IP/SOC Verification

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Verification Challenges (1)

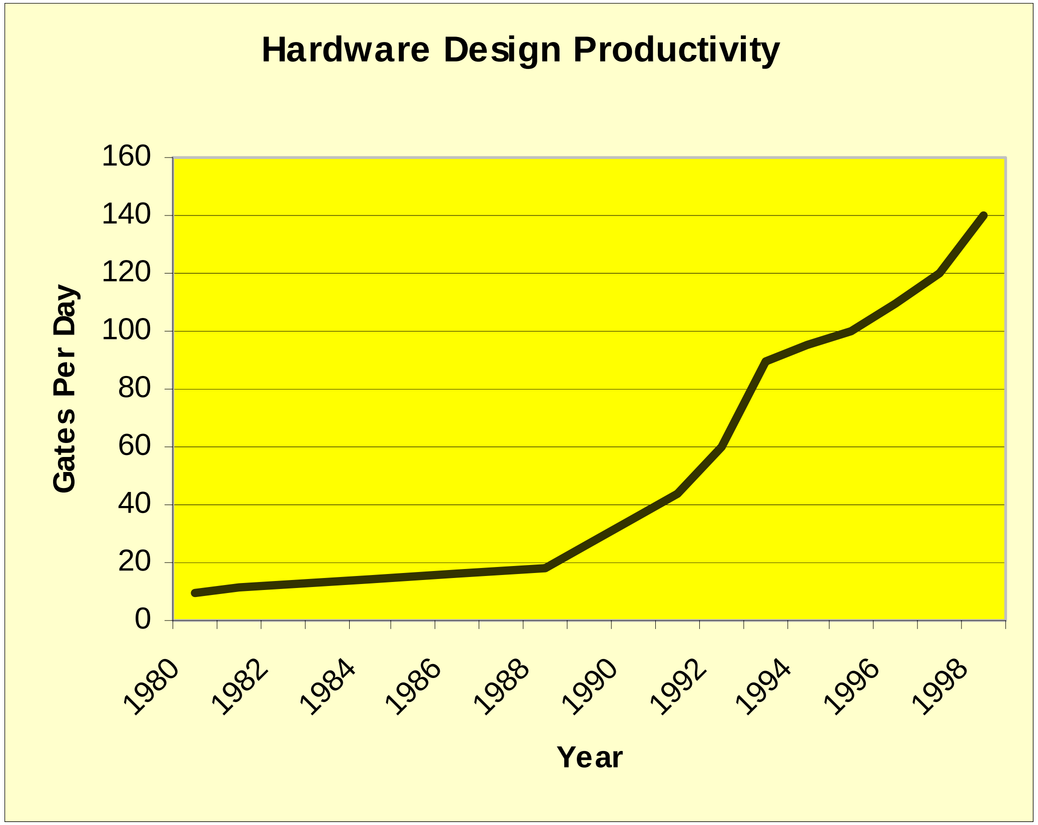
- Verification goals is ***100% correct***
  - ***Mission impossible***
- Macro-level testbenches and test suite must be **reusable**
  - For next redesigned macro
  - For integration team
    - Verified in standalone as well as in final applications
    - Testbench must be compatible with the system level verification tools

IP Verification

Tian-Sheuan Chang

Verification Challenge (2)

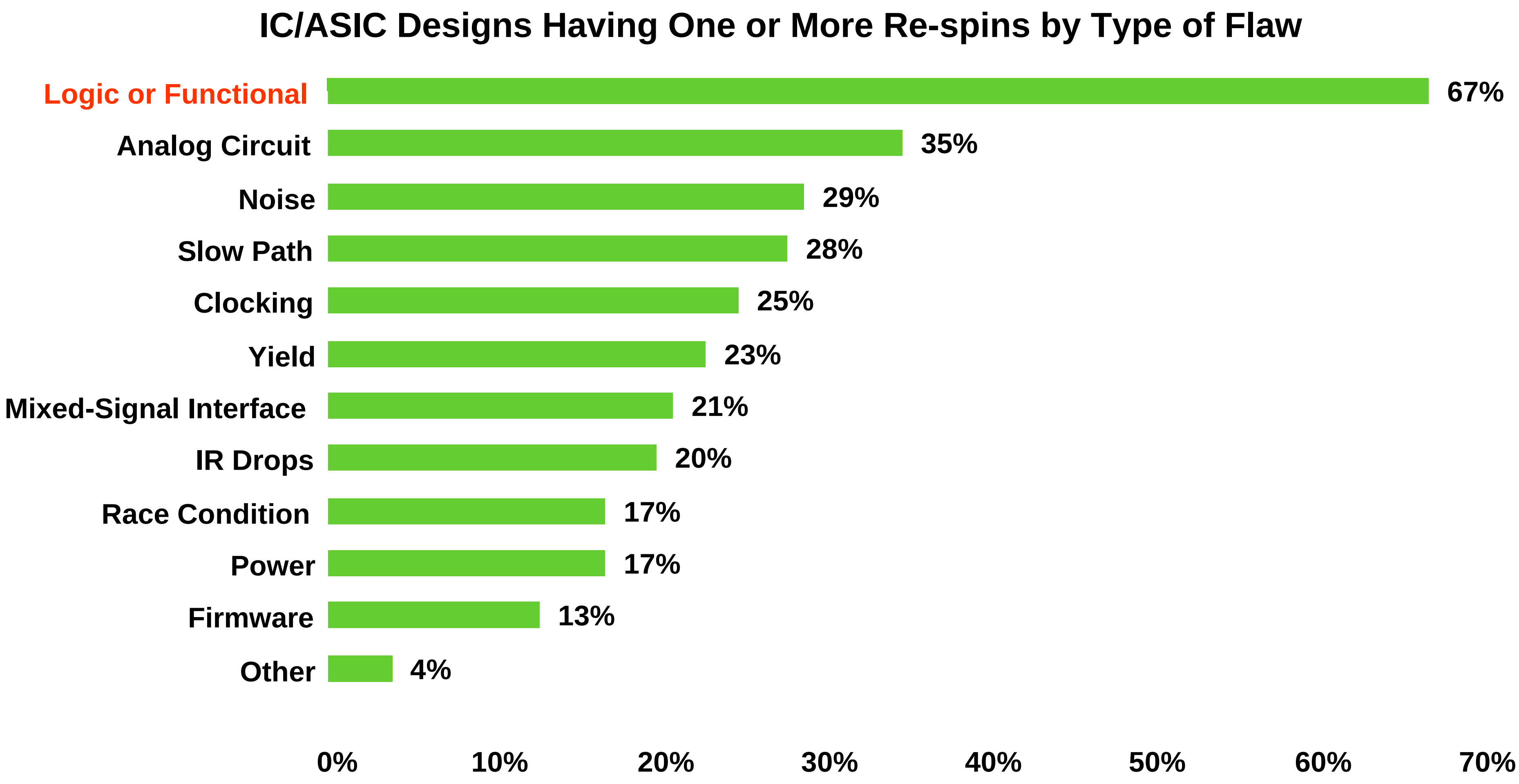
- Design Productivity has risen tenfold since 1990
  - Gain by synthesis tools contributed to this challenge
- Only able to verify approximately 100 gates/day



IP Verification

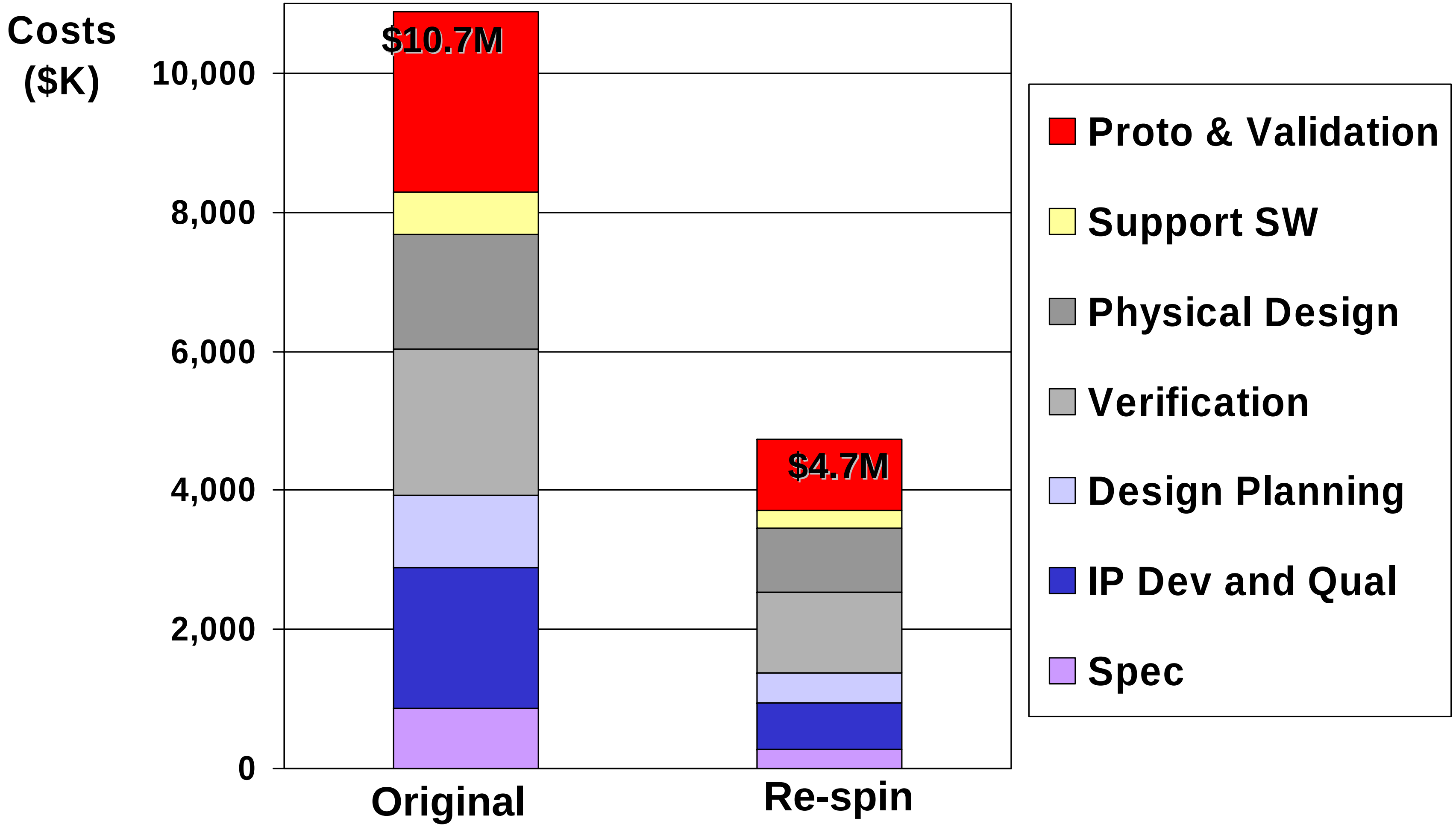
Tian-Sheuan Chang

Verification Challenge (3)



Source: Collett International Research (Apr02)

Re-spins are EXPENSIVE



Plus a) lost revenue, b) opportunity costs

Source: International Business Strategies, 2002

Verification and Design Reuse

- Reuse is about trust
- The key to design reuse is gaining that trust
- Verification for Reuse
  - Complete functional verification
  - All possible configurations
  - All possible uses

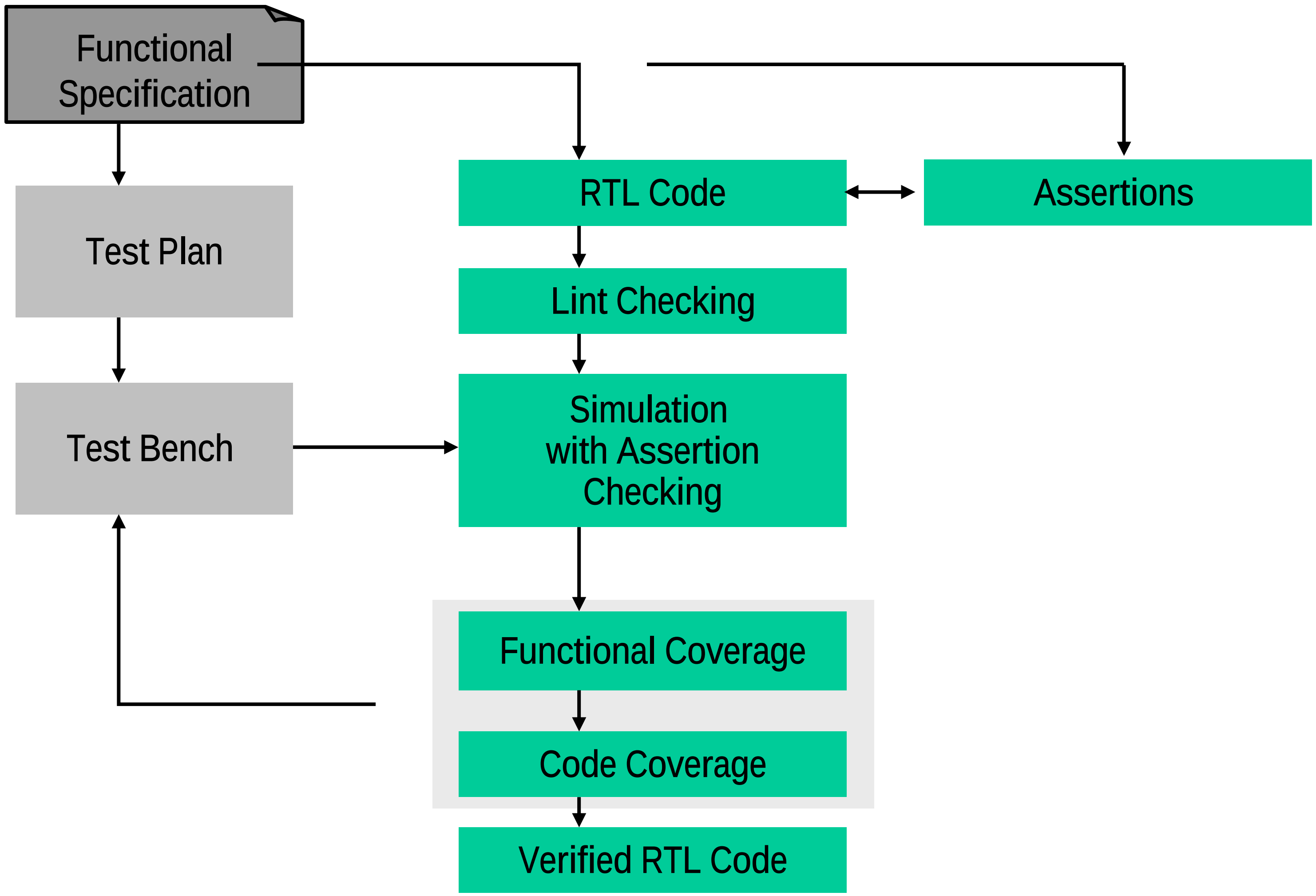
IP Verification

Tian-Sheuan Chang

Outline

- Verification challenges
- *Verification process*
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Boosting Productivity throughout the Verification Flow



IP Verification

Tian-Sheuan Chang

Verification Plan

- Part of early design cycle
- Verification takes over **70%** of development time
- Contents
  - Test strategy for subblock and top level
  - Simulation environment including a block diagram
  - Test bench components – BFM, bus monitors
  - Required verification tools
  - List of specific tests for the key features
  - Target code coverage
  - Regression test environment and regression procedure
  - Criteria when the verification process is completed

Role of Verification Plan

- Specifying the specification
- Defining First-Time Success
  - Ensures all essential features are appropriately verified
  - Which features must be exercised under what conditions and what is the expected response
- Define features priority
- How many testbenches must be written
- How complex they need to be
- How they depend on each other

Benefit of Verification Plan

- Force designers to **think through** the very time-consuming process before performing them
- **Peer review** allows a pro-active assessment of the entire scope
- Focus efforts first for area of **most needed and greatest payoff**
- Minimize the redundant effort
- Tracked and managed more **effectively**
- Enable verification tests and testbench early
- Enable a separated **verification team** in parallel to reduce design cycle

Tian-Sheuan Chang

From Specification to Feature

- Component-Level Features
  - Unit, reusable, ASIC level
  - Do not involve system-level interaction with other component
- System-Level Features
  - A subset of an ASIC, a few ASICs, an board design
  - Minimize the features verified at this level
  - Limited to connectivity, flow-control and inter-operability

Tian-Sheuan Chang

From Feature to Testcase

- Prioritize
  - **Must-have** – verify all possible configuration & usage
  - **Should-have** – verify basic functionality
  - **Nice-to-have** – verify only as time allow
- Group into testcases
  - Configuration, verification strategy
  - Testcase: labeled, objective description(list of features)
- Design for verification
  - Identify “hard-to-verify” features

From Testcase to Testbench

- Testcase naturally fall into **groups**
  - Configuration of the design
  - **Abstraction level** for the stimulus and response
  - Verify closely-related features
- Testbench
  - One testcase per testbench
  - Grouping several testcases into a single testbench
- Verifying testbenches
  - Review by other verification engineer
  - Simulation output log

Tian-Sheuan Chang

Verification Strategies

- Three phases
  - Subblocks
    - **Exhaustive** functionality verification
    - Ensure no syntax errors in the RTL code
    - Basic functionality is operational
    - Method: simulation, code coverage, TB automation
  - Macro
    - **Interface verification** between subblocks
    - Backward compatible (regression test suite)
    - Method: simulation, code coverage, TB automation, hardware accelerator
  - Prototyping
    - Real prototype runs real application software
    - Method: emulator, FPGA, ASIC test chip
- Bottom up approaches
  - **Locality**
  - Easier and faster to catch bugs at the lower level

Tian-Sheuan Chang

Types of Verification Tests

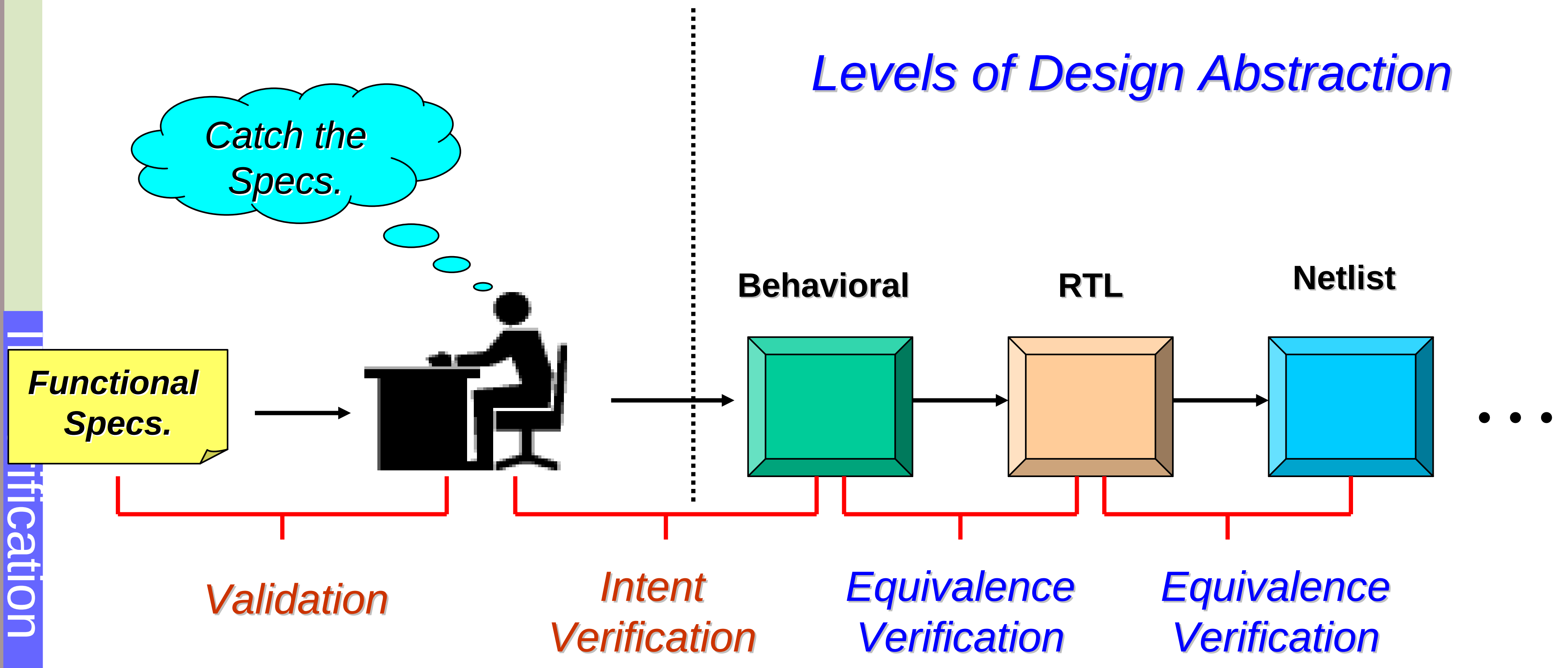
- Compliance testing
  - For standard based design
- Corner case testing
- Random verification
  - Inputs are subjected to **valid** individual operations
  - Prediction of the expected outputs is more complicated
  - Create the condition you have not thought
  - Hit corner cases
- Assertion-based verification (Property checking)
- Real code testing
  - Avoid misunderstand specification
- Regression testing
  - Verify that bug fixing won't create new bugs
  - Run on regular basis

Taxonomy

- Functional Verification
  - Dynamic
    - Simulation based
    - Require input vectors
    - No 100% guarantee
  - Formal/static
    - Property
      - Mathematical proof
      - No input vectors
      - 100% guarantee
    - Classifications
      - Equivalence checking
      - Model checking
  - Semi-/dynamic formal
    - Simulation based
    - Check assertions during simulations
- Timing Verification
  - Dynamic timing analysis
    - Simulation based
    - Require input vectors
    - No 100% guarantee
    - Used in gate-level simulation
    - Useful for timing verification of power-up sequences and timing exception path, e.g. asynchronous logic, multi-cycle paths, false paths,
  - Static timing analysis
    - Exhaustive search
    - No input vectors
    - 100% guarantee
    - No simulation required
    - Fastest approach
    - Sometimes pessimistic due to incorrect timing exceptions

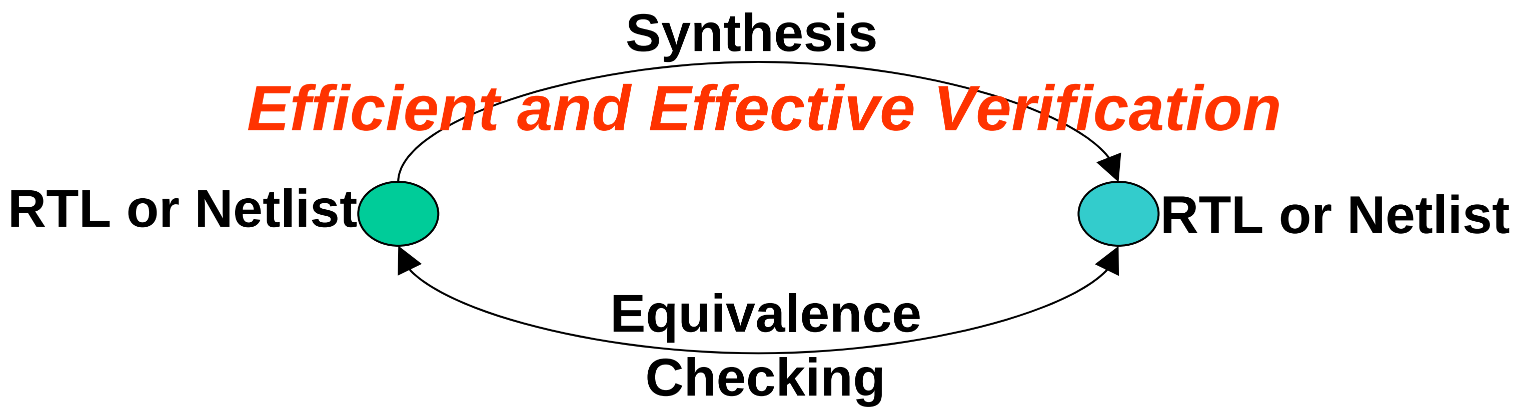
Tian-Sheuan Chang

Classification of Verification



Functional Verification Methodology

- RTL remains the ***golden*** model throughout the course of functional verification
- Apply extensive functional verification on RTL
  - Simulation, code coverage, functional coverage, property checking, assertion-based checking
- Use ***equivalence checking*** to keep it golden for successive design transformations



IP Verification

Tian-Sheuan Chang

Outline

- Verification challenges
- Verification process
- *Verification tools*
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Verification Tools (1/2)

- Simulation
  – Event driven: good debug environment
  – Cycle based: fast simulation time
- Code coverage
  – No. of executed lines / total lines
  – Coverage on RTL structure
  – Verification Navigator, CoverMeter
- Hardware verification languages
  – A language providing power constructs for generating stimulus and checking response
  – Aid creating verification IP and reusable testbenches
  – Vera, e, System Verilog, TestBuilder

Verification Tools (2/2)

- Functional coverage
  – Coverage on functionality
- Formal property checking
  – Verplex BlackTie, 0-In Search/Confirm
- Verification IP (VIPs)
  – Bus functional model (BFM) and bus monitors for standard protocols
- Hardware modeling
- Emulation
- Prototyping
  – FPGA
  – ASIC test chip

Tian-Sheuan Chang

Inspection as Verification

- Fastest, cheapest and most effective to detect and remove bugs
- How
  - Design (specification, architecture) review
  - Code (implementation) review
    - Line-by-line fashion
    - At the subblock level
    - Reviewer should fully understand the implementation
    - Purpose is to help drive quality and not for performance assessment
- Lint tool help spot defects w/o simulation
  - VN-Check, nLint, LEDA

Tian-Sheuan Chang

Adversarial Testing

- Designer
  - Focus on proving the design is **right**
- Verification team
  - Prove the design is **broken**
  - Keep with the latest tools and methodologies
- The combination of the two gives the best results

Tian-Sheuan Chang

Limited Production

- Even after robust verification and prototyping, it's still not guaranteed to be bug free
- A limited production for new macro is necessary
  - 1 to 4 customers
  - Small volume
  - Reduce the risk of supporting problems

Tian-Sheuan Chang

Coverage

- A metric identifies important:
  - Structures in a design representation
  - e.g. HDL lines, FSM states, paths in netlist
- Classes of behavior
  - Transactions, event sequences
- Maximize the probability of simulating and detecting bugs, at minimum cost (in time, labor, and computation) [Dill ICCAD 99]
- Difficult to formally prove that a coverage metric provides a good proxy for bugs
- Goal
  - Comprehensive validation without redundant effort

Coverage Metric Classifications

- Ad-hoc metrics
  - Bug detection frequency
  - Length of simulation after last bug
  - Total number of simulation cycles
- Code coverage
  - Line coverage
  - Branch coverage
  - Path coverage
  - Expression Coverage
  - Toggle Coverage
- Functional coverage

Coverage (1/2)

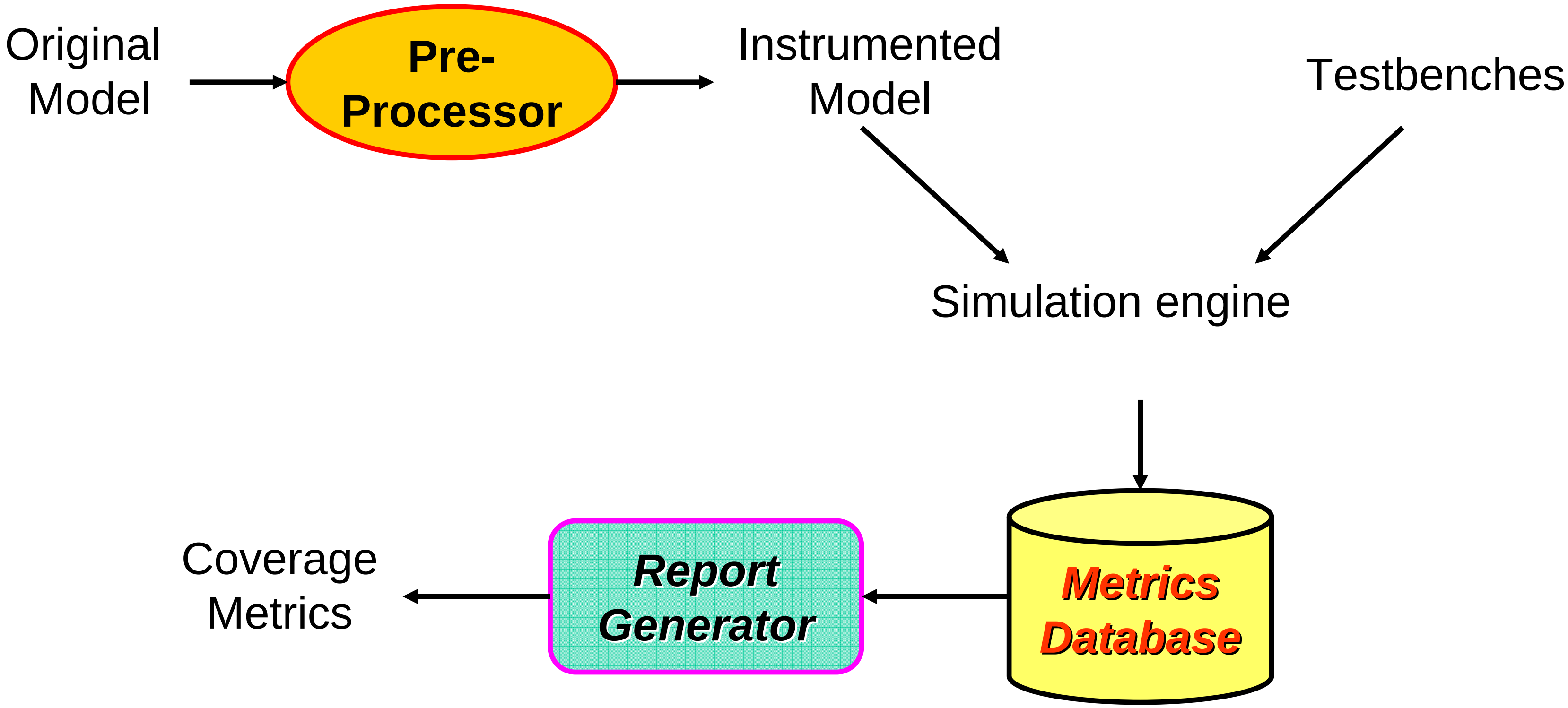
- Hardware code coverage
  - Statement, branch, condition, path, toggle, triggering, FSM
  - Recommended **100% statement, branch and condition**
  - 100% code coverage does not mean 100% functional coverage
  - Optimize regression suite runs
    - Redundancy removal
    - Minimize regression test suites
  - Quantitative stopping criterion
  - **Verify more but simulate less**
- Functional coverage
  - A user-defined metric that reflects the degree to which functional features have been exercised during the verification process.

Tian-Sheuan Chang

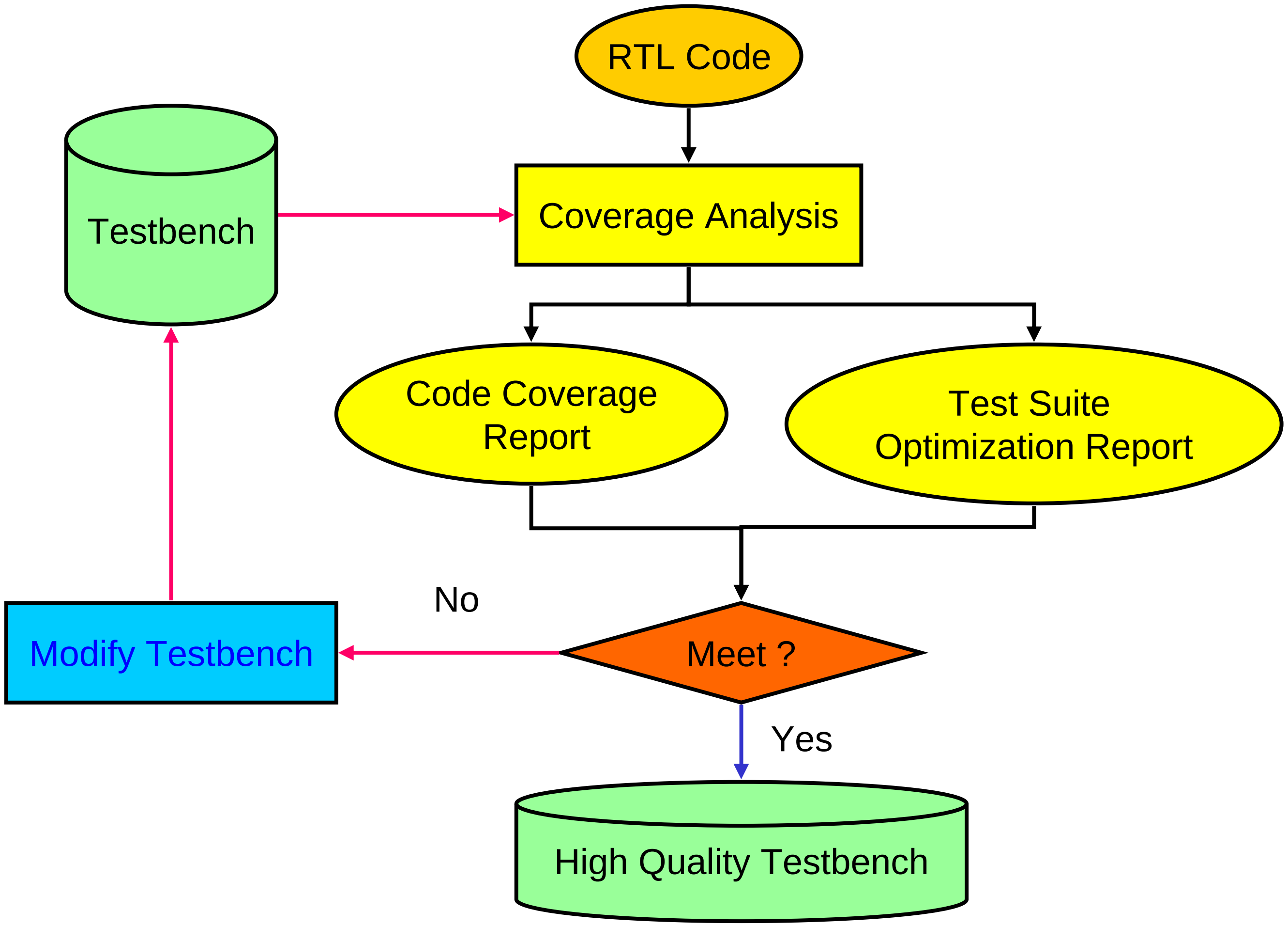
Coverage (2/2)

- The "**Coverage First**" Paradigm
  - Identify areas that were sufficiently exercised, and therefore need not be exercised any further
  - Replace the need to write a lot of deterministic, delicately crafted test, by showing that these scenarios were already encountered
- Functional Coverage
  - You can achieve 100% code coverage, and still miss key areas where bugs can be hiding.
  - It can eliminate the need to write many of the most time consuming and hard to write tests.

Code Coverage Process



Code Coverage Flow



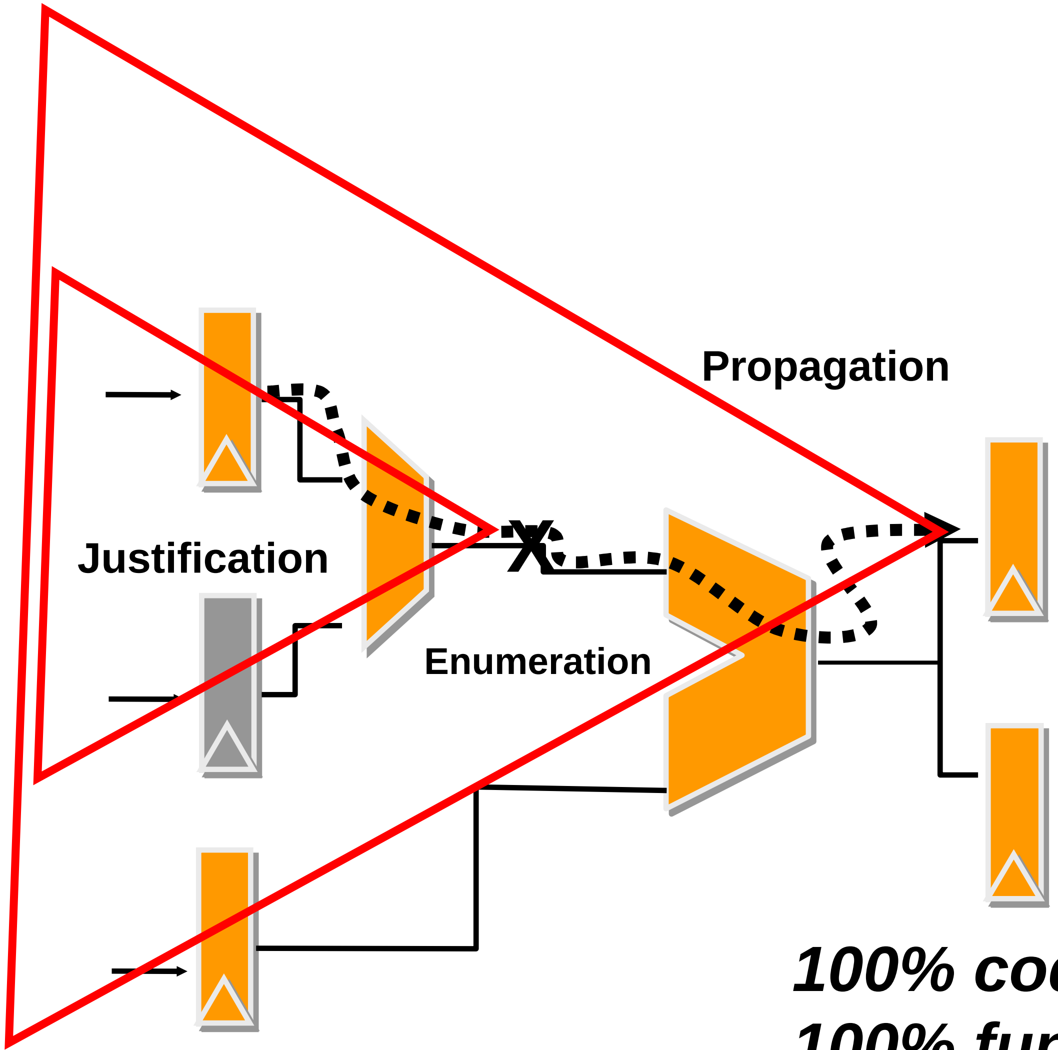
IP Verification

Tian-Sheuan Chang

Drawbacks of Code Coverage

- No **qualitative** insight into functional correctness
- Limited to measuring what is **controllable**
- Activating an erroneous statement does not mean the bug will manifest itself to an observable output
  - Like testing problems
  - Cases found where 90% line coverage only achieved 54% observability coverage [Devadas et al. ICCAD 96]

Problems with Existing Coverage Tools



Controllability

vs.

Observability

100% code coverage does not imply 100% functional coverage !

Increase Observability

- Black-box testing vs. white-box testing
- Event-Monitors and Assertion Checkers
  - Halt simulation (if desired)
  - Simplifies Debugging
  - Increases test stimuli observability
  - Measure functional coverage (using a line cover tool)
  - Enables formal and semi-formal techniques
  - Capture and validate design assumptions and constraints

Assertion-based Verification

Power of Assertion (1/3)

- DEC Alpha 21164 project [Kantrowitz et al.,DAC 1996]

Assertion Checkers	34%
Cache Coherency Checkers	9%
Reference Model Comparison	
Register File Trace Compare	8%
Memory State Compare	7%
End-of-Run State Compare	6%
PC Trace Compare	4%
Self-Checking Test	11%
Manual Inspection of Simulation Output	7%
Simulation hang	6%
Other	8%
Simulation hang	6%
Other	8%

IP Verification

Tian-Sheuan Chang

Power of Assertion (2/3)

- DEC Alpha 21264 project [Taylor et al.,DAC 1998]

Assertion Checker	**25%**
Register Miscompare	**22%**
Simulation "No Progress”	**15%**
PC Miscompare	**14%**
Memory State Miscompare	**8%**
Manual Inspection	**6%**
Self-Checking Test	**5%**
Cache Coherency Check	**3%**
SAVES Check	**2%**

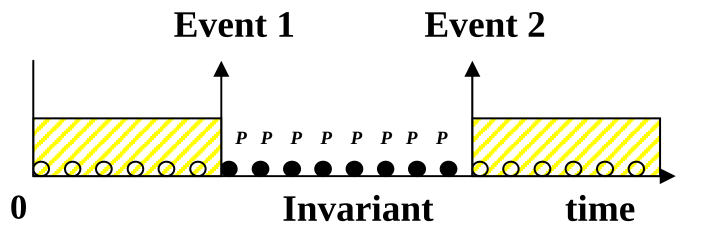
Power of Assertion (3/3)

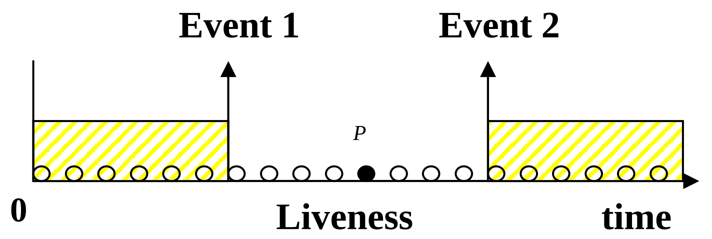
More evidences

- 17% of bugs were identified by assertions on Cyrix M3 (p1) project [1998]
- 50% of bugs were identified by assertions on Cyrix M3 (p2) project [1998]
- 85% of all bugs were found using OVL assertions on HP [2000]
- 400 bugs (Intel) were found from formal proofs of assertions [2001]

Assertion Types

- Invariant
  - assert_never(ck,event1, expression, event2)
  - assert_always(ck,event1, expression, event2)
- Liveness
  - assert_eventually(...)
  - assert_eventually_always(...)
- Other
  - assert_one_hot(...)
  - event_monitor(...)





Open Verification Library (OVL)

- Free download from www.verificationlib.org
  - Verilog, VHDL and PSL flavors

```
assert_always
assert_change
assert_decrement
assert_delta
assert_even_parity
assert_increment
assert_handshake
assert_never
assert_no_overflow
assert_no_transition
assert_no_underflow
assert_odd_parity
assert_one_hot
assert_proposition
assert_range
assert_time
assert_transition
assert_unchange
assert_win_change
assert_win_unchange
assert_window
assert_zero_one_hot
```


Tian-Sheuan Chang

Assertion-based Verification

- Assertion
  - Design assumption and properties
    - "input should range from 0 to 240"
    - "after **req** raises, **gnt** is expected within 10 clock cycles"
  - Break the simulation when assertion fails
  - Both the spatial and temporal relationship can be asserted
  - Help designers to locate bugs at right place and time
- Approaches
  - Library based
    - Open verification library. www.verificationlib.org
  - Language based
    - PSL (Sugar), System Verilog DAS (OVA)
- On average, 1 line in assertion language = 50 lines in Verilog
- Concept extended to functional monitors and functional coverage

Improving Verification with Assertions

- New Designs:
  - Capture requirements and assumptions while writing HDL
  - Use assertions to validate signal assumptions throughout the design process, e.g., block -> system transition
- IP / Design Reuse
  - Assertions validate correct stimulation of IP within system
    - Travel with IP
    - Provide immediate feedback to IP users
    - Reduce support calls to IP vendors
    - Document behavior and expectations

Tian-Sheuan Chang

Benefits of Assertion-Based Verification

- Reduces debugging time
  - Assertions can continuously monitor internal signals in the design, catching violations early in the design process
- Documents design
  - Assertions can be used to capture designer's intent
- Monitors I/O
  - Assertions can be used to verify protocols
- Improves design quality
  - Enables comparing the design specification with the circuit - throughout the design process
  - Assertions can be thought of as a "partial specification" for your design

Tian-Sheuan Chang

Outline

- Verification challenges
- Verification process
- Verification tools
- *RTL logic simulation*
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Tian-Sheuan Chang

Fast Simulation

- How to make simulation more productive ?
  – Make simulation more **efficient**
    - Coding style, faster workstation, hardware accelerator
  – Make simulation more **effective**
    - Code coverage, functional coverage, ABV

Fast Simulation Principle

A design project must include tailored RTL to achieve the fastest simulation possible.

Tian-Sheuan Chang

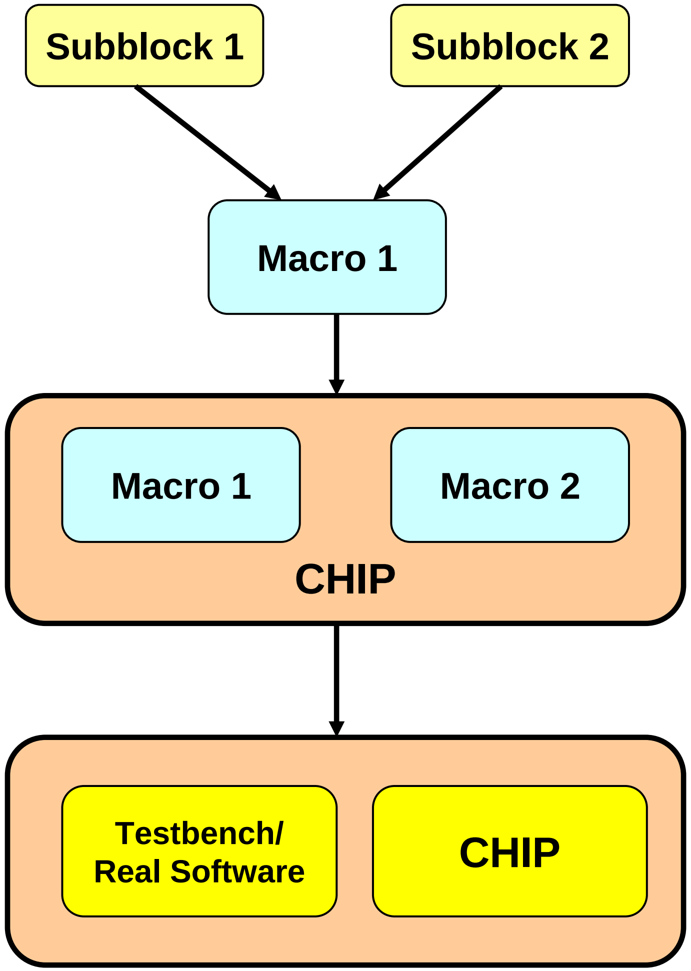
RTL Logic Simulation

- Noble goal - eliminate all design errors before silicon
- Realistic goal - achieve **self test** on first silicon

Project simulation phases

- Debugging
  - Full accessibility, fast turnaround time
- Performance profiling
  - To accelerate the simulation
    - Log files over networking, large log files
    - Bad memory allocated policy
- Regression
  - **Efficiency is the king**
    - Cycle-based, 2-state simulation
- Recreating hardware problems
  - Simulation debug & regression

Choosing Simulation Tools



Subblock module test stage
- Interpreted, event-driven simulator (VSS, Verilog-XL, VCS)

Block-level Integration stage
- Compiled, event-driven simulator or cycle-based simulator

Chip-level Integration stage
- Cycle-based simulator start
- Modules can migrate to emulation when relatively bug-free
- Testbench migrates to emulation last

Software testing stage
- Emulation
- Chip and testbench are in the emulator for max performance

Difference in Different Modes

	Debugging Phase	Regression Phase
Verilog Compilation	Std Vendor Model	Cycle-based Model
Signal Accessibility	Full	Limited
Waveform Viewing	Frequent	Seldom
Logging Output	Full	Limited
PLI C/C++	Debug mode ON	Debug mode OFF

Visit Minimization

- Visit **buses** instead of bits
- Bypass evaluation visits to intermediate logic not in an active path
  - Use condition like `if()`
- Eliminate event visits by using cycle-based evaluation

Visit Minimization Principle

For best simulation (and any EDA tool) performance, minimize the frequency and minimize granularity of visits.

Tian-Sheuan Chang

2-State Simulation

2-state methods in place of X

- **Zero-initialization** finds bugs that X can't
  - Due to X-state optimism
  - For more robust power-up verification
- **Random initialization** should do better
  - Capability to regenerate the specific random sequence
  - Keep the random seed
- Transform Z's at tri-state boundaries to **random 2-state** values
- 2-state simulates **faster** than 4 state

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- *RTL formal verification*
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Tian-Sheuan Chang

RTL Formal Verification

- Increasingly complex systems require more time to verify functionality
- Process of verifying design transformations should be automated
- **Orthogonal Verification Principle**
  - Separate verification of *circuit equality* vs. *circuit functionality*
- Coding techniques to facilitate formal verification

Tian-Sheuan Chang

Equivalence Checking

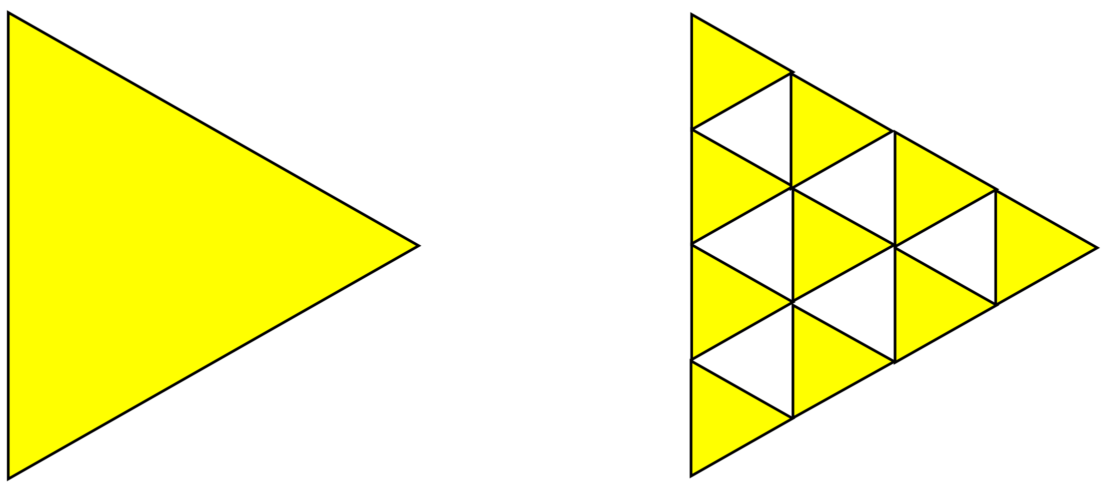
- Checking after
  - Synthesis
  - Scan chain insertion
  - Clock-tree synthesis
  - Manual modification
  - Place and route
  - ECO
- Equivalence checking for large designs
  - Tough due to exponential-of-input size nature
  - Logic cone partitioning is required

Tian-Sheuan Chang

Cutpoint

- Internal cross-design signal equivalence pairs are referred as cutpoint
- Partition large cones of logic into smaller cones for the proof

Cutpoints



Functional complexity Isolation

```
// Not so good Cutpoints
assign c_indx = (((coord_x * coord_y) & indx_mask) + indx_offset);
```

```
// Better Cutpoints
mult_16x15 mult1 (coord_x, coord_y, mult_prod);
assign c_indx = ((mult1_prod & indx_mask) + indx_offset);
```

Cutpoint Identification Principle

A single design decision pertaining to functional complexity must be isolated and localized within a module to facilitate equivalence checking cutpoint identification

IP Verification

Tian-Sheuan Chang

Test Expression Observability (1/2)

Tian-Sheuan Chang

Test Expression Observability Principle

Complex test expressions within a Verilog `case` or `if` statement must be factored into a variable assignment.

Test Expression Observability (2/2)

```
// Not so good
case ((a & b | c ^ d) || mem[idx])
  4'b0100: c_nxt_st = r_nxt_st << 1;
  4'b1000: c_nxt_st = r_nxt_st >> 1;
  default: c_nxt_st = r_nxt_st;
endcase;
```

```
//Good
c_nxt_st_test = (a & b | c ^ d) || mem[idx];
case (c_nxt_st_test)
  4'b0100: c_nxt_st = r_nxt_st << 1;
  4'b1000: c_nxt_st = r_nxt_st >> 1;
  default: c_nxt_st = r_nxt_st;
endcase;
```


Tian-Sheuan Chang

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- *Verifiable RTL – good stuff*
- Verifiable RTL – bad stuff
- Testbench design
- SOC verification

Tian-Sheuan Chang

Why Verifiable RTL

- A lot of guideline for reuse and synthesis exists
- **Lack** of RTL coding guidelines to optimize the verification process
- This vacuum becomes a problem as:
  - Design complexity increases
  - Advance verification processes are considered
    - Cycle-based simulation, 2-state simulation, property checking, equivalence checking, emulation
- **Verifiable RTL style** consists of
  - A verifiable subset of Verilog
  - A set of RTL coding guidelines
  - A set of fundamental principles

Tian-Sheuan Chang

RT-Level X-State Optimism (1/2)

- **Optimism – State Machine**

```
case (d)
 2'b00 :   e = 2'b01;
 2'b01 :   e = 2'b11;
 2'b10 :   e = 2'b10;
 default : e = 2'b00;
endcase
```

- If d == 2'bXX, case statement always takes the default branch!
- Alternate branches never test during startup!

RT-Level X-State Optimism (2/2)

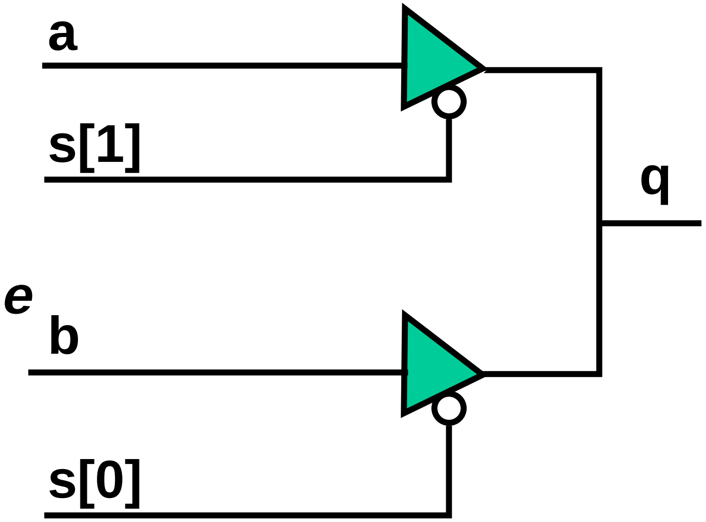
- **Accuracy Impractical**

```
case (d)
   2'b00 :     e = 2'b01;
   2'b0X :     e = 2'bX1;
   2'b01 :     e = 2'b11;
   2’bX0 :     e = 2’bXX;
   2'bX1 :     e = 2'bXX;
   2'b11 :     e = 2'b00;
   2'b1X :     e = 2'bX0;
   2'b10 :     e = 2'b10;
   2'bXX :     e = 2'bXX;
 endcase
```

X? In Real World

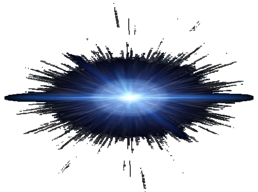
```
module mux (a,b,s,q);
output q;
reg a, b, q;
reg [1:0] s;
always @(a or b or s)
begin
   case (s)//synopsys full_case
      2'b11: q = 1'bz;
      2'b01: q = a;
      2'b10: q = b;
    endcase
end
endmodule
```

There are no X's
in the real circuit!



If s[1] = 0 and s[2] = 0

we might be SMOKING!



Semantic Mismatch

IP Verification

Tian-Sheuan Chang

How Slow Can Your Simulation Go?

```
for (i=0; i<64; i=i+1) begin
    bit5 = (i > 31);
    bit4 = (i > 15) && (i < 32) || (i > 47);
    bit3 = (i > 7) && (i < 16) || (i > 23) && (i < 32) || (i > 39) && (i < 48) || (i > 55);
    bit2 = (i > 3) && (i < 8) || (i > 11) && (i < 16) || (i > 19) && (i < 24) || (i > 27) && (i < 32) ||
            (i > 35) && (i < 40) || (i > 43) && (i < 48) || (i > 51 ) && (i < 56) || (i > 59);
    bit1 = (i == 2) || (i == 3) || (i == 6) || (i == 7) || (i == 10) || (i == 11) || (i == 14) || (i == 15) ||
            (i == 18) || (i == 19) || (i == 22) || (i == 23) || (i == 26) || (i == 27) || (i == 30) ||
            (i == 31) || (i == 34) || (i == 35) || (i == 38) || (i == 39) || (i == 42) || (i == 43) ||
            (i == 46) || (i == 47) || (i == 50) || (i == 51) || (i == 54) || (i == 55) || (i == 58) ||
            (i == 59) || (i == 62) || (i == 63);
    bit0 = (i == 1) || (i == 3) || (i == 5) || (i == 7) || (i == 9) || (i == 11) || (i == 13) || (i == 15) ||
            (i == 17) || (i == 19) || (i == 21) || (i == 23) || (i == 25) || (i == 27) || (i == 29) ||
            (i == 31) || (i == 33) || (i == 35) || (i == 37) || (i == 39) || (i == 41) || (i == 43) ||
            (i == 45) || (i == 47) || (i == 49) || (i == 51) || (i == 53) || (i == 55) || (i == 57) ||
            (i == 59) || (i == 61) || (i == 63);
    tmp [i] = pd [i] && (bit5 ~^ cell[5]) && (bit4 ~^ cell[4]) && (bit3 ~^ cell[3]) && (bit2 ~^
            cell[2]) && (bit1 ~^ cell[1]) && (bit0 ~^ cell[0]);
end // for
hit = | tmp ;
```


Better Ways for Speed

Tian-Sheuan Chang

parallel mask fashion for more parallelism -- 1000x faster

```
tmp= pd  & (~(64'hffffffff00000000  ^ {64{cell[5]}}))
         & (~(64'hffff0000ffff0000  ^ {64{cell[4]}}))
         & (~(64'hff00ff00ff00ff00  ^ {64{cell[3]}}))
         & (~(64'hf0f0f0f0f0f0f0f0  ^ {64{cell[2]}}))
         & (~(64'hcccccccccccccccc  ^ {64{cell[1]}}))
         & (~(64'haaaaaaaaaaaaaaaa  ^ {64{cell[0]}}));

hit = | tmp;
```

bit-indexing for more and more parallelism -- 3000x faster

```
hit = pd[cell];
```

Verifiable Subset

- Two ways of constructed a design
  - to make it so ***simple*** that there are obviously ***no deficiencies***
  - to make it so ***complicated*** that there are ***no obvious deficiencies***
- However, synthesizer vendors tend to enlarge the synthesizable subset
- Where there are 2/3/4 ways to express the same thing in RTL
  - PICK Simple ONE, Simple wins in verification

Verifiable Subset Principle

A design project must select a simple HDL verifiable subset, which serves all verification tools within the design flow as well as providing an uncomplicated mechanism for conveying clear functional intent between designers

Tian-Sheuan Chang

Verifiable Verilog Keyword

- Verifiable subset is a subset of synthesizable subset
- 27 out of the 102 Verilog-1995 keywords
- “for” looping construct could be used for extra exception

always	else	initial	parameter
assign	end	inout	posedge
begin	endcase	input	reg
case	endfunction	module	*tri*
casex	endmodule	negedge	*tri0*
default	function	or	*tri1*
	if	output	wire

IP Verification

Tian-Sheuan Chang

Unsupported Operators

operator	example	function
-	-a	unary minus
*	a * b	multiply
/	a / b	divide
= = =	a = = = b	equality (0/1/X/Z)
! = =	a ! = = b	inequality (0/1/X/Z)

IP Verification

Tian-Sheuan Chang

Asynchronous Principle

- Asynchronous - not addressed by RTL verification. Requires:
  - Protocol verification - Petri net modeling
  - Failure rate analysis - Circuit analysis

Asynchronous Principle

A design project must **minimize and isolate** resynchronization logic between asynchronous clock domains.

Tian-Sheuan Chang

Combinational Feedback Principle

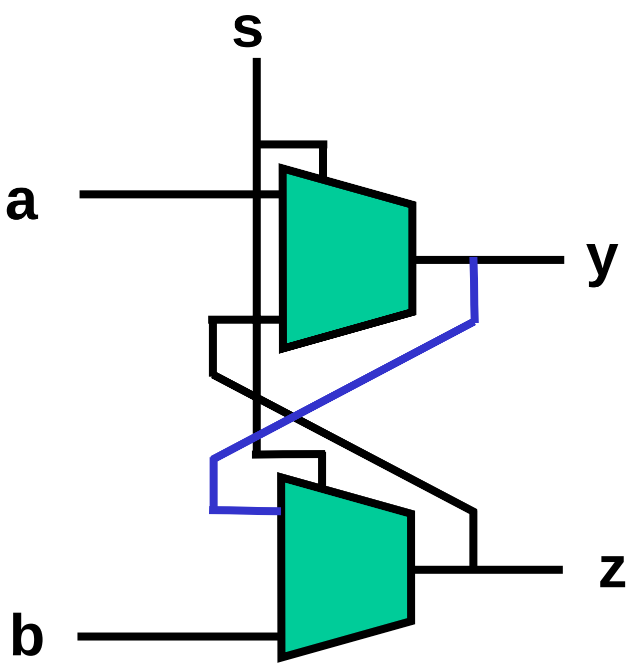
- Forms of Feedback
  - Design errors
  - False path
  - Apparent

Combinational Feedback Principle

Designers must not use any form of combinational logic feedback (real, false-path, apparent) in their Verilog.

Tian-Sheuan Chang

False Path




```
module m (s, a, b, y, z);
  input s;
  input a, b;
  output y, z;
  wire s, a, b;
  wire y, z;
  assign y = s ? a : z;
  assign z = s ? y : b;
endmodule
```

IP Verification

Tian-Sheuan Chang

Apparent Feedback

```
module m (a, d);
   input a;
   output d;
   reg b, d;
   wire c;
always @(a or c)
   begin
      b = a;
      d = c;
   end
   assign c = b;
endmodule
```

```
module m (a, d);
   input a;
   output d;
   reg b, c, d;
always @(a)
   begin
      b = a;
      c = b;
      d = c;
   end
endmodule
//order dependent
```

Fix 1

```
module m (a, d);
   input a;
   output d;
   wire b, c, d;
   assign b = a;
   assign d = c;
   assign c = b;
Endmodule
//order
//independent
```

Fix 2

IP Verification

Tian-Sheuan Chang

Verifiable case/casex

- case/casex practices supporting verifiable RTL
  - Fully specified case/casex statements
  - Consistent test signal and constant widths

Tian-Sheuan Chang

Fully specified case/casex

- Pros
  - Faster boolean equivalence checking
    - No don't care conditions
  - RTL - Gate-level simulation alignment
  - Improved RTL simulation:
    - Performance (no X state)
    - Verification - startup state, fault simulation
  - RTL Manufacturing test simulation
- Cons
  - Worse synthesis result (not always true)
    - Alternative solution exists
  - Loss of simplicity
    - Alterative solution has more complicated coding style

case/casex – Verification vs Synthesis

```
module one_hot(c_hot,c_code);
  input [7:0] c_hot;
  output [2:0] c_code;
  reg [2:0] c_code;
always @ (c_hot) begin
  case (c_hot) // synthesis full_case, for synthesis?
    8'b10000000: c_code = 3'b000;
    8'b01000000: c_code = 3'b001;
    8'b00100000: c_code = 3'b010;
    8'b00010000: c_code = 3'b011;
    8'b00001000: c_code = 3'b100;
    8'b00000100: c_code = 3'b101;
    8'b00000010: c_code = 3'b110;
    8'b00000001: c_code = 3'b111;
    default:     c_code = 3'b000; // or for verification
    endcase
end // always (c_hot)
endmodule // one_hot
```

If default case is used, 3X gates are generated

Partially-Specified to Fully-Specified

- For smaller case statements
  - minimization savings not worth loss of verifiability
- For larger case statements –
  - use alternative (fully specified) coding style

case/casex – Alternative for One-Hot

```
module one_hot(c_hot,c_code);
  input [7:0] c_hot;
  output [2:0] c_code;
  reg [2:0] c_code;
  reg [2:0] c_code0,c_code1,c_code2,c_code3;
  reg [2:0] c_code4,c_code5,c_code6;
always @ (c_hot) begin
  c_code6 = (c_hot [6]) ? 3'b001 : 3'b000;
  c_code5 = (c_hot [5]) ? 3'b010 : 3'b000;
  c_code4 = (c_hot [4]) ? 3'b011 : 3'b000;
  c_code3 = (c_hot [3]) ? 3'b100 : 3'b000;
  c_code2 = (c_hot [2]) ? 3'b101 : 3'b000;
  c_code1 = (c_hot [1]) ? 3'b110 : 3'b000;
  c_code0 = (c_hot [0]) ? 3'b111 : 3'b000;
  c_code = c_code0 | c_code1 | c_code2 | c_code3 |
           c_code4 | c_code5 | c_code6;
end // always (c_hot)
endmodule // one_hot
```


Tian-Sheuan Chang

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- *Verifiable RTL – bad stuff*
- Testbench design
- SOC verification

X-state Pessimism

X-state Pessimism - arithmetic

```
reg [15:0] a,b,c;
…
begin
   b = 16'b0000000000000000;
   c = 16'b000000000000X000;
   a = b + c;
   $display(" a = %b",a);
end
```

a = 16'bXXXXXXXXXXXXXXXX

IP Verification

Tian-Sheuan Chang

X-state Optimism - case Statement

```
reg [1:0] d,e;
…
begin
   case (d)
        2'b00 :   e = 2'b01;
        2'b01 :   e = 2'b11;
        2'b10 :   e = 2'b10;
        default : e = 2'b00;
   endcase
   $display(" e = %b",e);
end
```

- If d contains an X then e = 2'b00
- RTL simulation will miss verifying alternate branches (especially at the start-up sequences)

Tian-Sheuan Chang

Accuracy impractical

- Simulation performance.
- Labor content.
  - Added X-state tests
  - branch to boolean conversion
- Complex verification
- Completeness
- Synthesis

Tian-Sheuan Chang

Prohibit X for "don't care's"

```
...
case (select)
  2'b01 :   mux = b;
  2'b10 :   mux = c;
  default : mux = 2'bX;
endcase
```


Tian-Sheuan Chang

X in “don’t care’s”

- **Mask errors** which can’t be found at RT-level simulation
- Slows RT-level simulation
- Slows RTL-to-gate equivalence checking
- Causes **semantic mismatches** between RTL and gate-level simulation.

Tian-Sheuan Chang

Visit Minimization

- Criminals to degrade simulation performance
  - referencing bits instead of buses
  - Run-time configuration tests
  - loops.

Tian-Sheuan Chang

Bits v.s. Bus

BAD: Explicit bit visits

```
c_ecc_out_1 = c_in [10] ^
        c_in[11] ^ c_in[12] ^
        c_in[13] ^ c_in[14] ^
        c_in[15] ^ c_in[16] ^
        c_in[17] ^ c_in[18] ^
        c_in[19] ^ c_in[20] ^
        c_in[21] ^ c_in[22] ^
        c_in[23] ^ c_in[24] ^
        c_in[25] ^ c_in[26] ^
        c_in[27]^ c_in[28] ^
        c_in[32] ^ c_in[35] ^
        c_in[38] ^ c_in[39];
```

```
c_ecc_out_1 =
  ^ (c_in & 40'h003ffff893);
```

GOOD: Parallel value evaluation

IP Verification

Tian-Sheuan Chang

Run-Time Configuration

```
module fifo(
...
parameter WIDTH = 13;
parameter DEPTH = 32;
parameter ENCODE = 0;
...
function [31:0] encoder;
input [WIDTH-1:0] indata;
   begin
   if (ENCODE != 0) begin
      < calculate encode value based on indata >
      end
   else
      encoder = indata;
End
```

Use conditional compilation directives `` `if``, `` `else``, `` `elseif``, `` `endif`` instead

For-Loops

BAD: Individual bit visits, loop overhead

```
input [‘N-1:0] a;
output [‘N-1:0] b;
integer i;
reg [‘N-1:0] b;
always @ (a) begin
  for (i=0; i<=‘N-1; i=i+1)
     b[i] = ~a[i];
end
```

```
input [‘N-1:0] a;
output [‘N-1:0] b;
assign b = ~a;
```

GOOD: Parallel value evaluation

For-Loop: Bus Reversal

BAD:
For-loop

```
input [15:0] a;
output [15:0] b;
integer i;
reg [15:0] b;
always @ (a) begin
  for (i=0; i<=15;
        i=i+1)
      b[15 - i]] =
        ~a[i];
end
```

Better:
Concatenation

```
input [15:0] a;
output [0:15] b;
assign b = { a[0], a[1],
  a[2], a[3], a[4], a[5],
  a[6], a[7], a[8], a[9],
  a[10], a[11], a[12],
  a[13], a[14], a[15]  };
```

For Loop

- Simulate slow
  - from 10X to > 1000X slower than non-for loop versions.
- Synthesizes slow
- Memory clear
  - only legitimate for loop use in chip design.
- Avoid using the for loop whenever possible

Faithful Semantics

- Bad coding style - unequal design information
  - HDL simulator information not used in synthesis
  - Synthesis switches not used by simulator.
- X state

Faithful Semantics Principle

An RTL coding style and set of tool directives must be selected that insures semantic consistency between simulation, synthesis and formal verification tools.

Full_case & Parallel_case

- Fully-specify case/casex
  - Do not use full_case and parallel_case
- Eliminate case-item constant overlaps
- Find alternative coding if necessary
- Implement RTL priority encoder as multiplexer

Tian-Sheuan Chang

Verilog Initial Blocks

- Explicitly creates RTL-gate differences.
- Better - place in testbench
- Best - encapsulate within storage element (FF's memories) library modules

Careless Coding

- Incomplete sensitivity list
- Latch inference
- Incorrect procedural statement ordering

IP Verification

Tian-Sheuan Chang

Timing Problems

- Project-wide policy
- #0; delays
- Non-blocking assignment delay
- Testbench delays

Tian-Sheuan Chang

Race Condition

- Race

```
//file a.v
always @(posedge ck)
  begin
      b = a;
  end
//file b.v
always @(posedge ck)
  begin
      c = b;
  end
```

- No race

```
//file a.v
always @(posedge ck)
  begin
      b <= a;
  end
//file b.v
always @(posedge ck)
  begin
      c <= b;
  end
```

IP Verification

Tian-Sheuan Chang

Testbench Delays

- Testbench designers insert delays to offset timing with respect to clock edges for:
  – inserting control states
  – observing states
- Testbench timing often less disciplined than chip timing

User-Defined Primitive (UDP)

- Not RTL!
- Often preclude use of new RTL verification tools
- Sequential UDP's present special challenges

Just say no

Summary

- Code your RTL for synthesis and verification as well
- ***Verifiable RTL coding styles***
  - Prevent you from pitfalls in the verification process
  - Make you curse verification-related tools less
  - Increase the verification performance
  - Provide better verification outcome
  - ***Give you more robust design***

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- *Testbench design*
- SOC verification

Testbench Design (1)

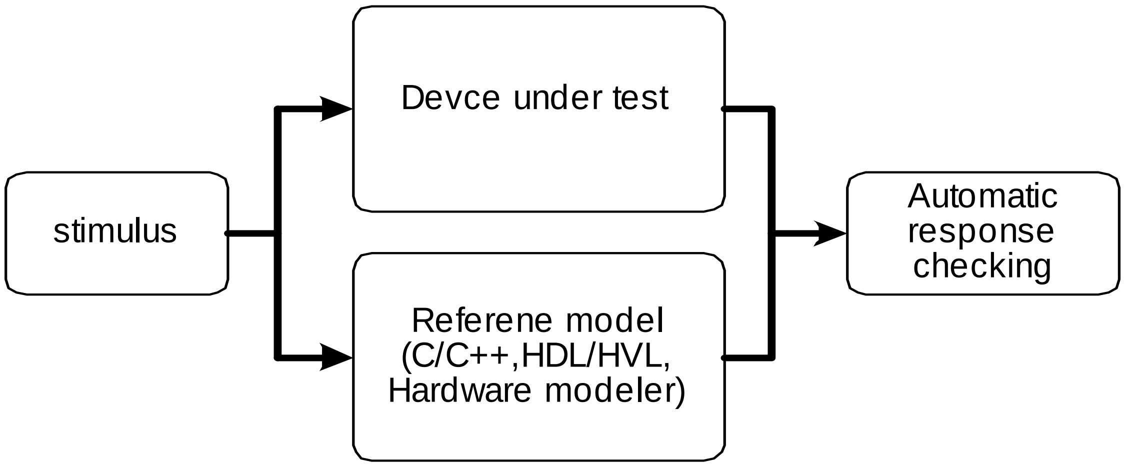
- The testbench design differs depending on the function of the macro
  - microprocessor macro, test program,
  - bus-interface macro, use bus functional models and bus monitors
- Subblock testbench

IP Verification

Tian-Sheuan Chang

Testbench Design (2)

- **Transaction-based** stimulus generation and response checking
  - Legal set of input
  - Corner case and random test
- **Auto or semi-auto stimulus** generation is preferred
- **Automatic response checking** is a must
  - Self-checking is recommended
  - Detect problems as early as possible
- **Reusable** testbench

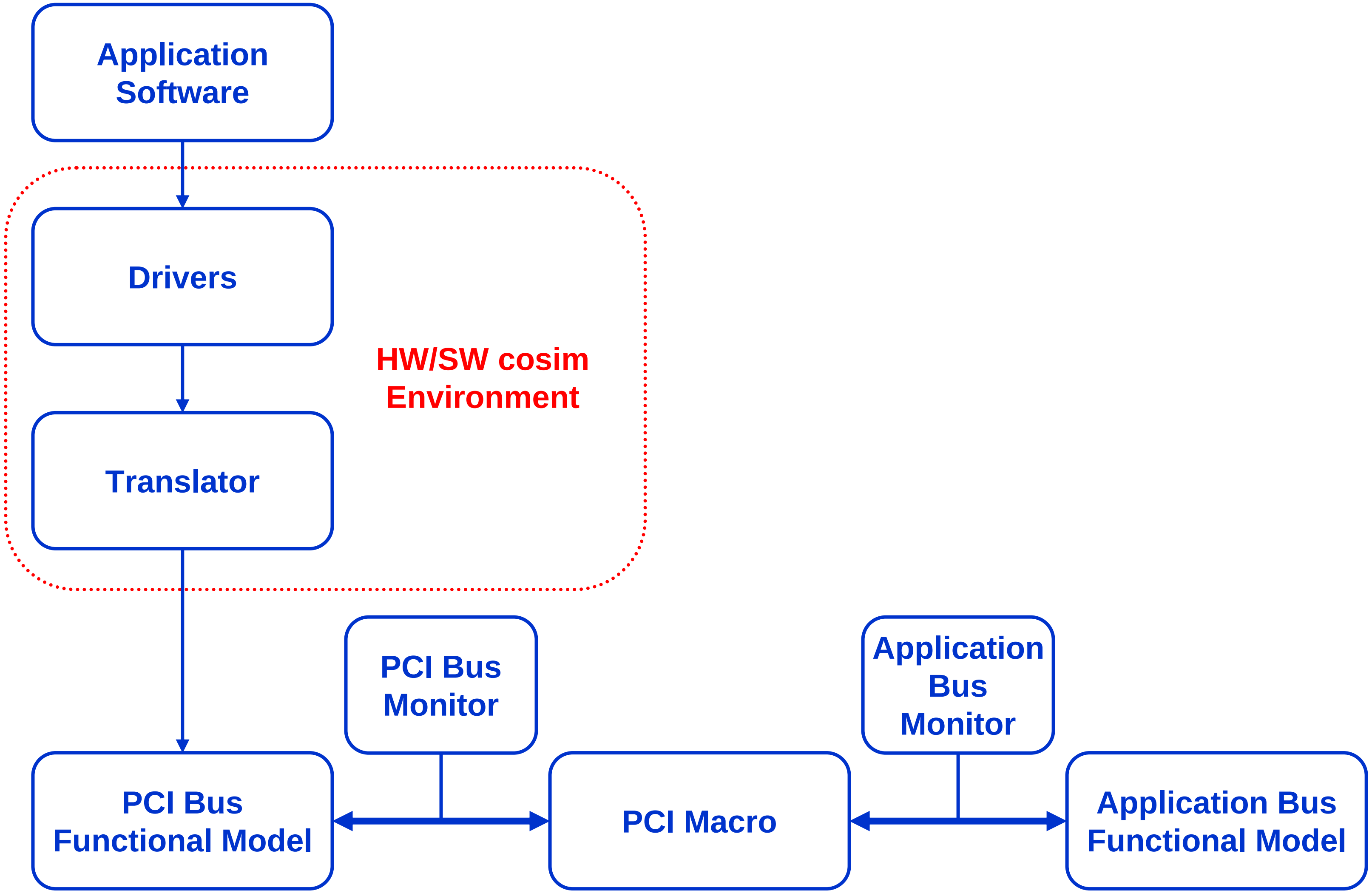


Testbench Authoring

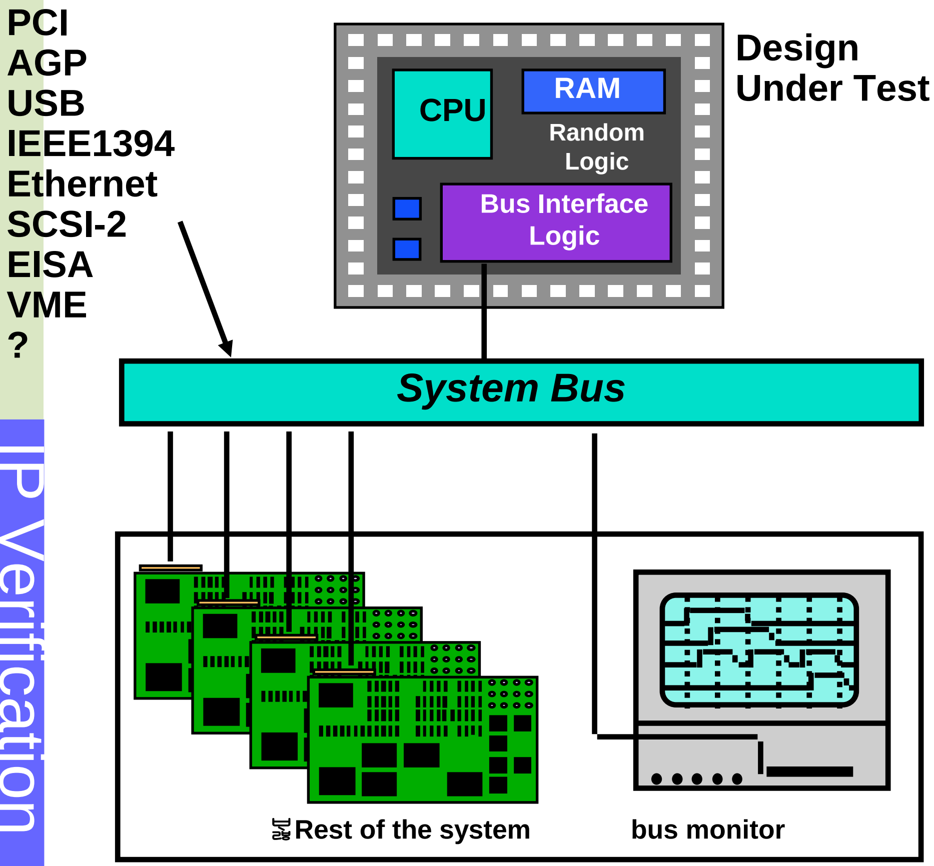
- An effective testbench
  - Concurrency
  - Encapsulation and abstraction
  - Self-checking
  - Automatic test stimulus generation
  - Reusable components
- Testbench authoring tools
  - **Partitioning** the responsibility among TVMs and tests
  - Specifying **cause and effect** relationships among transactions
  - Specifying **complex concurrency** using inter-transaction synchronization
  - Specifying **localized constraints** in the attributes of transaction

Tian-Sheuan Chang

Macro Testbench



Bus Functional Model



Bus Functional Model

Definition: ***Simulation model allowing designers to verify compliance to a particular specification prior to prototyping:***

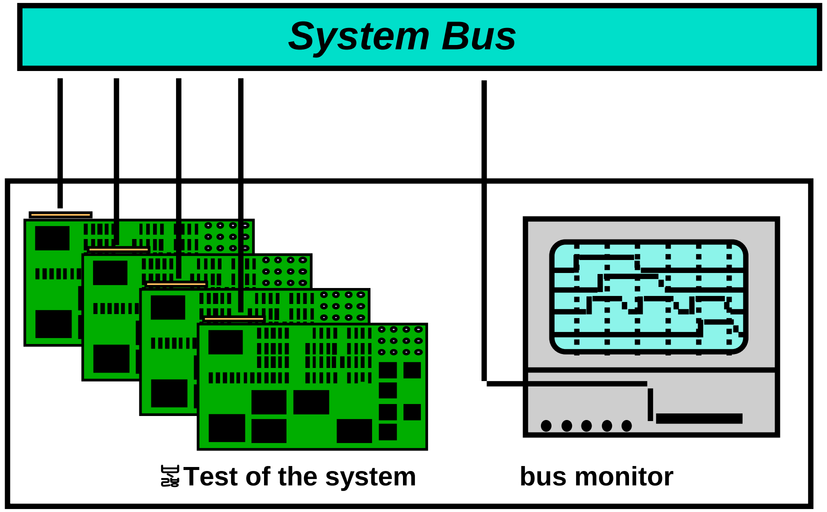
1. **Model the bus transactions on the bus, each read and write transaction is specified by the test developer**
2. **Monitors bus activity for protocol compliance**

IP Verification

Tian-Sheuan Chang

Benefits

- Test for interoperability during simulation
- Verify compliance prior to fabrication
- Generate test vectors more efficiently
- Learn a new bus faster
- BFM is written in RTL, C/C++, or testbench automation tools
  - Flexibility
  - Visibility into model operation

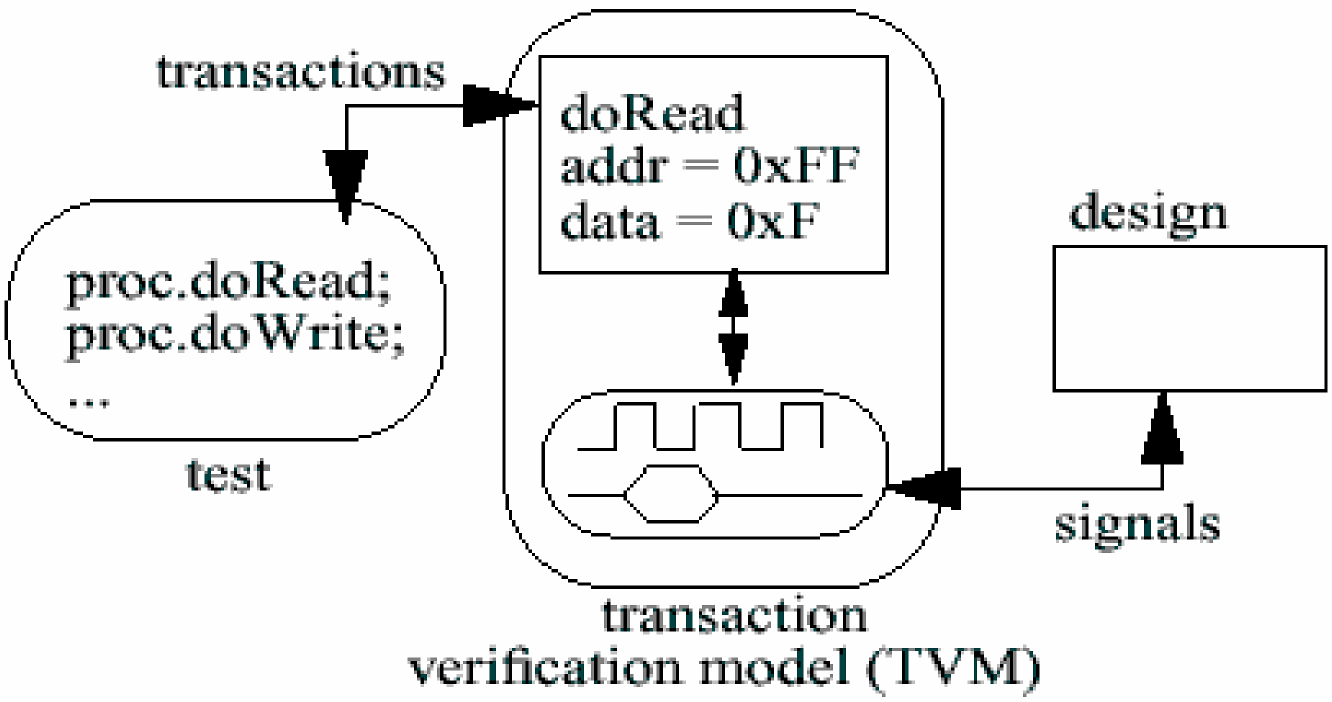


Verification Suite Design

- Once built the testbench , we can develop a set of tests to verify the correct behavior of the macro
- Test sets
  - functional testing
  - corner case testing
  - code coverage
  - random testing

Tian-Sheuan Chang

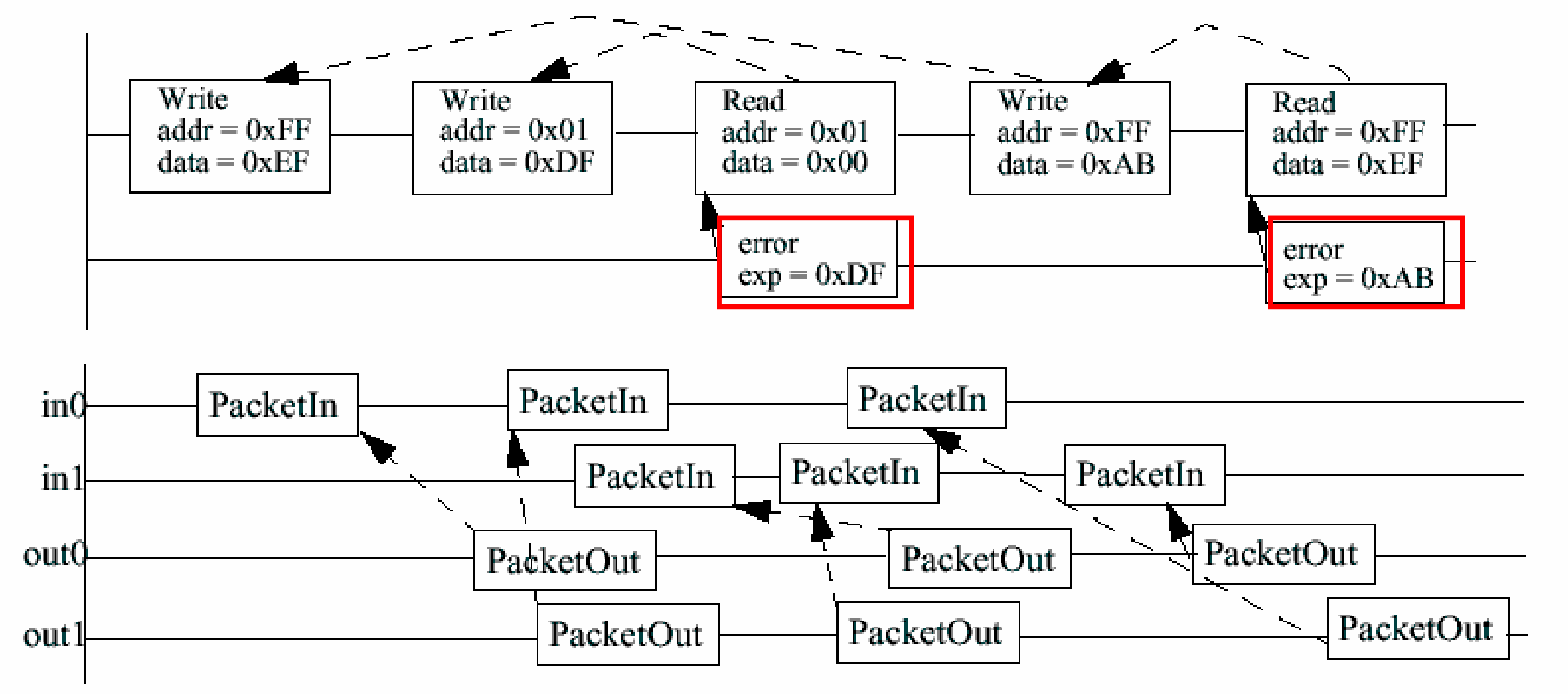
Transaction-based Verification



Efficient Simulating Debug (1)

- Transaction viewing
  – The abstract information about a transaction is displayed.
- Cause and effect
  – The relationships among transactions are displayed.
- Error transactions
  – An error detected during simulation is recorded.
- Concurrency
  – Out-of-order/pipelined transactions are displayed

Efficient Simulating Debug (2)



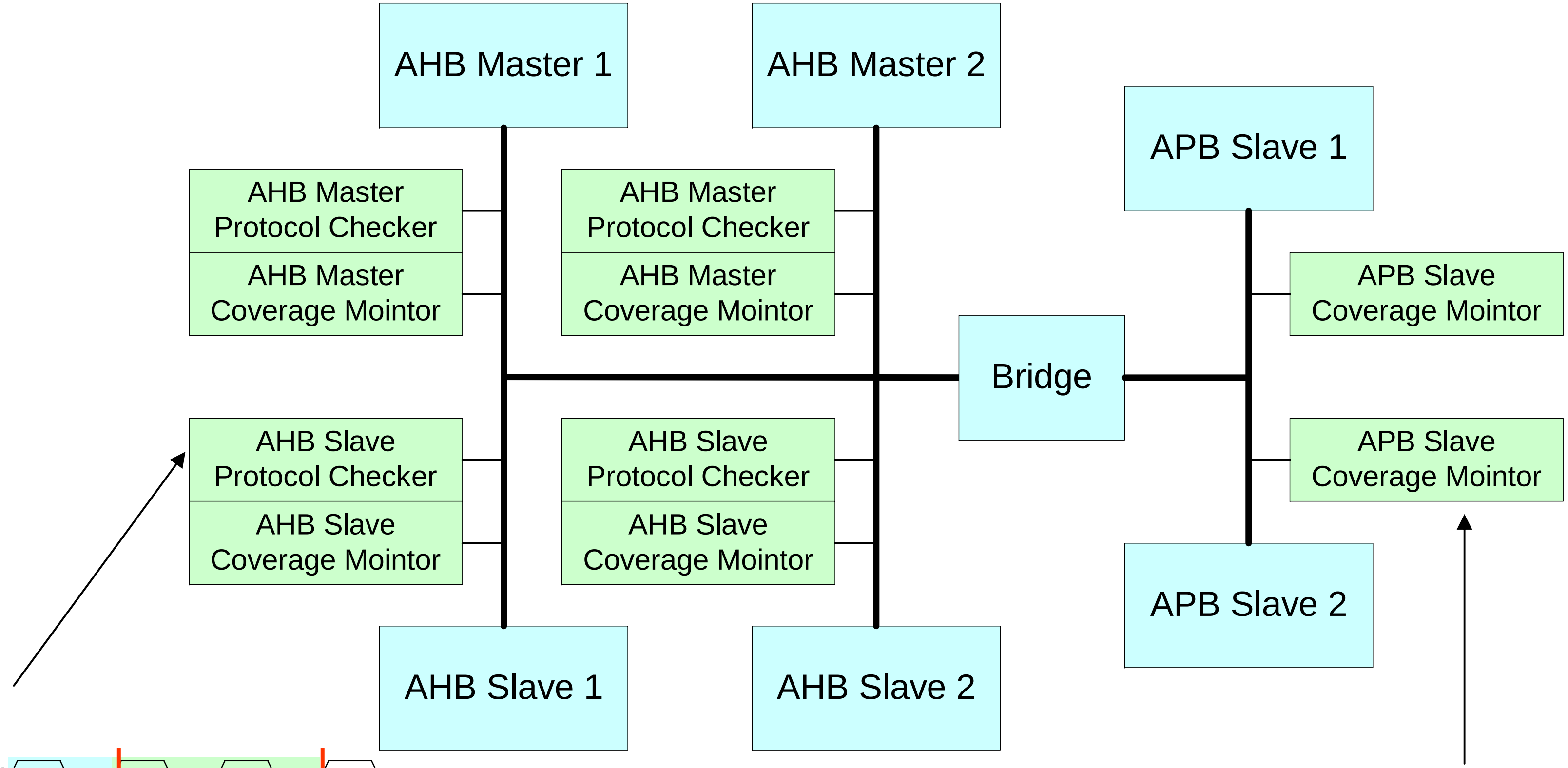
Behavioral Models

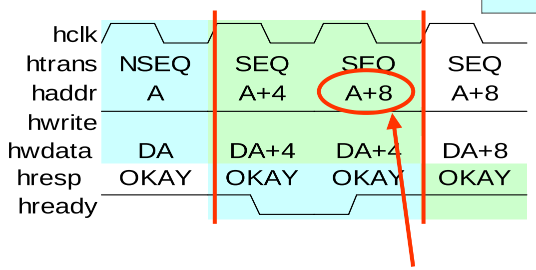
- Describe the black-box functionality of a design, required for all IPs
- Benefits
  - Audit of the specification
  - Development and debug of testbench in parallel with RTL coding
  - **System verification** can start earlier
  - It can be used as an **secure** evaluation and integration tool by your customer
  - Faster to write, debug, simulate and time-to-market
- Cost
  - Require additional resource to write the behavior model
  - Maintenance requires additional efforts
- BFM is required particularly for interface Ips
- ISA Model is required for processor IPs
- Commercial available for standard based IP
  - Verification IP, e.g. PCI, IEEE 1394, USB

Verification IP

- A package including well-designed and well verified BFM/monitor for a specific protocol/interface
  - AMBA, Ethernet, SONET, UTOPIA, PCI, USB, UART, CAN, ..
  - Avoid re-invent-the-wheel
  - Accelerate the verification

Verify AMBA System





**AHB Master Protocol violation:
Address was changed while in transfer extension (hready low).
Ref. Spec. section 3.4**

```
Transfer Response Summary
----------------------------------------
Response Type     |     Count
----------------------------------------
     OKAY         |       104
     ERROR        |       0
     RETRY        |       5
     SPLIT        |       13
----------------------------------------
```

Verification Support

- Protocol Checker
  - Monitor the transactions on an interface and check for any invalid operation
    - Embedded in the test bench
    - Embedded in the design
  - Error and/or warning messing of bus protocol
- Expected results checker
  - Embedded in the test bench
  - Checks the results of a simulation against a previously specified, expected response file.
- Performance monitor
  - Number of transfers, idle cycles...

Tian-Sheuan Chang

Simulation Management

- Pass or Fail ?
  - Produce a message that the simulation was terminated normally
- SDF Back-Annotation
  - Very time-consuming
  - Invoke the simulation once for multiple testcases
- Output File Management
  - A copy of output message: verilog.log
  - Dump waveform only for needed
  - Run multiple simulations in parallel
    - Use "-l" option to change the name of the output log file
    - Use script to help manage the configuration of a simulation and the name of output file

Tian-Sheuan Chang

Regression

- A regression suite ensures that modifications to a design remain **backward compatible** with previously verified functionality
- Regressions are run at regular intervals
- Provide a fast mode
- Regression Management
  - Simulation never terminate
  - Put a time bomb in all simulations to prevent simulation running forever
  - Success or failure of each testcase should be checked after regression test

Tian-Sheuan Chang

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- *IP Modeling*
- SOC verification

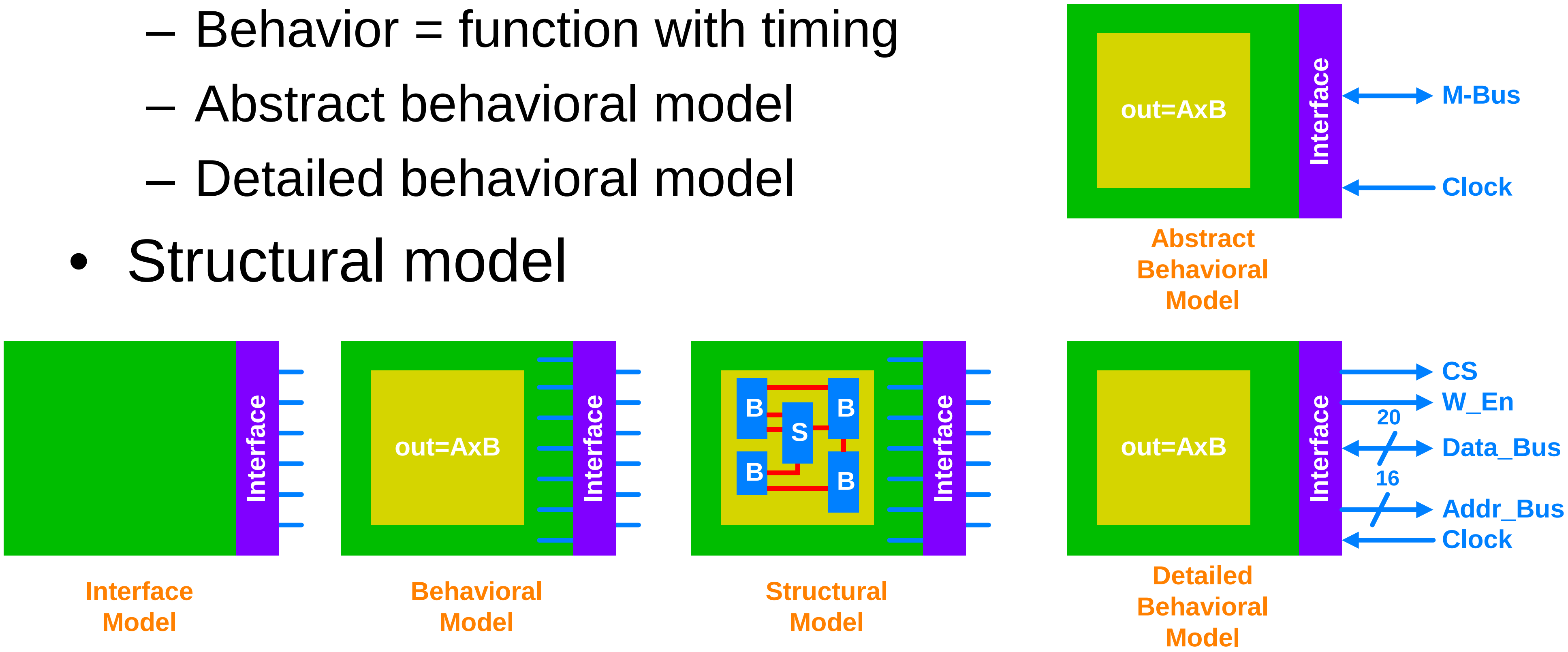
Tian-Sheuan Chang

The Intent of Different Level of IP Model

- Design exploration at higher level
  - Import of top-level constraint and block architecture
  - Hierarchical, complete system refinement
  - Less time for validating system requirement
  - More design space of algorithm and system architecture
- Simple and efficient verification and simulation
  - Functional verification
  - Timing simulation/verification
  - Separate internal and external (interface) verification
  - Analysis: power and timing
- Verification support: e.g., monitor, checker...

General Modeling Concepts

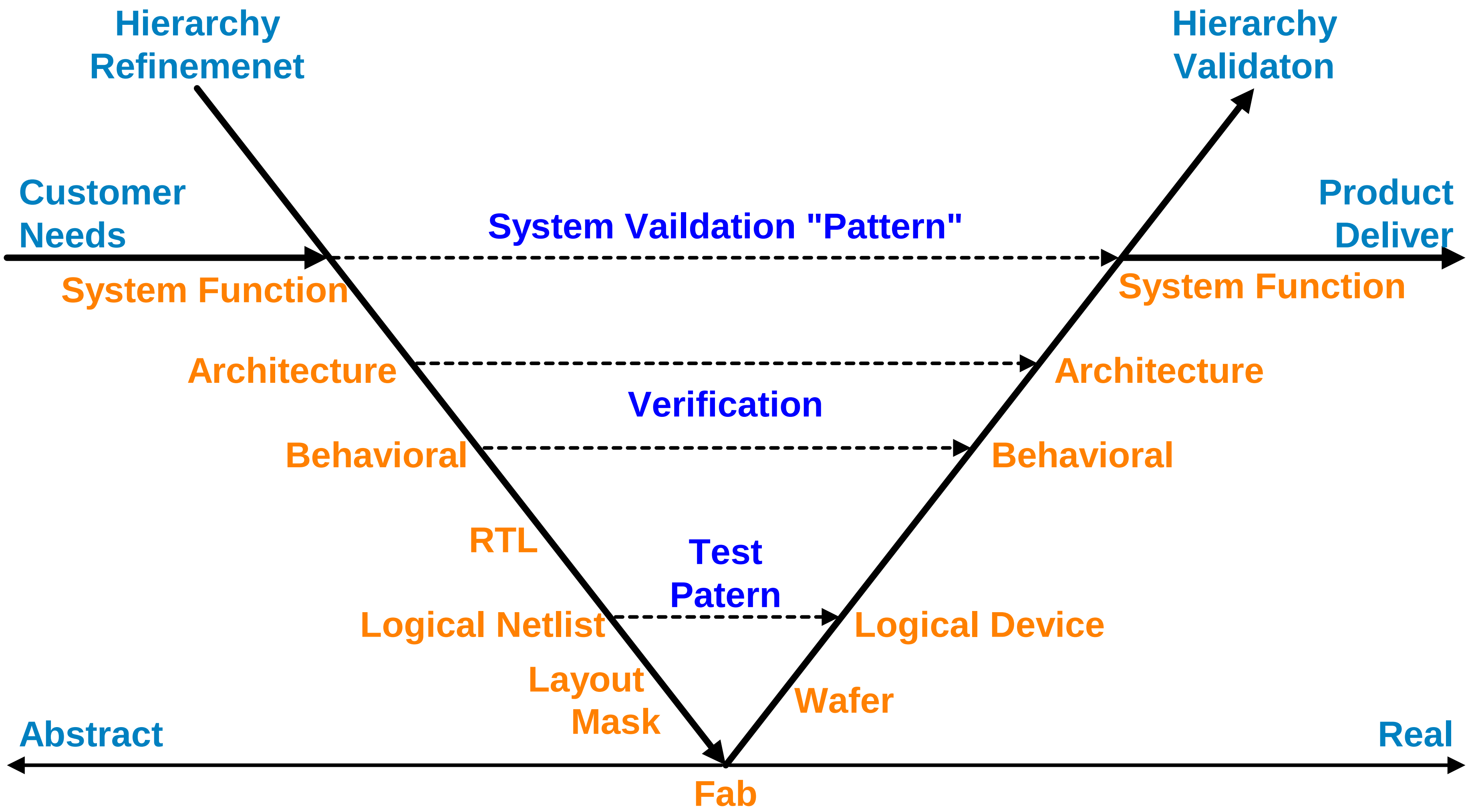
- Interface model
  - Synonym: bus functional, interface behavioral
- Behavioral model
  - Behavior = function with timing
  - Abstract behavioral model
  - Detailed behavioral model
- Structural model



Issues of IP Modeling

- Attributes
  - What is the sufficient set of model attributes?
  - How are these model attributes validated?
  - How is the proper application of an abstract model specified?
- Two important dimensions of time
  - Model development time is labor intensive: model reusability
  - Simulation time depends upon strategy chosen for mixed domain simulations

From Requirement to Delivery

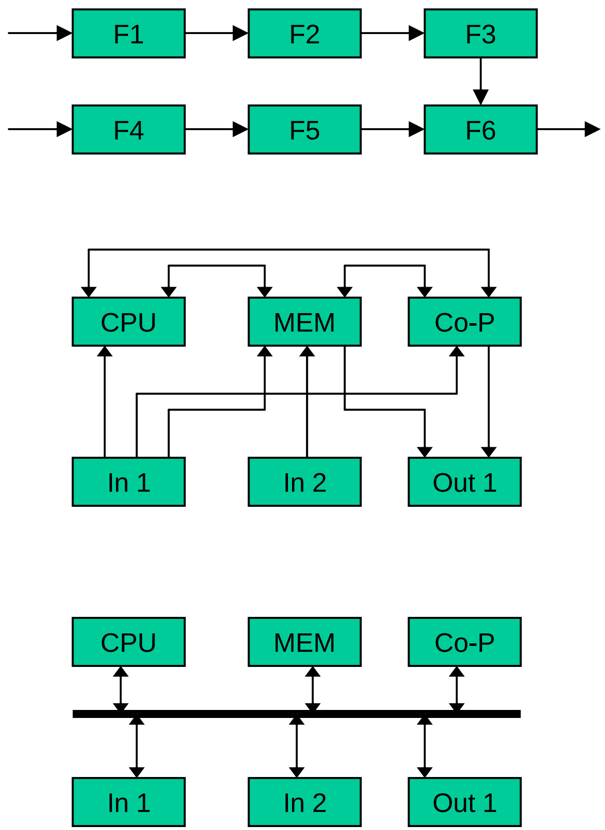


IP Verification

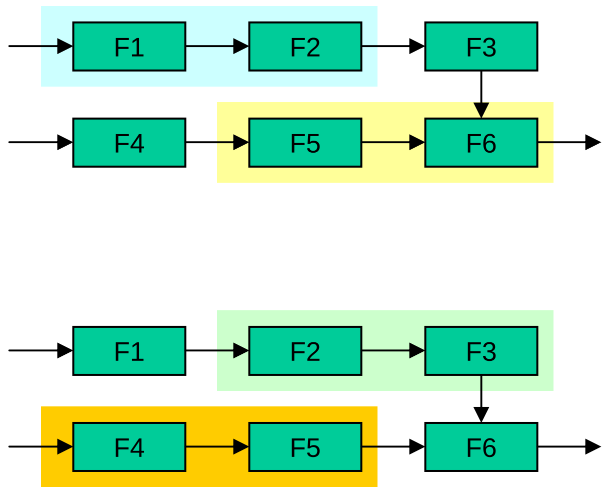
Tian-Sheuan Chang

Example: Hierarchical Design Refinement

Vertical refinement

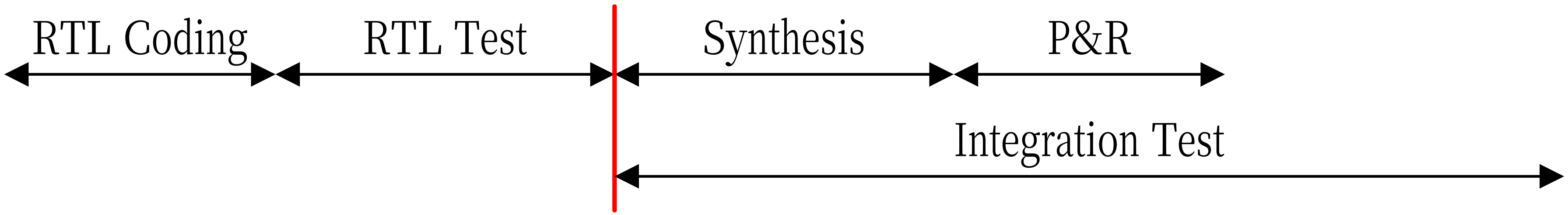


Horizontal refinement: Partition

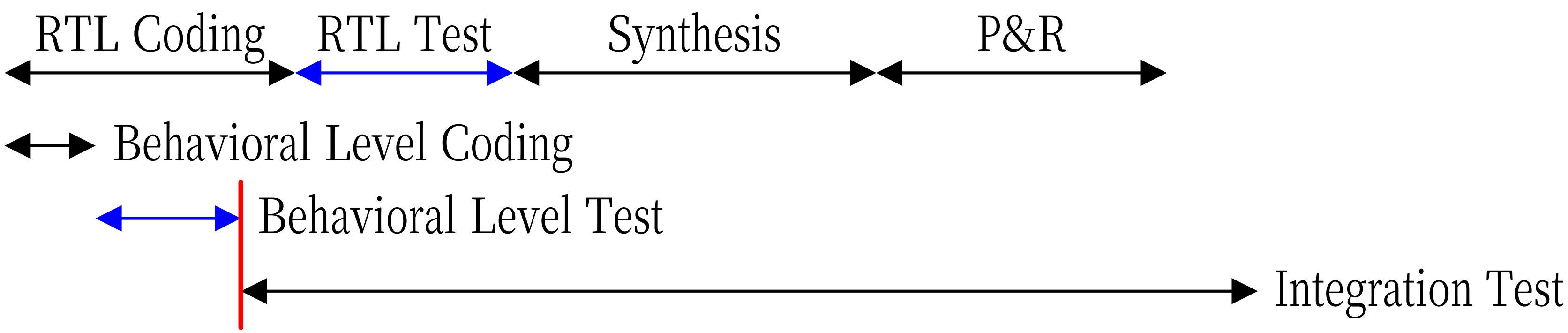


Example: Manage Size and Run-Time

Start at RTL



Start at behavioral level



IP Verification

Tian-Sheuan Chang

IP Modeling

General Modeling Concept

Primary Model Classes:
- Behavioral Model
- Functional Model
- Structural Model

Specialized Model Classes:
- Performance Model
- Interface Model
- Hybrid Model

Computational Model Classes:
- Data Flow Graph Model
- Other Models

System Models
- Executable Specification
- Mathematical-Equation Model
- Algorithm Model

Architecture Models
- Token-based Performance Model
- Abstract-Behavioral Model
- Data Flow Graph (DFG) Task Primitive
- Instruction Set Architecture (ISA) Model

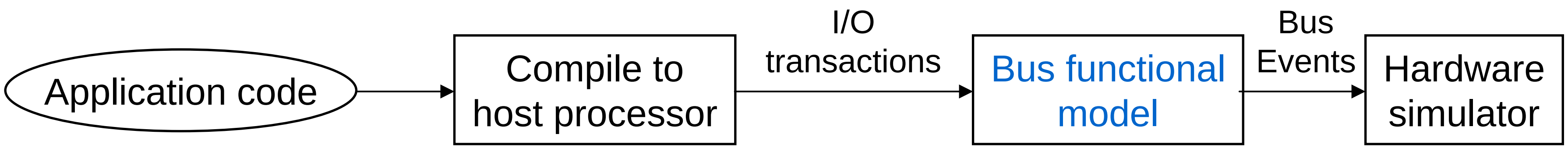
Hardware Models
- Detailed-Behavioral Model
- Register Transfer Level (RTL) Model
- Logic-Level Model
- Gate-Level Model
- Switch-Level Model
- Circuit-Level Model

Software Models
- Pseudo-Code
- High Level Language (HLL)
- Assembly code
- Micro-Code
- Object Code

Precision Axis
- Temporal Precision Axis
- Data Precision Axis
- Functional Precision Axis
- Structural Precision Axis
- Software Programming Precision Axis

CPU Model

- CPU model enable
  - Estimate software performance
  - Analyze system trade offs
- CPU model
  - Bus functional model



  - Instruction set simulator (ISS)
    - Instruction accurate
    - Cycle accurate
  - Virtual processor model (Cadence VCC technology)

ARM Modeling (1/4)

Concept

↓

Silicon

- System model
- Instruction set simulators (ISS)
- Co-verification model
- Bus Interface model
- Behavioral/RTL model
- Design signoff models
- Hardware modeling
- Gate Level netlist model

ARM Modeling (2/4)

- System Model
  - Provision of customized Software Debugger/ARMulator packages, suitable for dataflow simulation environments.
  - Cadence Signal Processing Worksystem (SPW) and Synopsys COSSAP Stream Driven Simulator
- Co-verification model
  - Each ARM processor core contains a co-verification simulator component and a bus interface model component
  - Co-verification simulator: combines the properties of an advanced ISS with the bus cycle accurate pin information capability required to drive a hardware simulator
  - CoWare N2C Design System, Synopsys Eaglei, to name a few.

ARM Modeling (3/4)

- Bus interface models (BIM)
  - Run a list of bus transactions to stimulate simulated hardware under test
  - Allowing the designer to concentrate on the hardware design without waiting for the ARM control software to be developed.
  - Generated using ModelGen
- Design signoff models
  - Full architectural functionality and full timing accurate simulation
  - Accept process specific timing and back annotated timing
  - Used to ‘sign off’’ design before committing silicon
  - Be compiled 'C' code which enables protection of the inherent IP and superior simulation execution speed over pure HDL models
  - Generated using ModelGen

ARM Modeling (4/4)

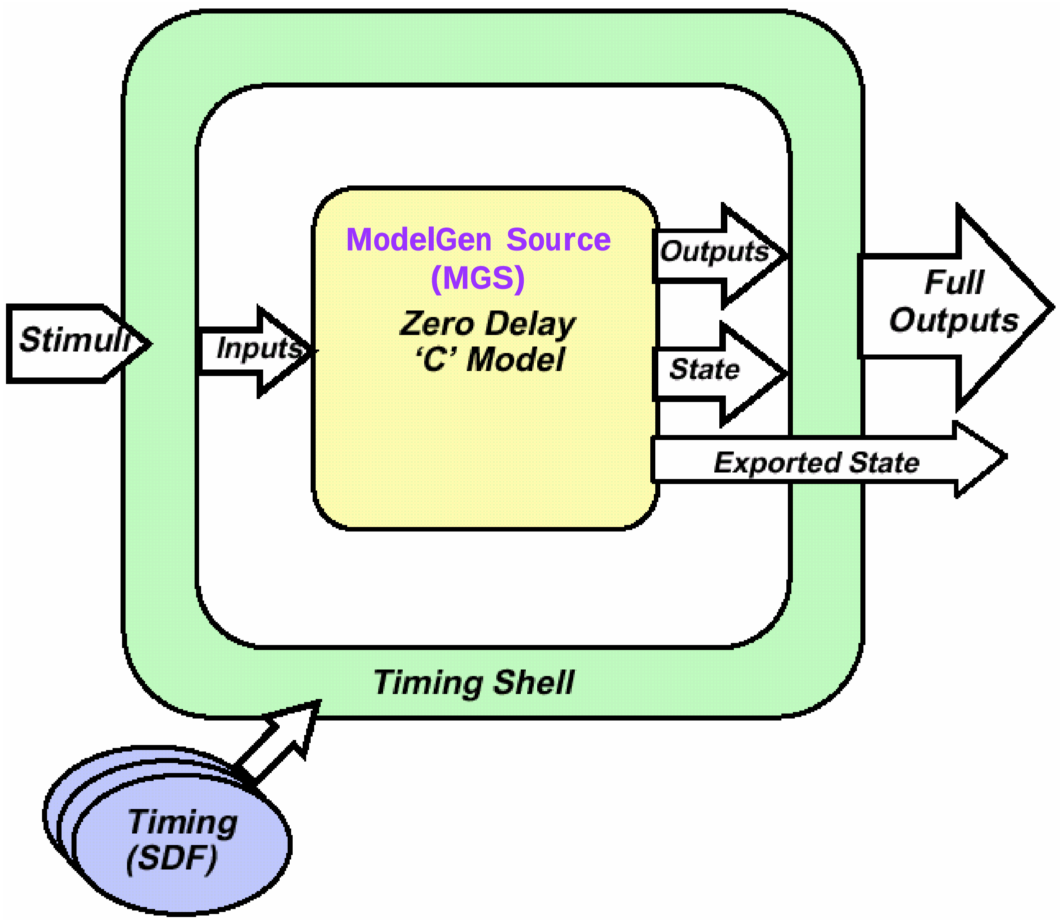
- Hardware Modeling
  - Real chip-based products, based on real silicon
  - For logic and fault simulation
  - Synopsys ModelSource hardware modeling systems
- Fault grading netlist
  - Full custom marcocells yields models suitable for hardware accelerated fault grading, system simulation and emulation
  - Emulator: IKOS, Mentor Graphics and Quickturn; Simulation: IKOS

Intent of ModelGen

- Key requirements for ARM's modeling environment:
  - Deliver highly secure models
  - Minimize time spent creating, porting and re-verifying models
  - Support mixed-source languages—HDL, C and full custom modeling
  - Support multiple design and verification environments
  - Enable efficient simulation
  - Provide a timing annotation solution that does not compromise IP security

“ModelGen” Timing Shell

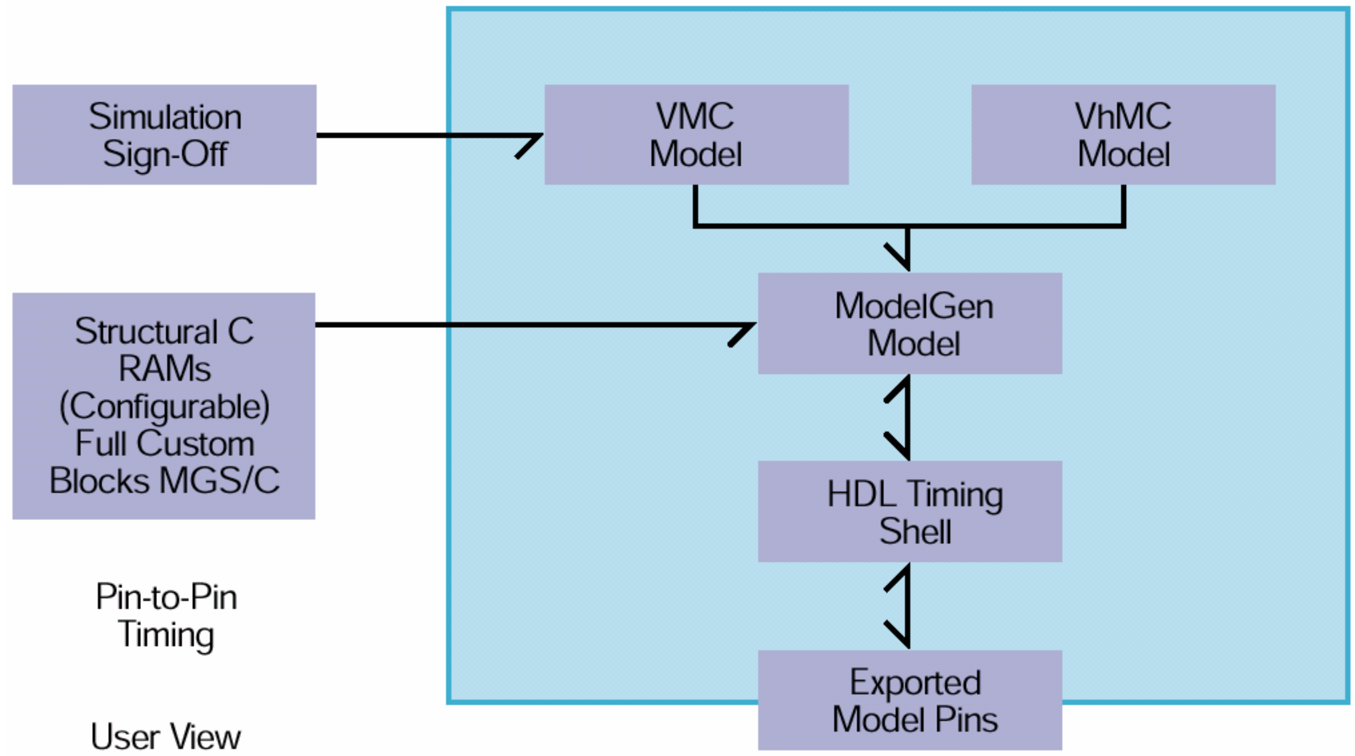
- Overview:
  - Black-box model
    - Obscured IP
  - User supplied timing (SDF)
  - Single model
    - Easily verifiable
  - Exported State
  - Programmer model
    - Nine-value Logic/Full
  - Supports checkpointing



IP Verification

Tian-Sheuan Chang

Example of Model Generation Flow

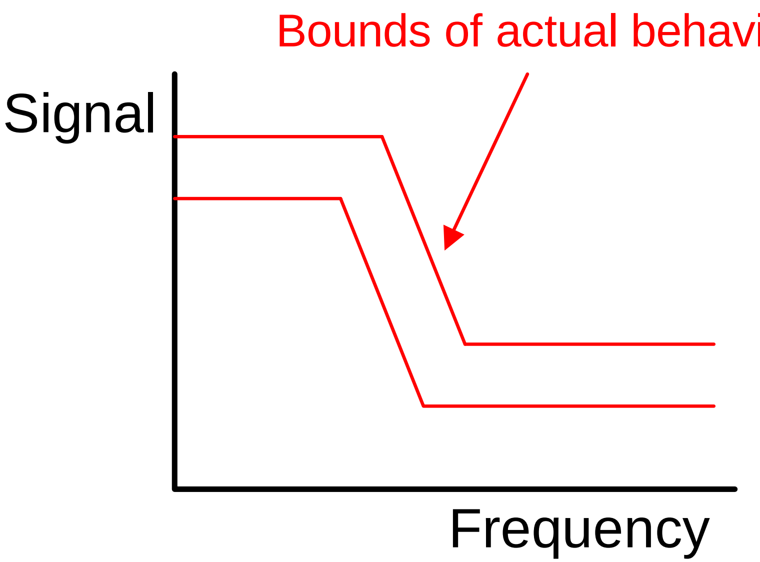


Synopsys VMC/VhMC based model generation flow

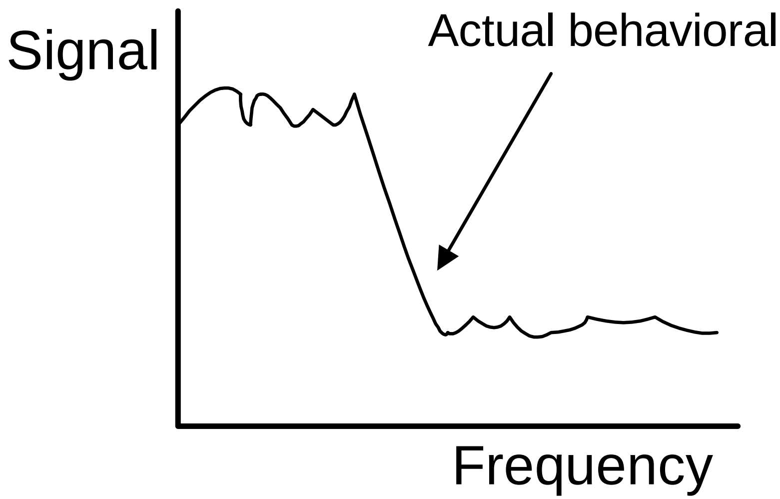
VMC: Verilog Model Compiler, VhMC: VHDL Model Compiler

Behavioral Model for A/MS

- Describes the functionality and performance of a VC block without providing actual detailed implementation.
- Needed for system designers to determine the possibility of implementing the system architecture
- It is a kind of *abstract* behavioral model



Behavioral Model



Block Detail Model

Tian-Sheuan Chang

Functional/Timing Digital Simulation Model

- Used to tie in functional verification and timing simulation with other parts of the system
- Describes the functionality and timing behavior of the entire A/MS VC between its input and output pins.
- Pin accurate not meant to be synthesizable
- It is a kind of *detailed*-behavioral model
- Example of PLL: represent the timing relationship of reference clock input vs. generate output clock.
  – Model it by actually representing the structure of the PLL, or
  – Model it as just a delay value based on a simple calculation from some parameters.

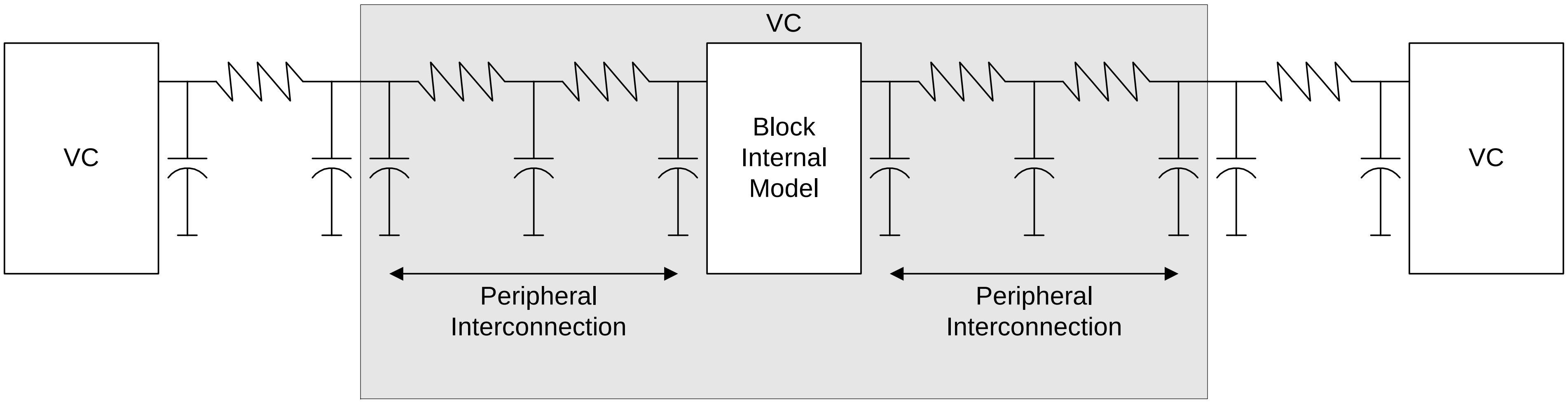
Tian-Sheuan Chang

Interface Model

- Describes the operation of a component with respect to its surrounding environment.
- The external connective points (e.g ports or parameters), functional and timing details of the interface are provided to show how the component exchanges information with its environment.
- Also named as *bus functional model* and *interface behavioral model*
- For A/MS VC
  - Only the digital interface is described
  - Analog inputs and outputs are not considered

Peripheral Interconnect Model

- Specifies the interconnection RCs for the peripheral interconnect between the physical I/O ports and the internal gates of the VC
- Used to accurately calculate the interconnect delays and output cell delays associated with the VC
- Used only for the digital interface of the A/MS VC



Power Model

- Defines the power specification of the VC
- Should be capable of representing both dynamic power and static power
  - Dynamic power may be due to capacitive loading or short-circuit currents
  - Static power may be due to state-dependent static currents
- Required for all types of power analysis: average, peak, RMS, etc.
- Abstract level
  - Black/gray box, RTL source code and cell level

Tian-Sheuan Chang

Basic Power Analysis Requirements

- Any power analysis should include effects caused by the following conditions and events:
  - Switching activity on input ports, output ports, and internal nodes
  - State conditions on I/O ports and optionally internal nodes
  - Modes of operations
  - Environmental conditions such as supply voltage and external capacitive or resistive loading.

Physical Modeling

- Physical block implementation of hard, soft and firm VCs.
- Two models for hard VCs
  - Detailed model
    - Description of the physical implementation of the VC at the polygon level
    - The preferred data format is GDSII 6.0.0
  - Abstract model
    - Contains enough information to enable floorplanning, placement, and routing of the system level chip
      - Footprint
      - Interface pin/port list, shape(s), and usage
      - Routing obstructions within the VC
      - Power and ground connections
      - Signature
    - The preferred data format is the MACRO section of VC LEF 5.1

Outline

- Verification challenges
- Verification process
- Verification tools
- RTL logic simulation
- RTL formal verification
- Verifiable RTL – good stuff
- Verifiable RTL – bad stuff
- Testbench design
- *SOC verification*

Tian-Sheuan Chang

System Verification

- It begins during system specification.
- Develop system-level behavioral model.
- Successful System-Level Verification
  - Quality of the test plan
  - Quality and abstraction level of the models and testbenches used
  - Quality and performance of the verification tools
  - Robustness of the individual predesigned blocks

IP Modeling

General Modeling Concept

Primary Model Classes:
- Behavioral Model
- Functional Model
- Structural Model

Specialized Model Classes:
- Performance Model
- Interface Model
- Hybrid Model

Computational Model Classes:
- Data Flow Graph Model
- Other Models

System Models
- Executable Specification
- Mathematical-Equation Model
- Algorithm Model

Architecture Models
- Token-based Performance Model
- Abstract-Behavioral Model
- Data Flow Graph (DFG) Task Primitive
- Instruction Set Architecture (ISA) Model

Hardware Models
- Detailed-Behavioral Model
- Register Transfer Level (RTL) Model
- Logic-Level Model
- Gate-Level Model
- Switch-Level Model
- Circuit-Level Model

Software Models
- Pseudo-Code
- High Level Language (HLL)
- Assembly code
- Micro-Code
- Object Code

Precision Axis
- Temporal Precision Axis
- Data Precision Axis
- Functional Precision Axis
- Structural Precision Axis
- Software Programming Precision Axis

SOC Verification

- System
  - Validate through
    - Prototype, real chip or FPGA
  - Methodology
    - High level model execution
    - Hardware/software co-simulation
    - Prototype or run software on sample chip
  - Rapid prototyping is necessary for verification
    - since RTL or gate-level simulation is the bottleneck when developing a derivative design
  - The most appropriated rapid-prototyping device for platform design consists of
    - A hardwire hardware kernel (real chip)
    - Slots of FPGA on the hardware kernel's bus for configurations

Tian-Sheuan Chang

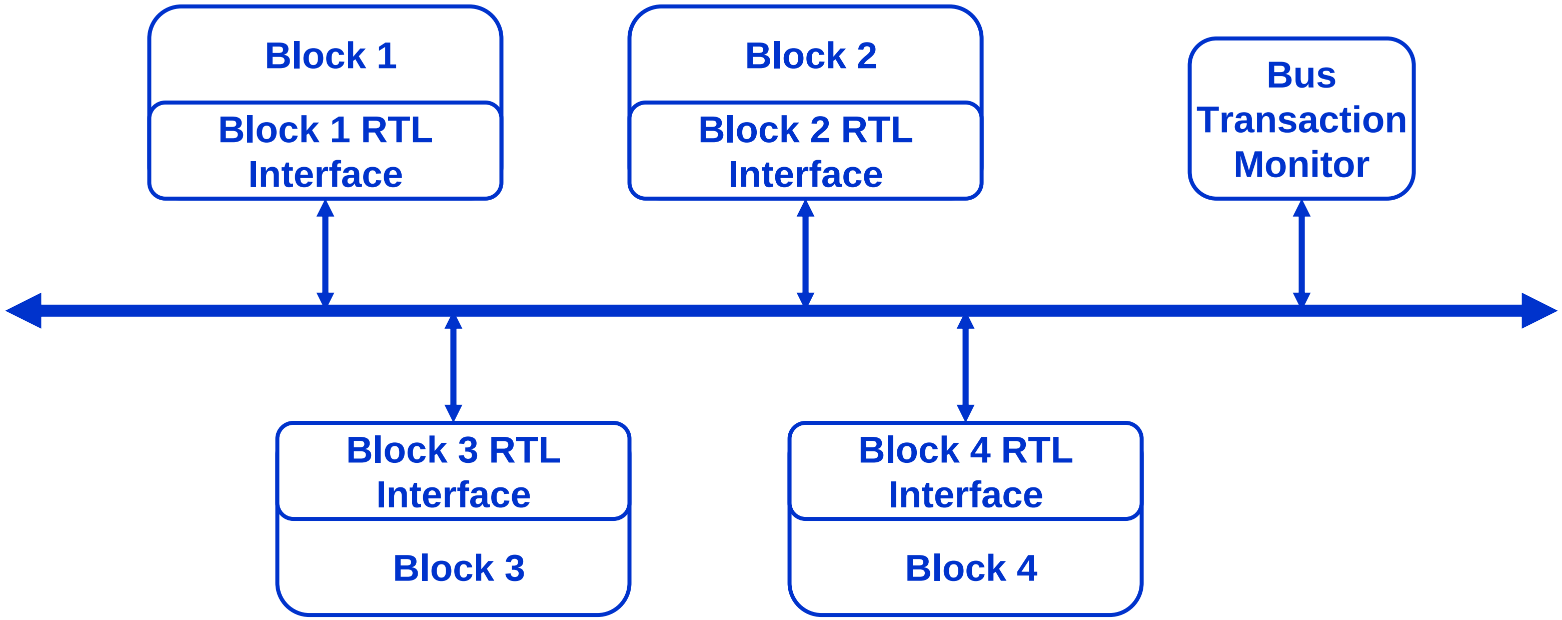
The Test Plan

- System-level verification strategy uses divide-and-conquer approach based on the system hierarchy.
  - Verify the leaf nodes.
  - Verify the interfaces between blocks that are functionally correct.
  - Run a set of increasingly complex applications on the full chips.
  - Prototype the full chip and run a full set of application software for final verification.
  - Decide when it is appropriate to release the chip to production.

Tian-Sheuan Chang

Interface Verification

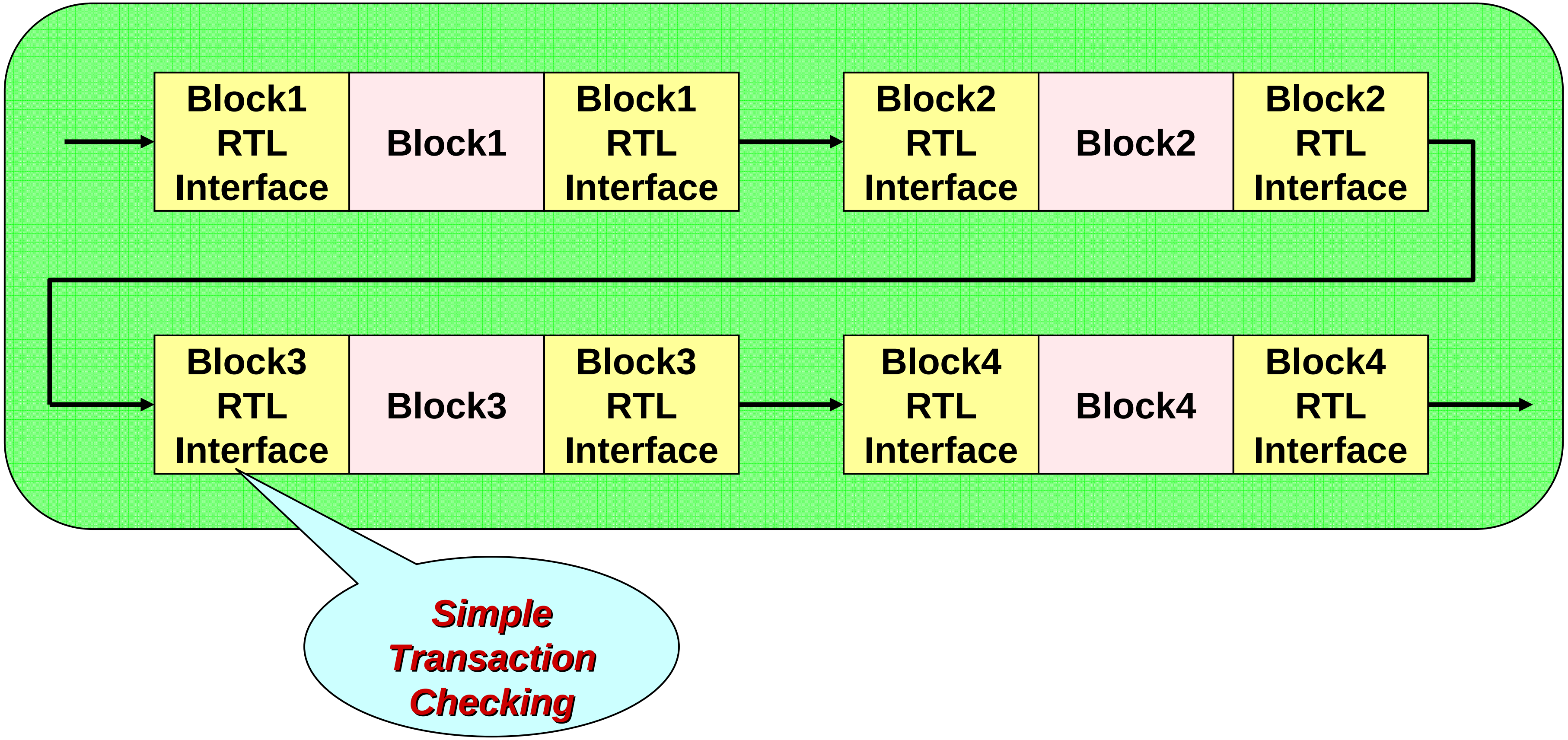
- Interface: address/data bus. Protocols
  - permitted sequence of control and data signals
  - use a bus transaction monitor to check the transaction



- Use BFM to check the data read and write

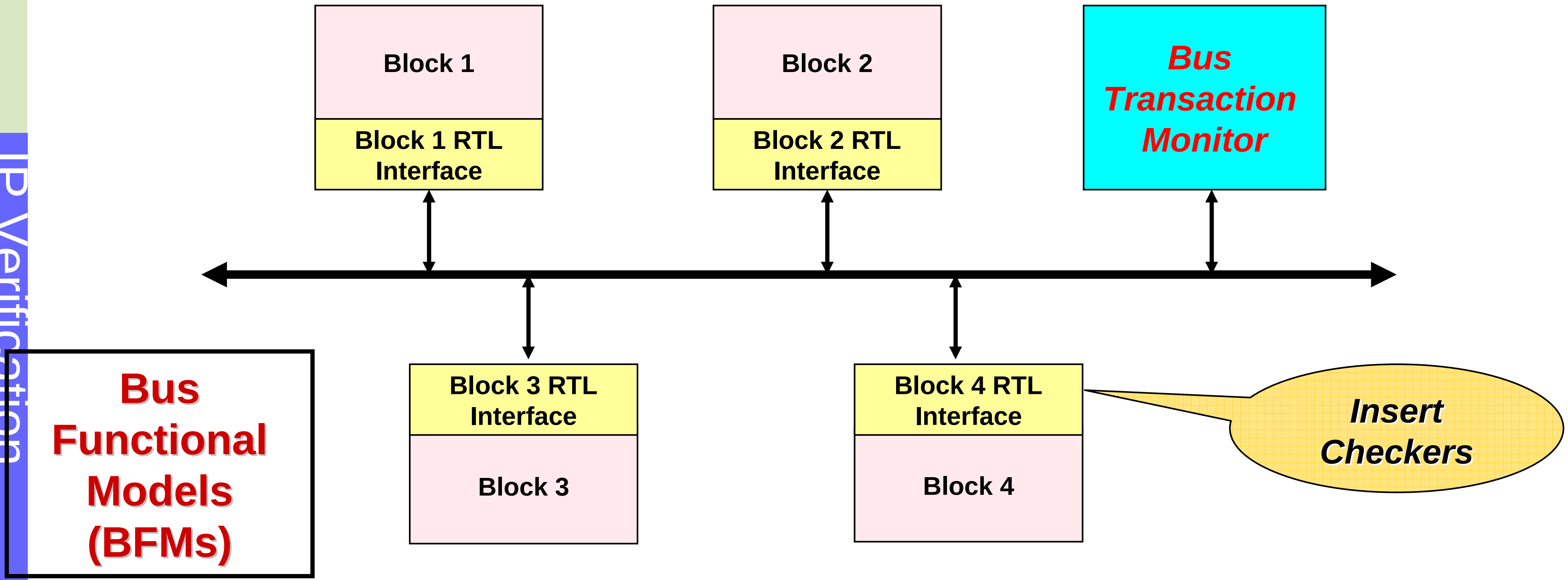
System Verification using Interface Testing (1)

- Chip with Point-to-Point Interfaces



System Verification using Interface Testing (2)

- Chip with an On-Chip-Bus



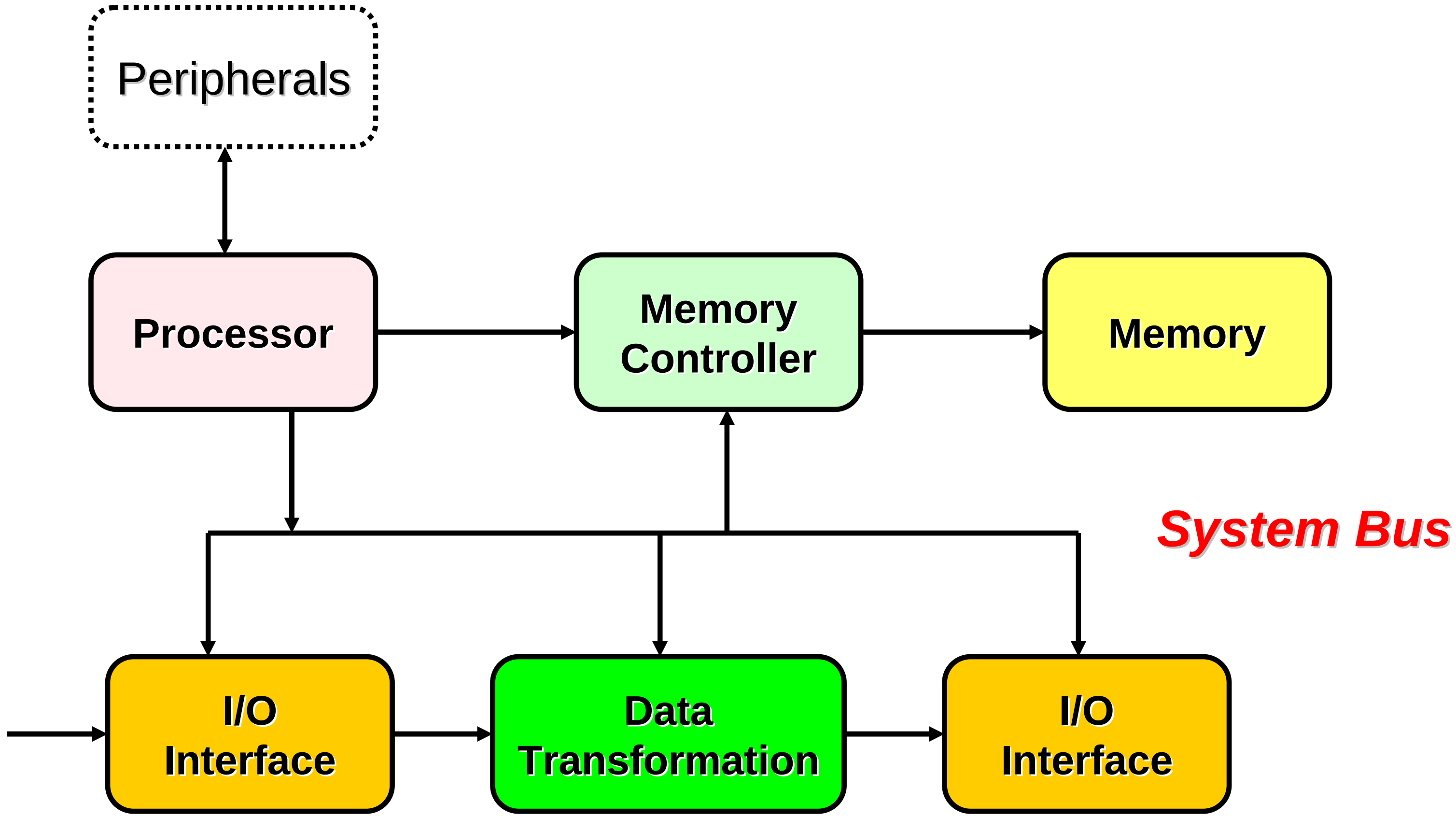
Functional Verification (1/2)

- Two basic approaches
  - increase level of abstraction so that software simulators running on workstations faster
  - use specialized hardware for performing verification, such as emulator or rapid prototyping
- Canonical SoC abstraction
  - Full RTL model for IP cores
  - behavior or ISA model for memory and processor
  - bus functional model and monitor to generate and check the transactions between IPs
  - generate real application code for the processor and run it on the simulation model

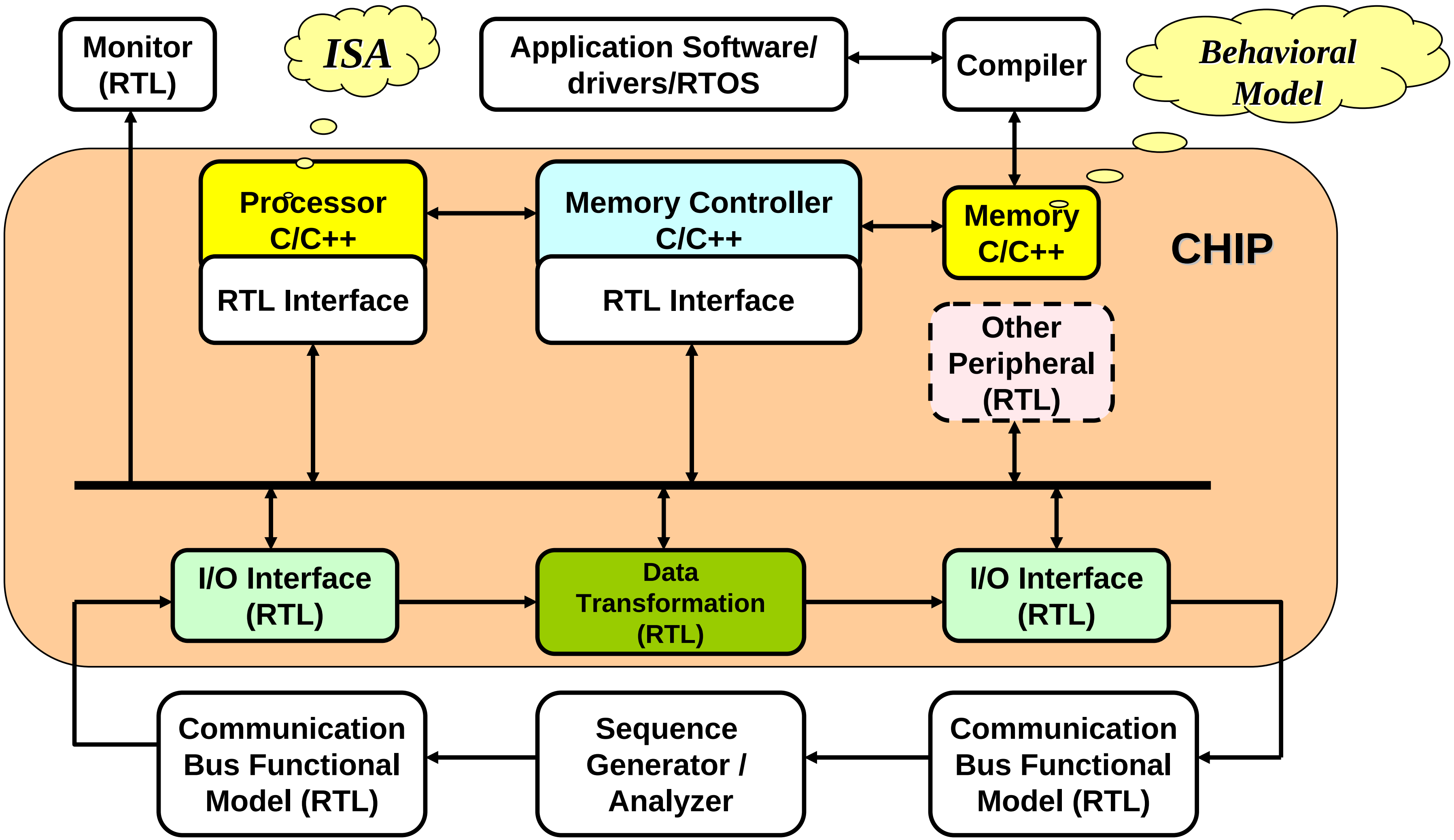
Tian-Sheuan Chang

Functional Verification (2)

- A canonical SOC design



Functional Verification (3)



IP Verification

Tian-Sheuan Chang

Application-Based Verification

- Run actual applications on the system (a full functional model).
  - Major Challenge
    - RTL Simulation is the bottleneck.
  - Two approaches to address this problem
    - Increase the level of abstraction of design.
    - Use specialized hardware for performance verification
      - Emulation
      - Rapid Prototyping

Tian-Sheuan Chang

Gate-Level Verification

- Correct functionality and timing
- Sign-Off Simulation
- Formal Verification
- Gate-Level simulation with Unit-Delay Timing
- Gate Level Simulation with Full Timing

Tian-Sheuan Chang

Sign-Off Simulation

- Gate-level simulation, parallel test vectors, full scan methodology
- RTL sign-off problems
  - Simulation speed is too slow
  - Parallel vectors with very low fault coverage
  - Parallel vectors do not exercise all the critical timing paths
- Traditional addressed problems
  - Verification that synthesis has generated a correct netlist
  - Verification that the chip, when fabricated, will meet timing
  - A manufacturing test
- Different Approaches
  - Formal Verification
  - Static Timing Analysis
  - Some Gate-level Simulation
  - Full Scan plus BIST

Tian-Sheuan Chang

Rapid Prototyping

- FPGA prototyping
  - Aptix (FPGAs + programmable routing chips
- Emulation-based testing
  - FPGA-based or processor-based
  - QuickTurn and Mentor Graphics
- Real silicon prototyping
  - faster and easier to build an actual chip and debug it
  - design features in the real silicon chip
    - good debug structure
    - ability to selectively reset the individual IP blocks
    - ability to selectively disable various IP blocks to prevent bugs from affecting operations of the system

Specialized Hardware for System Verification

- System simulation through specialized hardware systems for verification.
  - Zycad, IKOS
    - These accelerators map the standard, event-driven software simulation algorithm onto specialized hardware.
    - Parallel execution on multiple processors.
  - Emulation Systems
    - Non-synthesizable code, especially testbenches, must run the host machine.
    - The partitioning of the circuits among numerous FPGAs.
    - The use of FPGA makes controlling and observing individual nodes in the circuit difficult.

Tian-Sheuan Chang