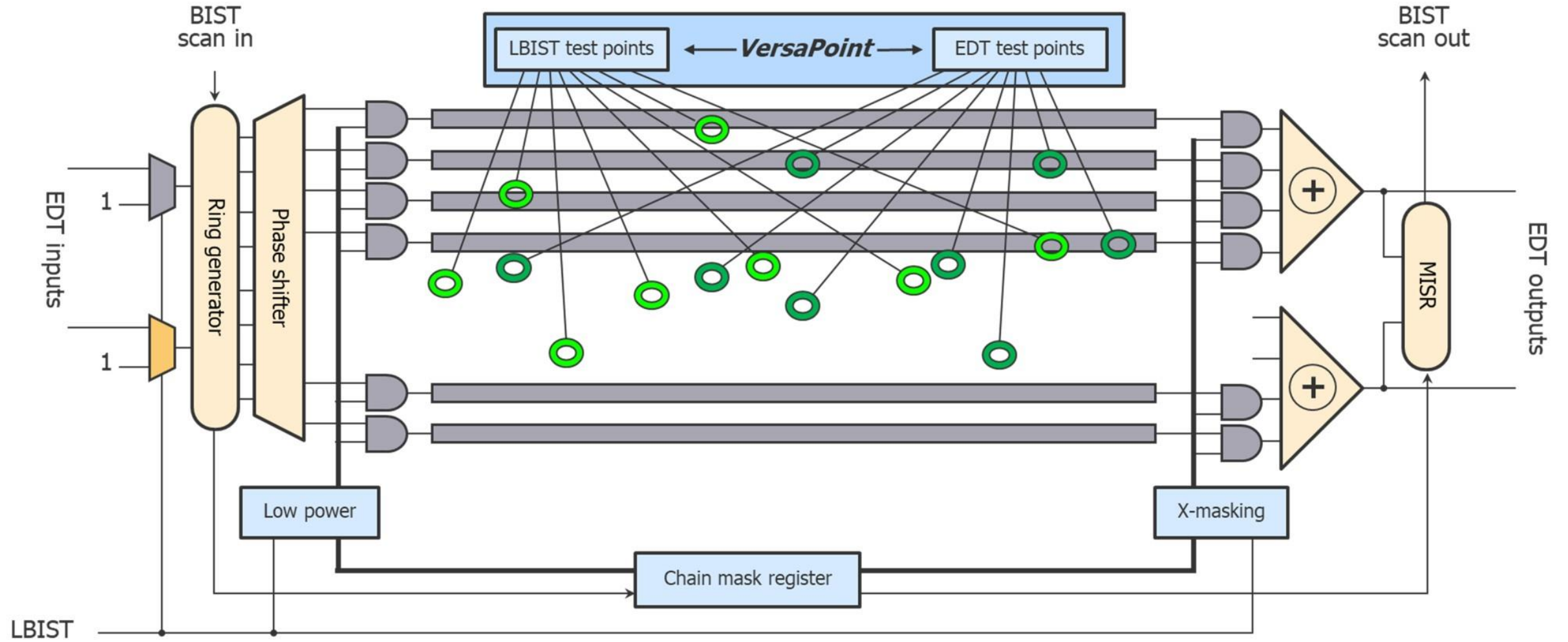
JTAG State Machine

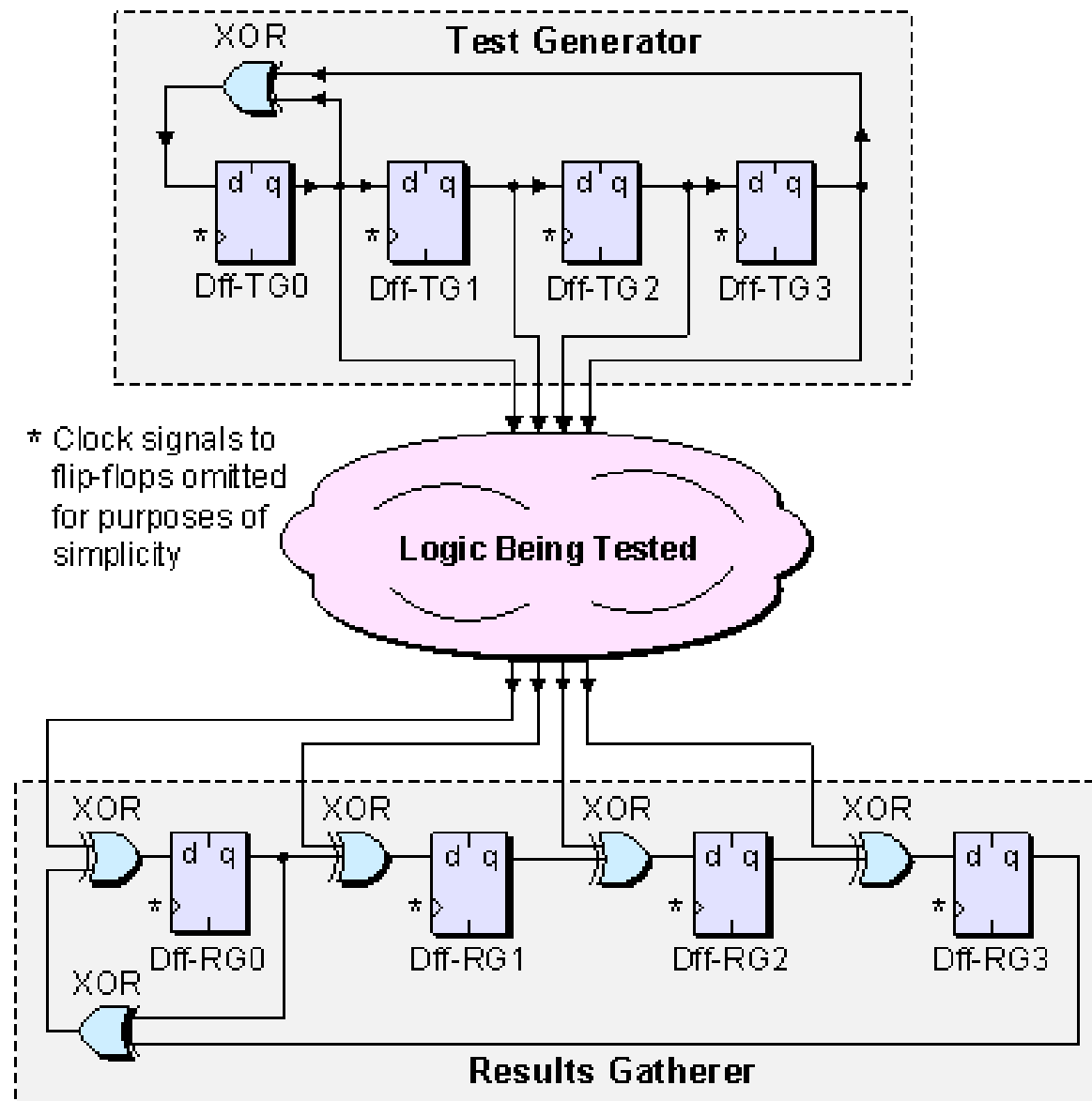
- The JTAG State Machine in operation a very simplified representation.
- In reality, the signal sampling is slightly different: sampling of signals happening on the rising or the falling edge, differently
- The "clock signal" visualization above is slightly buggy in that there is only one State Machine operation per clock cycle. Sampling happens with each Clock cycle.
- Note that data is read into and out of the TDI and TDO pins while the state machine is in the Shift-IR or Shift-DR states.
- The specific differences between these states, and the registers they utilize are beyond the scope of this course.
- IR (Instruction Register) and DR (Data Register) are like function calls. The value in IR is the "function" and the value in DR is the "argument". The JTAG specification doesn't define what the "functions" should be, that's up to the manufacturer.

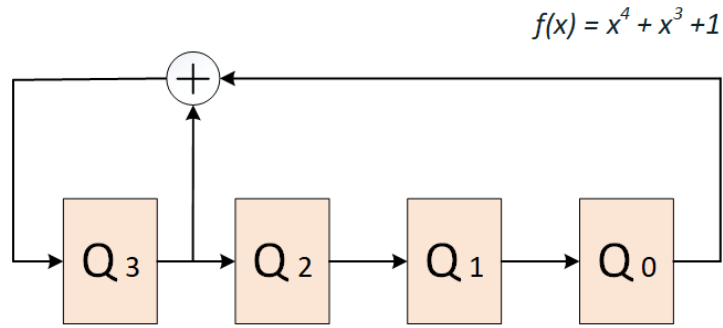
Logic Built In Self Test (LBIST)

- Test Access Port (TAP) to start.
- Automatic Test Equipment (ATE).
- Logic Built In Self Test (LBIST)
- Test Pattern Generator (TPG)
- Automatic Test Pattern Generator (ATPG)
- Response Analyzer (RA)
- Linear Feedback Shift Register (LFSR)

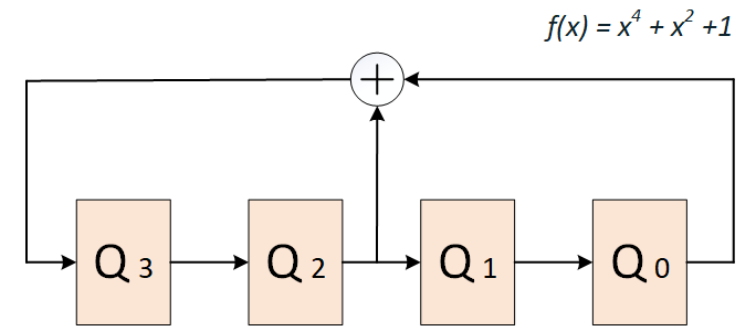




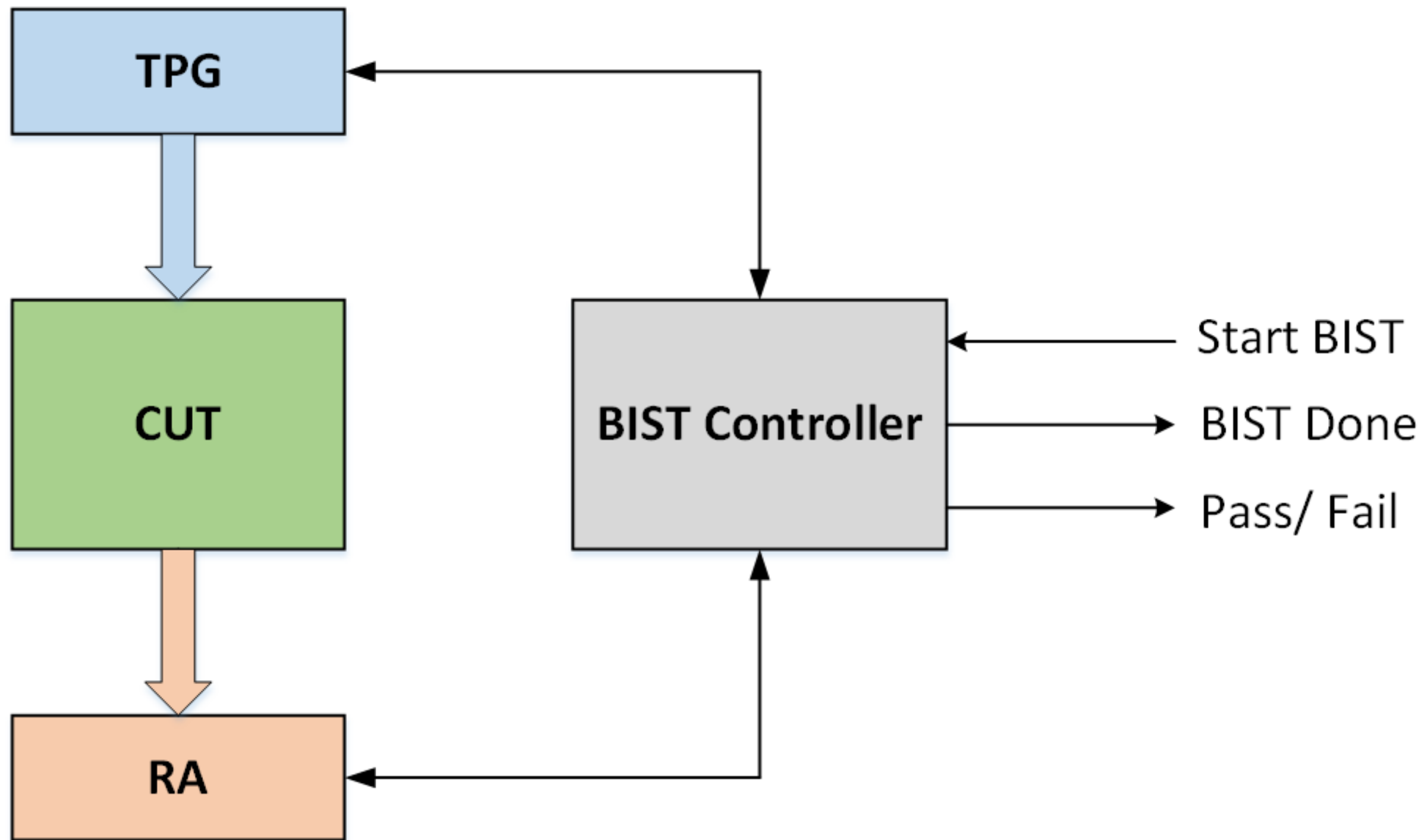


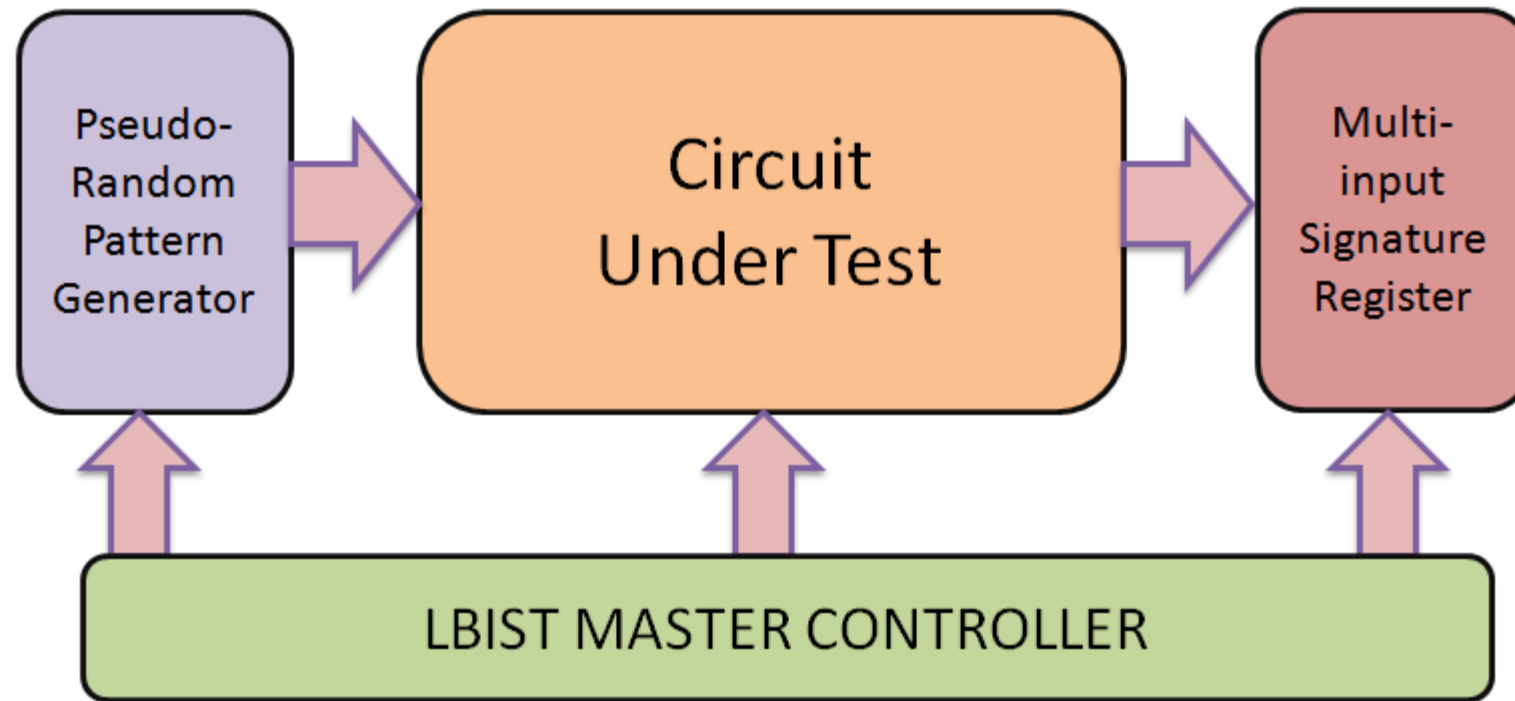


	Q_3	Q_2	Q_1	Q_0
Initial value :	1	0	0	0
1 st clock cycle :	1	1	0	0
2 nd clock cycle :	1	1	1	0
3 rd clock cycle :	1	1	1	1
4 th clock cycle :	0	1	1	1
5 th clock cycle :	1	0	1	1
6 th clock cycle :	0	1	0	1
7 th clock cycle :	1	0	1	0
8 th clock cycle :	1	1	0	1
9 th clock cycle :	0	1	1	0
10 th clock cycle :	0	0	1	1
11 th clock cycle :	1	0	0	1
12 th clock cycle :	0	1	0	0
13 th clock cycle :	0	0	1	0
14 th clock cycle :	0	0	0	1
15 th clock cycle :	1	0	0	0



	Q_3	Q_2	Q_1	Q_0
Initial value :	1	0	0	0
1 st clock cycle :	0	1	0	0
2 nd clock cycle :	1	0	1	0
3 rd clock cycle :	0	1	0	1
4 th clock cycle :	0	0	1	0
5 th clock cycle :	0	0	0	1
6 th clock cycle :	1	0	0	0





- **SISR (Single Input Signature Register)**
- **MISR (Multi-Input Signature Register)**
- **PRPG (Pseudo-Random Pattern Generator)**
- **RA (Response Analyzer)**

Advantages of LBIST:

There are many unique advantages of LBIST that make it desirable, especially in safety critical designs such as those used in automobiles and aeroplanes. LBIST offers many advantages as listed below:

- LBIST provides self-test capability to logic inside chip; thus, the chip can test itself without any external control and interference.
- This provides the ability to be tested at higher frequencies reducing test time considerably.
- LBIST can run while the chip is on field running functionally. Thus, it is very useful in safety critical applications wherein faults developed on field can be easily detectable at startup before chip goes into functional mode.

Overheads due to LBIST

- Along with many advantages, there are some overheads due to LBIST as mentioned below:
 - The LBIST implementation involves some hardware on-chip to control LBIST. So, there are area and power impacts due to these. In other words, the cost of chip increases.
 - Also, 'X'-masking involves addition of extra logic gates in already timing critical functional signals causing impact on timing as well.
 - Another disadvantage of using LBIST is that even the on-chip test equipment may fail. This is not the problem with testing using outside equipment with proven test circuitry

What is the role of Response Analyzer (RA)?

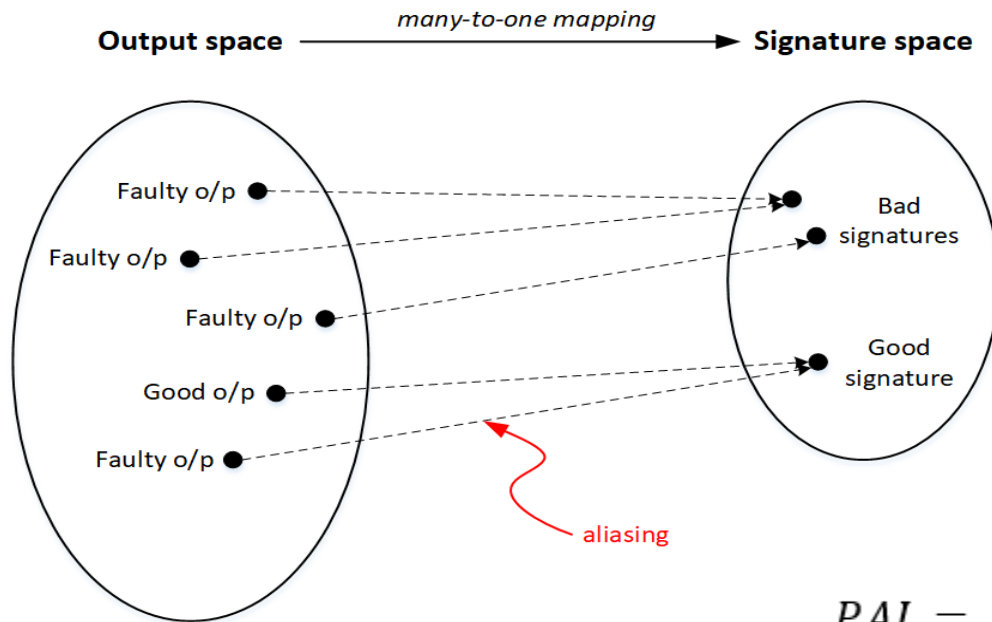
- It compresses the CUT output responses into a small signature, so that it can be stored on-chip.
- It compares the signature (generated in silicon) with the gold signature (generated in pre silicon) to determine Pass/Fail.
- It is also called signature analyzer or output response analyzer.

What are the characteristics of a good RA?

- The signature generated should be as small as possible, so that it occupies less memory while storing the gold signature on-chip.
- It should have correct Pass/Fail decision, i.e. low aliasing.
- The RA logic should be as small as possible, implies less area overhead.
- It should support diagnosis, implies when CUT fails it should be able to find the source of failure.

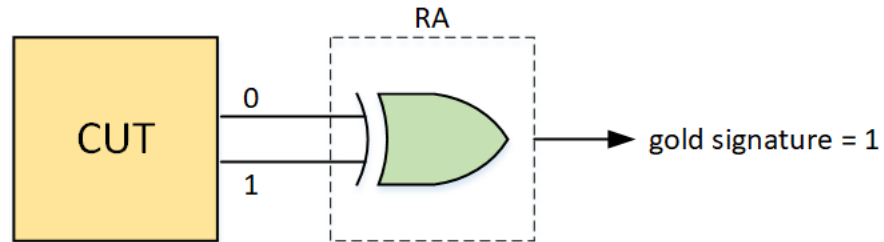
Aliasing

- Aliasing occurs when the signature generated by a faulty output is same as the gold signature. $\text{Signature}_{\text{faulty o/p}} = \text{Signature}_{\text{good o/p}}$
- Thus aliasing can lead to loss in fault coverage as we cannot cover the faults that will be generating the gold signature.



$$PAL = \frac{\text{Number of faulty outputs that generated gold signature}}{\text{Total number of faulty outputs}}$$

Probability of Aliasing (PAL)

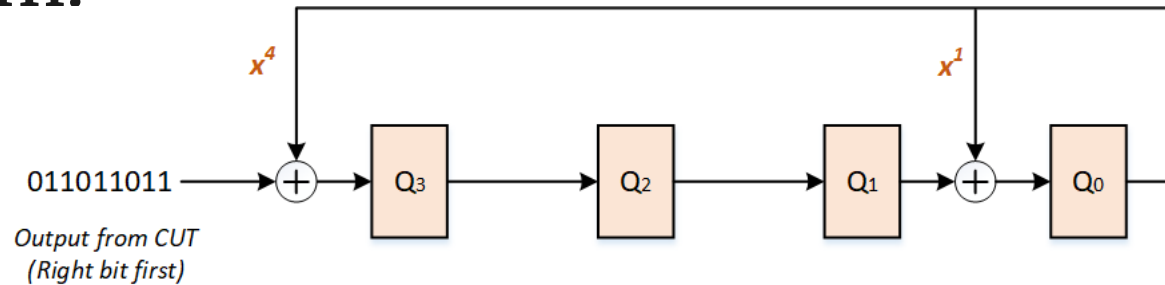


CUT's output	Signature (RA's output)
01	1 (gold signature)
10	1 (aliasing occurs)
00	0
11	0

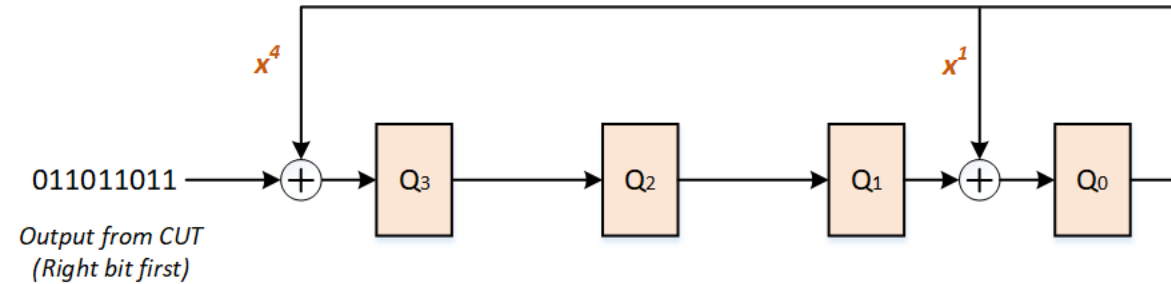
$$\text{Thus, } PAL = \frac{\{10\}}{\{00,10,11\}} = \frac{1}{3}$$

LFSR based Response Analyzer (RA) – serial

- Consider a LFSR with a characteristic polynomial $f(x) = x^4 + x + 1$.
- The external input to the LFSR is a bit stream coming from a CUT. Let us find the signature generated by this input stream.



LFSR based Response Analyzer (RA) – serial



Cycle	LFSR Input	Q ₃ Q ₂ Q ₁ Q ₀
0	011011011	0 0 0 0
1	01101101	1 0 0 0
2	0110110	1 1 0 0
3	011011	0 1 1 0
4	01101	1 0 1 1
5	0110	0 1 0 0
6	011	0 0 1 0
7	01	1 0 0 1
8	0	0 1 0 1
9		1 0 1 1

CRC Theory

- It represents the input bit streams by a polynomial.

Example: The bit stream 011011011 (The right most bit is the first bit to enter LFSR) shown above, can be represented as $x + x^2 + x^4 + x^5 + x^7 + x^8$.

- Now consider the same LFSR shown,
whose polynomial is $f(x) = x^4 + x + 1$.

The modular LFSR acts as a Modulo-2 divider, whose –
Dividend = LFSR input bit stream = $x + x^2 + x^4 + x^5 + x^7 + x^8$

Divisor = LFSR characteristic polynomial = $x^4 + x + 1$

Quotient = we calculate it

Remainder = the Signature

- *Note: In Modulo-2 arithmetic, Addition = Subtraction = XOR*

Modulo-2 Divison

$$\begin{array}{r} x^4 + x^3 + 1 \\ x^4 + x + 1 \overline{) \begin{array}{l} x^8 + x^7 + x^5 + x^4 + x^2 + x \\ x^8 + + x^5 + x^4 \\ \hline x^7 + + x^2 + x \\ x^7 + x^4 + x^3 \\ \hline x^4 + x^3 + x^2 + x \\ x^4 + + + x + 1 \\ \hline x^3 + x^2 + 1 \end{array}} \end{array}$$

As you can see the Remainder is $1 + x^2 + x^3$, which is equivalent to **1011** (same as the signature calculated from the table)

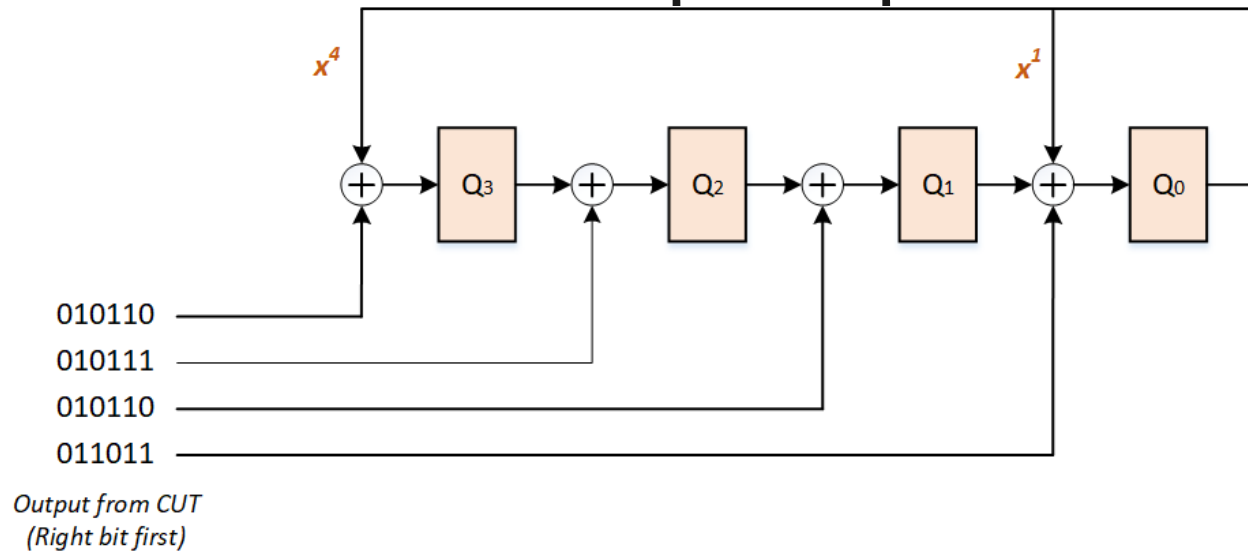
Probability of Aliasing (PAL) Estimate of LFSR based serial RA

M = length of input bit stream
N = degree of polynomial

$$PAL = \frac{2^{M-N} - 1}{2^M - 1} \approx 2^{-N} \text{ (if } M \gg N)$$

LFSR based RA – parallel

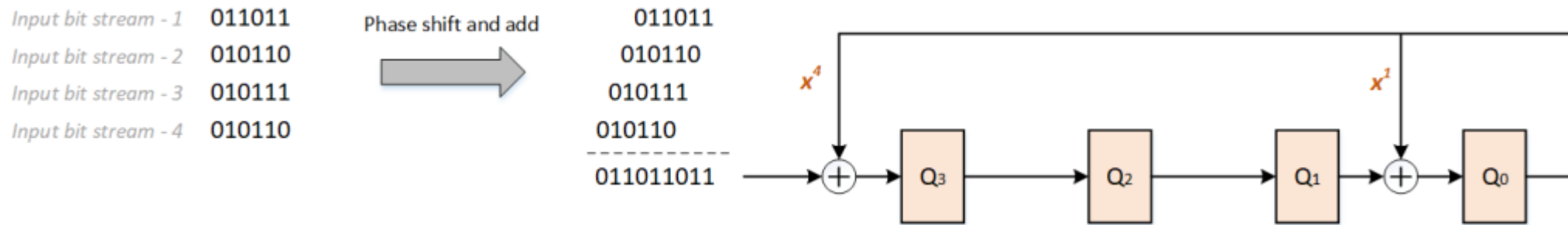
- Serial RA only compress one CUT output at a time, implies for compressing multiple CUT outputs at a time we need one LFSR for each CUT output, which will lead to too much hardware overhead. Therefore we use a parallel LFSR based RA called Multiple Input Shift Register (MISR).



Cycle	MISR Inputs	Q ₃ Q ₂ Q ₁ Q ₀
0	011011 010110 010111 010110	0 0 0 0
1	01101 01011 01011 01011	0 1 0 1
2	0110 0101 0101 0101	0 1 0 0
3	011 010 010 010	1 1 0 0
4	01 01 01 01	0 1 1 1
5	0 0 0 0	0 1 0 1
6		1 0 1 1

Equivalent LFSR of a MISR

- MISR and its input bit stream can be mapped to its corresponding equivalent LFSR by just phase shifting and adding the input bit stream as shown.



Probability of Aliasing (PAL) Estimate of MISR

- K = length of input bit stream
 M = length of equivalent LFSR bit stream = $K + N - 1$
 N = degree of polynomial

$$PAL = \frac{2^{M-N} - 1}{2^M - 1} = \frac{2^{K-1} - 1}{2^{N-K-1} - 1} \approx 2^{-N} \text{ (if } K \gg N)$$

Masking in MISR

- Masking means one error bit cancels another error bit before reaching the MISR feedback tap points.
- Consider the scenario shown below:

	<u>Golden input bit stream</u> <i>Input bit stream in the absence of Error</i>	<u>Case 1</u> <i>Input bit stream in the presence of Error</i>	<u>Case 2</u> <i>Input bit stream in the presence of Error</i>
<i>Input bit stream - 1</i>	011011	011011	011011
<i>Input bit stream - 2</i>	010110	01011 1	010 0 10
<i>Input bit stream - 3</i>	010111	0101 0 1	0101 0 1
<i>Input bit stream - 4</i>	010110	010110	010110
	-----	-----	-----
<i>Equivalent LFSR input bit stream</i>	011011011	01101 000 1	011011011
			↑ <i>Defect cannot be caught</i>

- Assuming there is no aliasing, the signature generated in Case 1 will be different than the golden signature. However the signature generated in Case 2 will be same as the golden signature as the equivalent LFSR input bit stream in the presence of error is same as the golden input bit stream. This is known as Masking.
- Probability of Masking (P_{masking}) = Probability of even number of 1s $P_{\text{masking}} \approx 2^{1-N-K}$, which is $\ll \text{PAL } (2^{-N})$

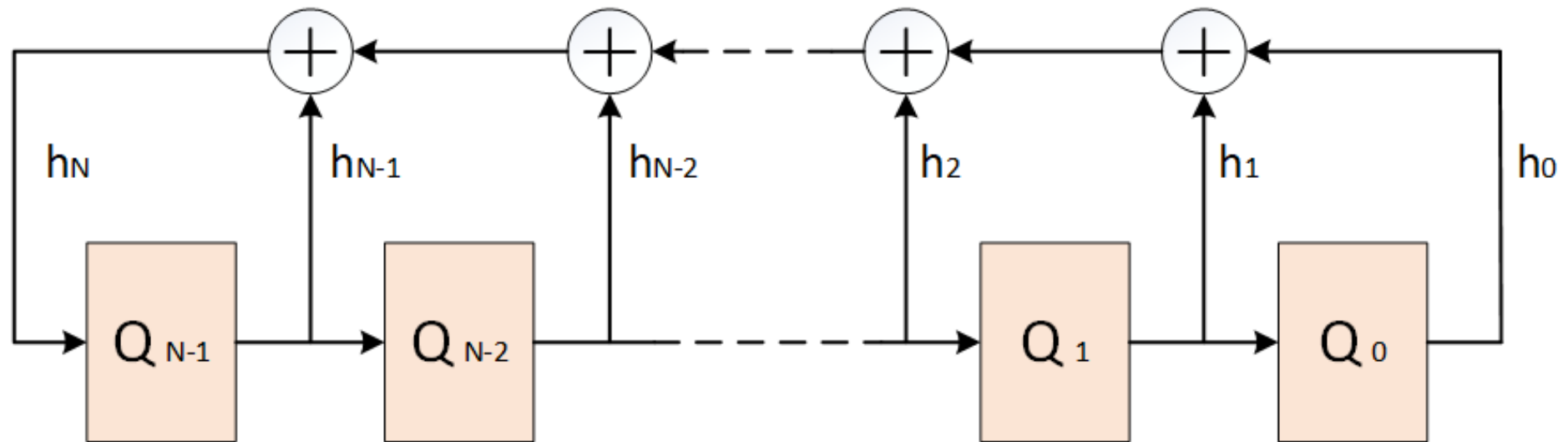
K = length of input bit stream
 N = degree of polynomial

Linear Feedback Shift Register (LFSR)

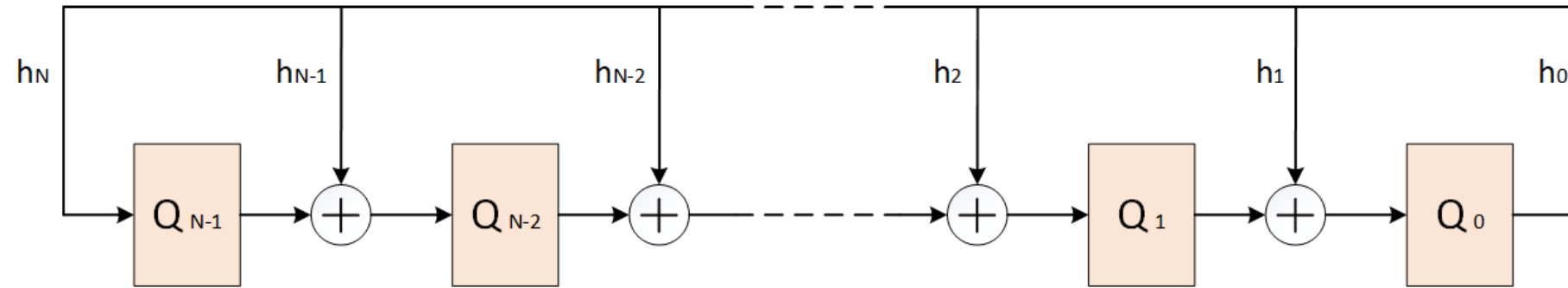
Intro

- An n -bit Linear Feedback Shift Register (LFSR) consists of ' n ' memory elements (or flops) and XOR gates.
- The output from a standard shift register is cunningly manipulated and fed back into its input in such a way as to cause the function to endlessly cycle through a sequence of patterns.
- There are basically two types of LFSR
 1. Standard Form (also known as External Feedback LFSR)
 2. Modular Form (also known as Internal Feedback LFSR)

Standard Form LFSR



Modular Form LFSR



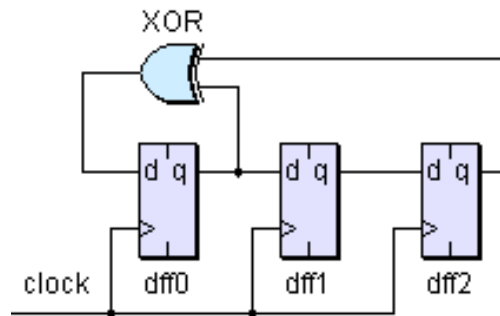
- LFSRs can be represented by its characteristics polynomial :

$$F(x) = h_n x^n + h_{n-1} x^{n-1} + \dots + h_1 x + h_0,$$

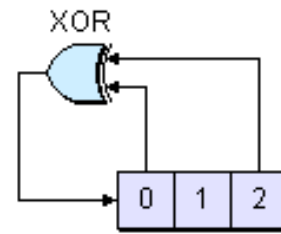
- where the term $h_i x^i$ refers to the i^{th} flop of the register.
- Standard Form LFSR
 - The modulo-2 sum of the selected stages indicated by the characteristics polynomial is fed back to the 1st stage of the LFSR.
 - The speed is limited by the depth of the linear logic in its feedback path. Effect speed is determined by the number of XOR gates in the feedback path.
- Modular Form LFSR
 - The output of the last stage of the LFSR is fed back to the stages indicated by the characteristics polynomial.
 - Implementation involves a large fan-out on the output of the last stage. Theoretically up to 'n' fan-outs possible for a n-bit LFSR, which leads to timing challenges for large LFSRs.

TAPs

- One of the more common forms of LFSR is formed from a simple shift register with feedback from two or more points, or *TAPs*, Test Access Port (TAP)



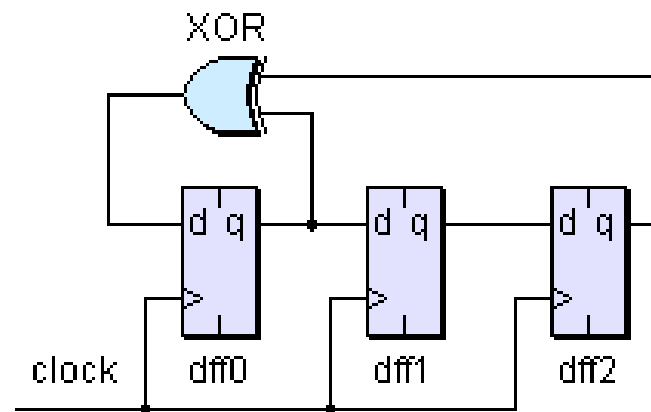
(a) Circuit



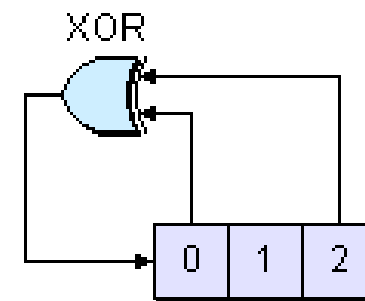
(b) Symbol

- The taps in this example are at bit 0 and bit 2, and can be referenced as [0,2].

- All of the register elements share a common clock input, which is omitted from the symbol (b) for reasons of clarity.
- The data input to the LFSR is generated by XOR-ing or XNOR-ing the tap bits;
- The remaining bits function as a standard shift register.

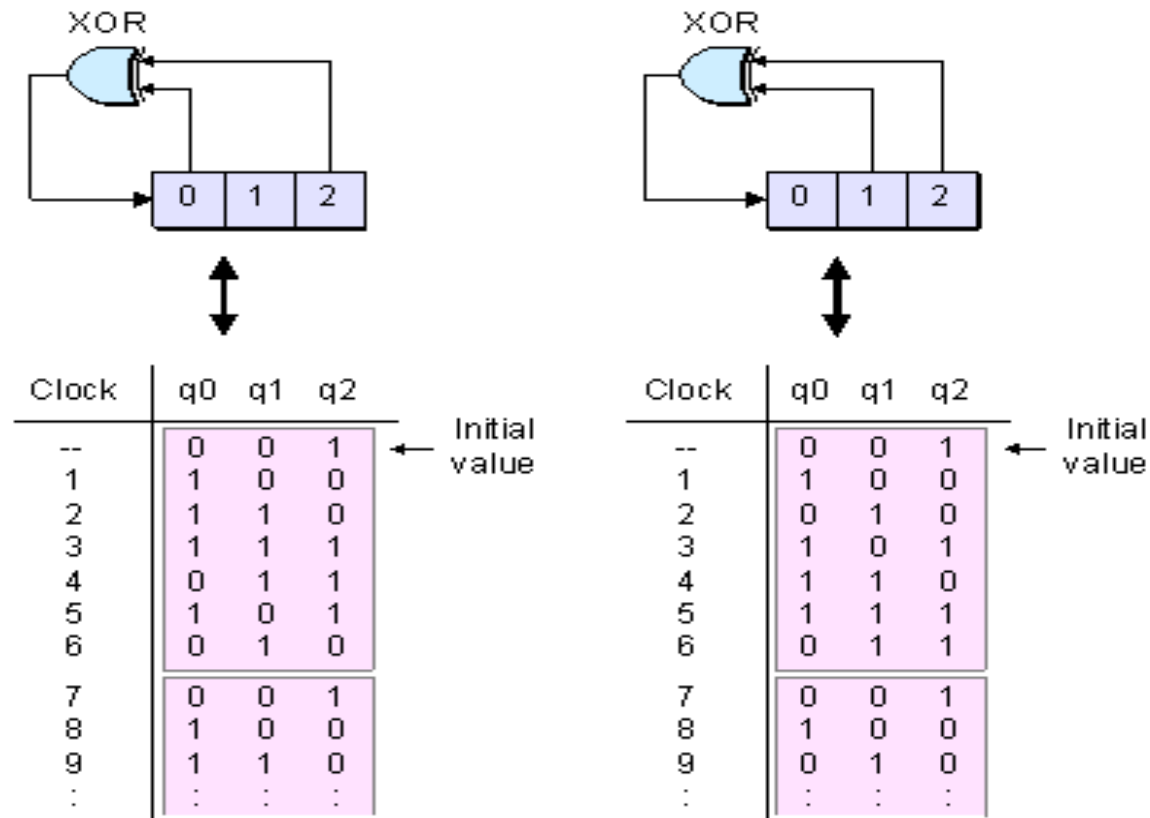


(a) Circuit

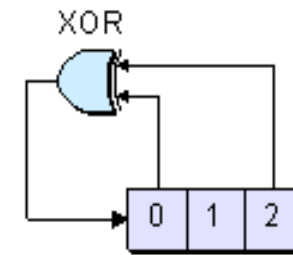


(b) Symbol

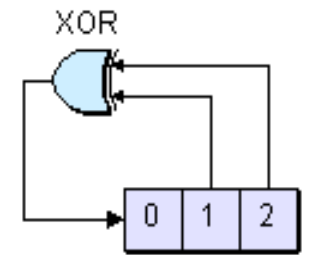
- The sequence of values generated by an LFSR is determined by its feedback function (XOR versus XNOR) and tap selection.
- For example, consider two 3-bit XOR based LFSRs with different tap selections



- Both LFSRs start with the same initial value but, due to the different taps, their sequences rapidly diverge as clock pulses are applied.
- In some cases an LFSR will end up cycling round a loop comprising a limited number of values.
- However, both of the LFSRs shown are said to be of maximal length because they sequence through every possible value (excluding all of the bits being 0) before returning to their initial values.



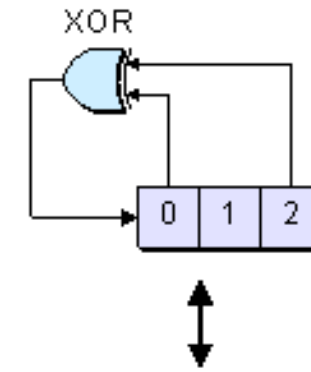
Clock	q0	q1	q2	
--	0	0	1	Initial value
1	1	0	0	
2	1	1	0	
3	1	1	1	
4	0	1	1	
5	1	0	1	
6	0	1	0	
7	0	0	1	
8	1	0	0	
9	1	1	0	
:	:	:	:	



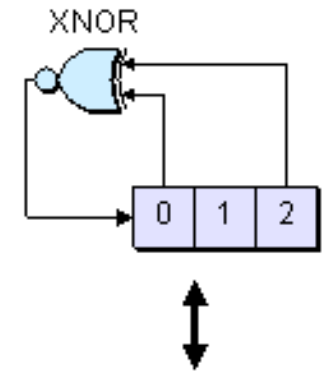
Clock	q0	q1	q2	
--	0	0	1	Initial value
1	1	0	0	
2	0	1	0	
3	1	0	1	
4	1	1	0	
5	1	1	1	
6	0	1	1	
7	0	0	1	
8	1	0	0	
9	0	1	0	
:	:	:	:	

- A binary field with 'n' bits can assume 2^n unique values, but a maximal-length LFSR with 'n' register bits will only sequence through $(2^n - 1)$ values.

- This is because LFSRs
 - XOR feedback paths will not sequence through the value where all the bits are 0,
 - XNOR equivalents will not sequence through the value where all the bits are 1



Clock	q0	q1	q2	
--	0	0	1	Initial value
1	1	0	0	
2	1	1	0	
3	1	1	1	All 1s
4	0	1	1	
5	1	0	1	
6	0	1	0	
7	0	0	1	
8	1	0	0	
9	1	1	0	
:	:	:	:	



Clock	q0	q1	q2	
--	0	0	1	Initial value
1	0	0	0	
2	1	0	0	
3	0	1	0	All 0s
4	1	0	1	
5	1	1	0	
6	0	1	1	
7	0	0	1	
8	0	0	0	
9	1	0	0	
:	:	:	:	

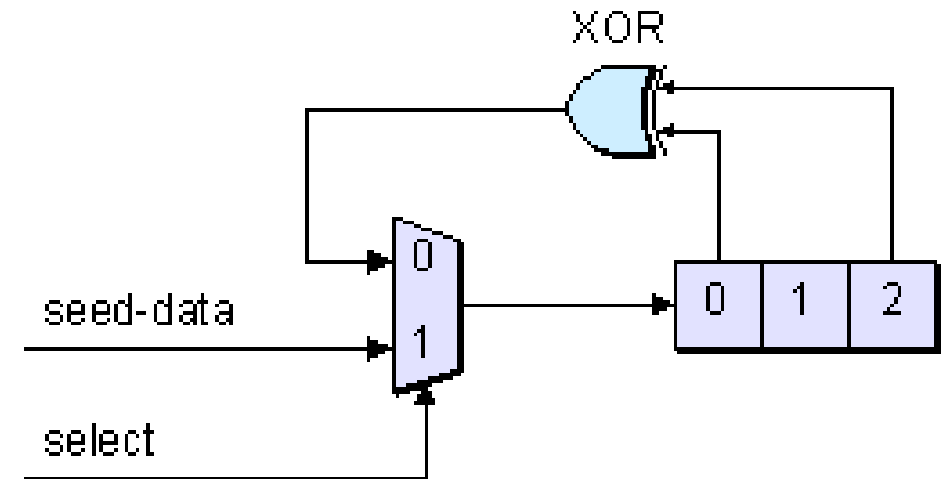
- Each LFSR supports a number of tap combinations that will generate maximal-length sequences.
- The problem is weeding out the ones that do from the ones that don't, because badly chosen taps can result in the register entering a loop comprising only a limited number of states.
- One of the taps for maximal-length LFSRs with 2 to 32 bits are presented in the right side (the * annotations indicate sequences whose length is a prime number).

# of Bits	Length of Loop	Taps
2	3 *	[0,1]
3	7 *	[0,2]
4	15	[0,3]
5	31 *	[1,4]
6	63	[0,5]
7	127 *	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1,023	[2,9]
11	2,047	[1,10]
12	4,095	[0,3,5,11]
13	8,191 *	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]
17	131,071 *	[2,16]
18	262,143	[6,17]
19	524,287 *	[0,1,4,18]
20	1,048,575	[2,19]
21	2,097,151	[1,20]
22	4,194,303	[0,21]
23	8,388,607	[4,22]
24	16,777,215	[0,2,3,23]
25	33,554,431	[2,24]
26	67,108,863	[0,1,5,25]
27	134,217,727	[0,1,4,26]
28	268,435,455	[2,27]
29	536,870,911	[1,28]
30	1,073,741,823	[0,3,5,29]
31	2,147,483,647 *	[2,30]
32	4,294,967,295	[1,5,6,31]

- The taps are identical for both XOR-based and XNOR-based LFSRs, although the resulting sequence will of course differ.
- As was previously noted, alternative tap combinations may also yield maximum-length LFSRs, although the resulting sequences will vary.
- For example, in the case of a 10-bit LFSR, there are two 2-tap combinations that result in a maximal-length sequence: [2,9] and [6,9].
- There are also twenty 4-tap combinations, twenty-eight 6-tap combinations, and ten 8-tap combinations that satisfy the maximal-length criteria.

Seeding an LFSR

- One quirk with an XOR-based LFSR is that, if it happens to find itself in the all 0s value, it will happily continue to shift all 0s indefinitely (similarly for an XNOR-based LFSR and the all 1s value).
- This is of particular concern when power is first applied to the circuit. Each register bit can randomly initialize containing either a logic 0 or a logic 1, and the LFSR can therefore “wake up” containing its “forbidden” value. For this reason, it is necessary to initialize an LFSR with a seed value.



Applications

- FIFO Applications
- Encryption and decryption applications
- Cyclic redundancy check (CRC) applications
- Data compression applications
- Built-in self-test (BIST) applications