# CS 250
# VLSI System Design

# Lecture 8 – Design Verification

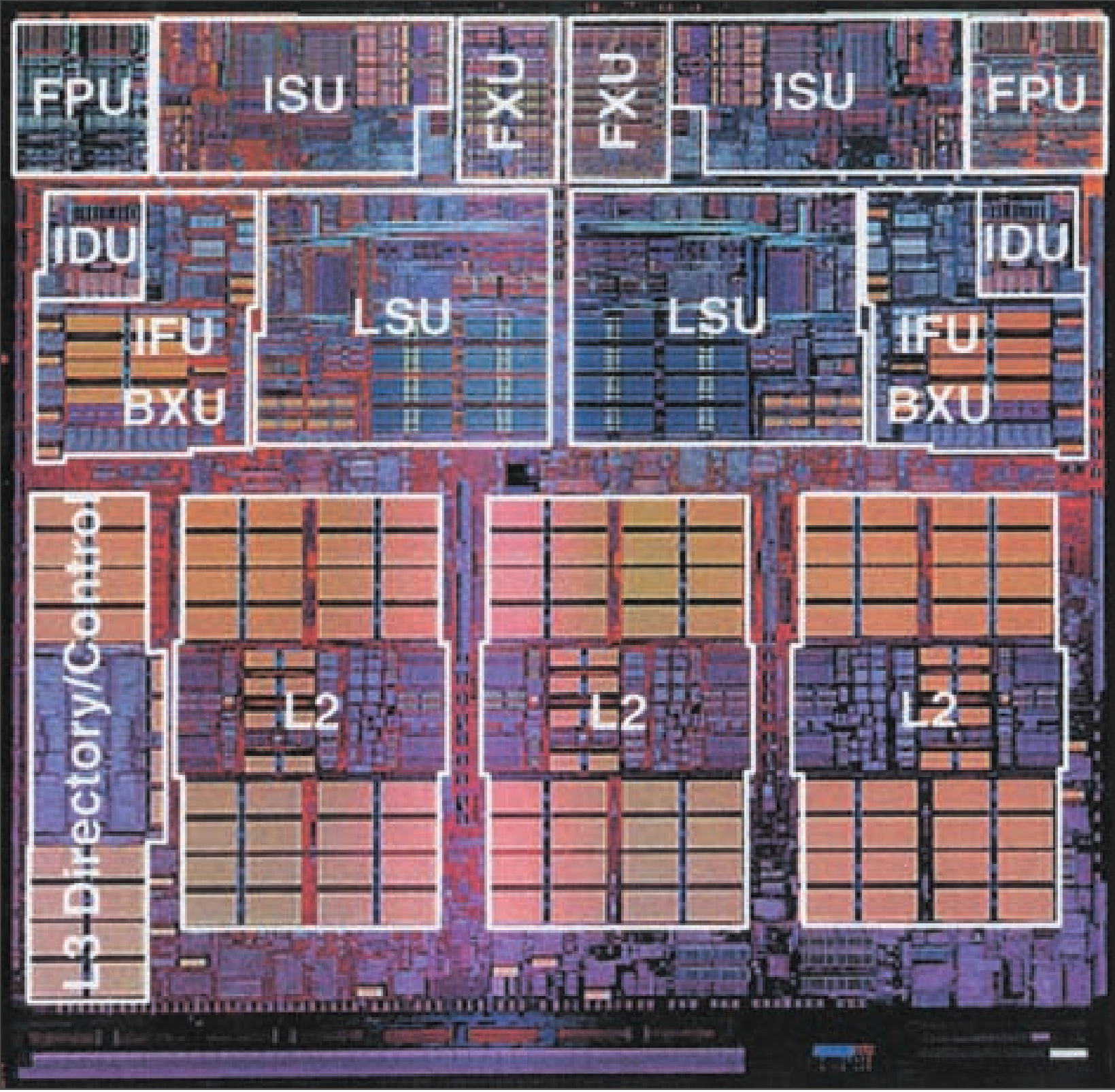
**2013-9-24**

**Professor Jonathan Bachrach**

**part one of today's lecture by John Lazzaro**

TA: Ben Keller

**www-inst.eecs.berkeley.edu/~cs250/**




# IBM Power 4

## 174 Million Transistors

## A complex design ...

**First silicon booted AIX & Linux, on a 16-die system.**

**96% of all bugs were caught before first tape-out.**

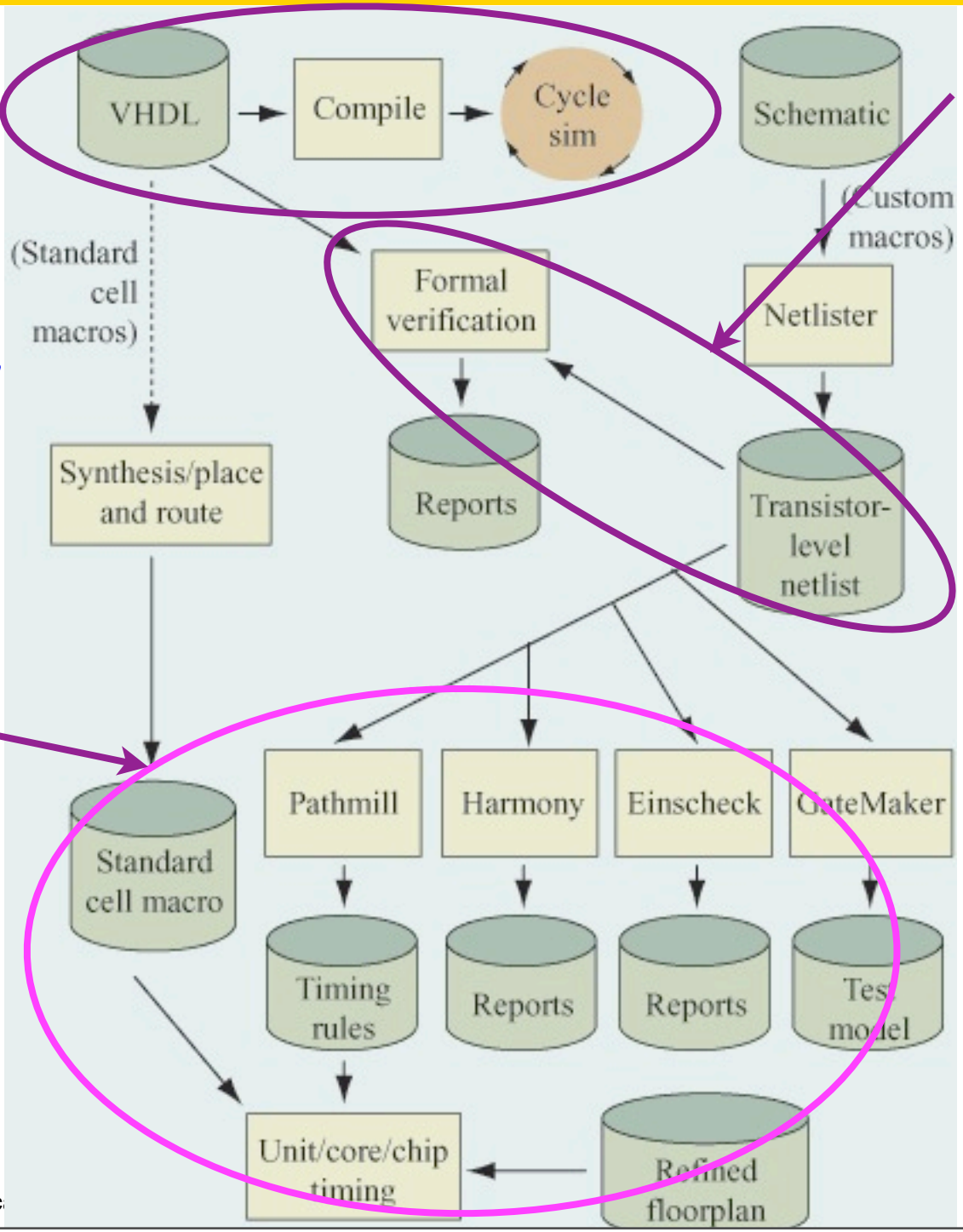
## How ???

# Three main components ...

(1) Specify chip behavior at the RTL level, and comprehensively simulate it.

(2) Use formal verification to show equivalence between VHDL RTL and circuit schematic RTL.

(3) Technology layer: do the the electrons implement the RTL, at speed and power?

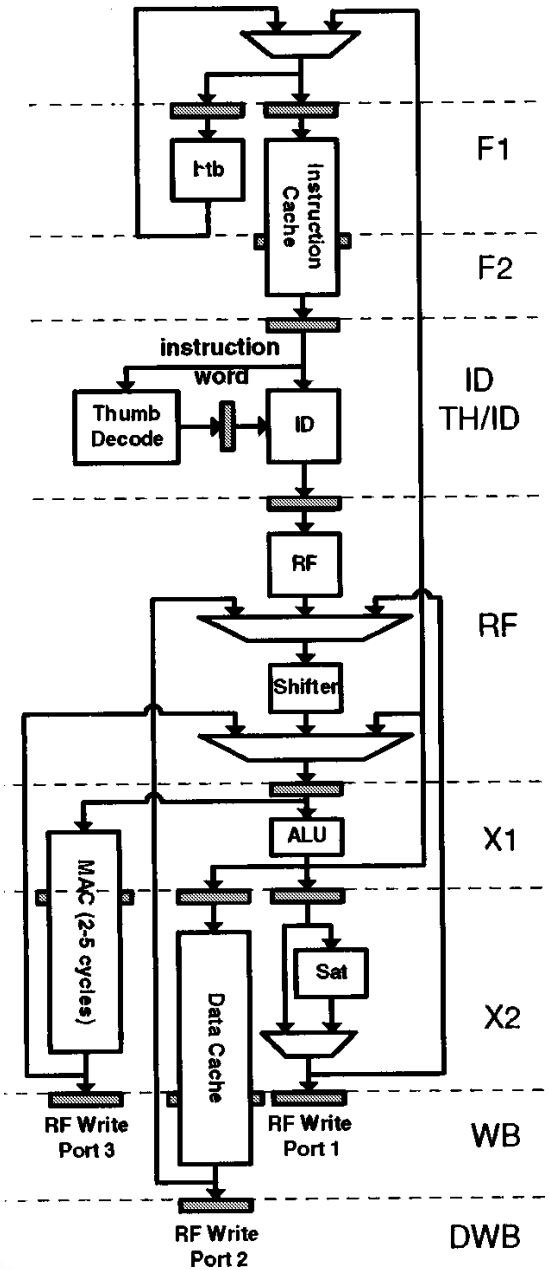



Today, we focus on (1).

# Today: Processor Design Verification

- Making a processor test plan
- Unit testing techniques
- State machine testing
- How to write test programs

# Lecture Focus: Functional Design Test




testing goal

Not manufacturing tests ...

**The processor design correctly executes programs written in the Instruction Set Architecture**

"Correct" == meets the "Architect's Contract"

Intel XScale ARM Pipeline, IEEE Journal of Solid State Circuits, 36:11, November 2001

# Architect's "Contract with the Programmer"

- To the program, it appears that instructions execute in the correct order defined by the ISA.
- As each instruction completes, the architected machine state appears to the program to obey the ISA.
- What the machine actually does is up to the hardware designers, as long as the contract is kept.

Accelerator instructions must define a "contract" ...

# When programmer's contract is broken ...

**Testing our financial trading system, we found a case where our software would get a bad calculation. Once a week or so.**

**Eventually, the problem turned out to be a failure in a CPU cache line refresh. This was a hardware design fault in the PC.**

**The test suite included running for two weeks at maximum update rate without error, so this bug was found.**

**Eric Ulevik**

# A 475M$ Bug

The correct value is

$$\frac{4195835}{3145727} = 1.3338204491362410 02$$

However, the value returned by the flawed Pentium is incorrect

$$\frac{4195835}{3145727} = 1.3337390689020375 89$$

## Pentium FDIV bug

From Wikipedia, the free encyclopedia

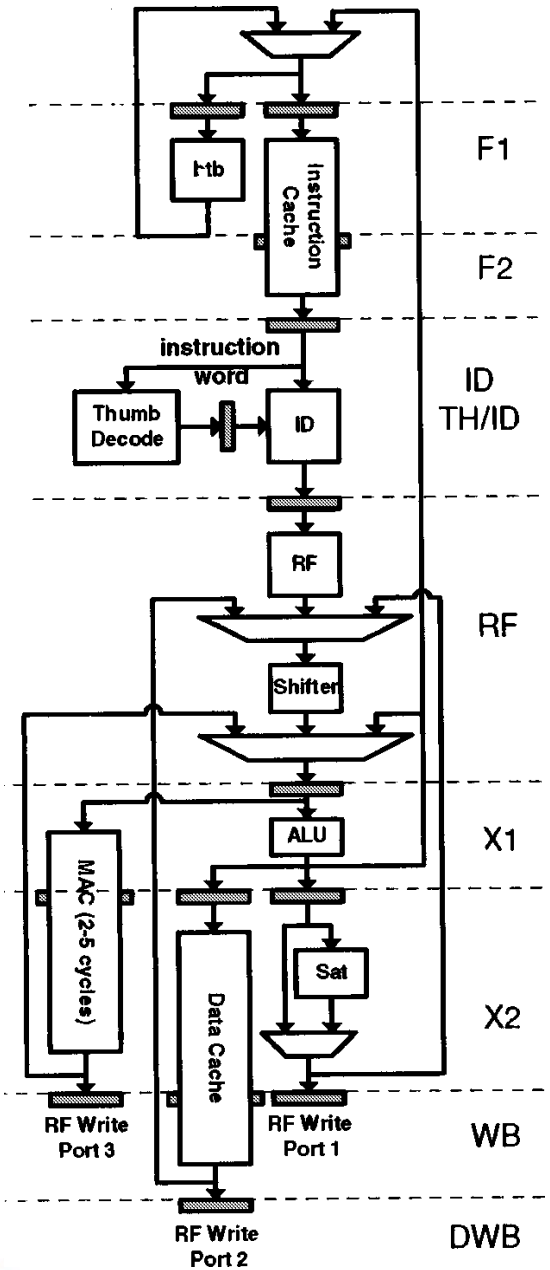
The **Pentium FDIV bug** was a bug in the Intel P5 Pentium floating point unit (FPU). Certain floating point division operations performed with these processors produced incorrect results. According to Intel, there were a few missing entries in the lookup table used by the digital divide operation algorithm.[1]

The flaw was independently discovered and publicly disclosed in October 1994 by Professor Thomas R. Nicely at Lynchburg College, Virginia, USA.[2]

Although encountering the flaw was extremely rare in practice (*Byte* magazine estimated that 1 in 9 billion floating point divides with random parameters would produce inaccurate results),[3] both the flaw and Intel's initial handling of the matter were heavily criticized. Intel ultimately recalled the defective processors.


66MHz Intel Pentium (sSpec=SX837) with the FDIV bug

# Three models (at least) to cross-check.




- **The "contract" specification**

  "The answer" (correct, we hope). Simulates the ISA model in C. Fast. Better: two models coded independently.

- **The Chisel RTL model**

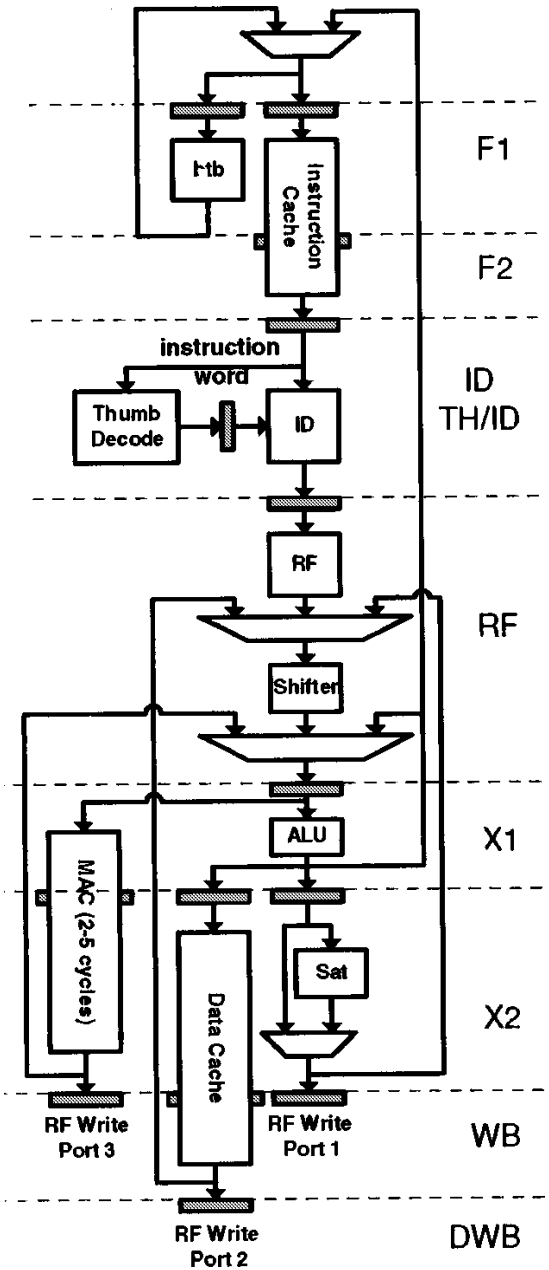
  Logical semantics of the Chisel model we will use to create gates. Runs on a software simulator or FPGA hardware.

- **Chip-level schematic RTL**

  Extract the netlist from layout, formally verify against Chisel RTL. Catches synthesis bugs. This netlist also used for timing and power.

Where do bugs come from?

# Where bugs come from (a partial list) ...




- **The contract is wrong.**
  You understand the contract, create a design that correctly implements it, write correct Chisel for the design ...
- **The contract is misread.**
  Your design is a correct implementation of what you think the contract means ... but you misunderstand the contract.
- **Conceptual error in design.**
  You understand the contract, but devise an incorrect implementation of it ...
- **Chisel coding errors.**
  You express your correct design idea in Chisel .. with incorrect Chisel semantics.

Also: CAD-related errors. Example: Chisel-to-Verilog translation errors.

# Four Types of Testing

# Big Bang: Complete Processor Testing

Top-down testing

- complete processor testing

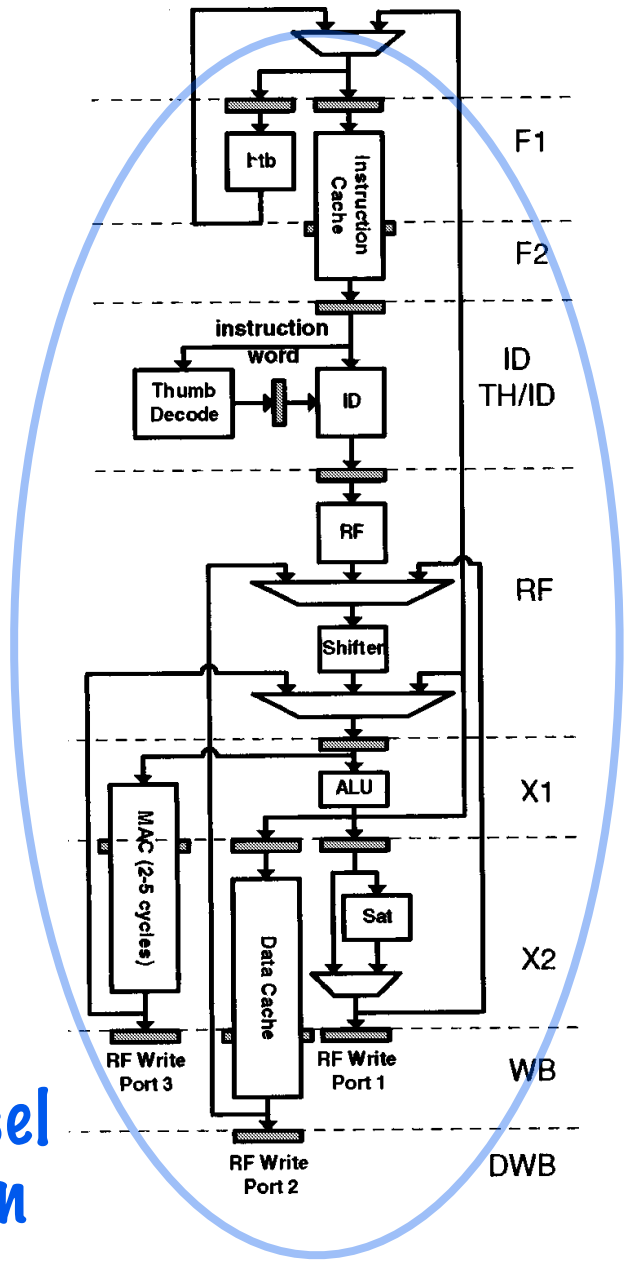
Bottom-up testing

## how it works

**Assemble the complete processor.**

**Execute test program suite on the processor.**

**Check results.**




Checks contract model against Chisel RTL. Test suite runs the gamut from "1-line programs" to "boot the OS".

# Methodical Approach: Unit Testing

Top-down testing

complete processor testing

- unit testing

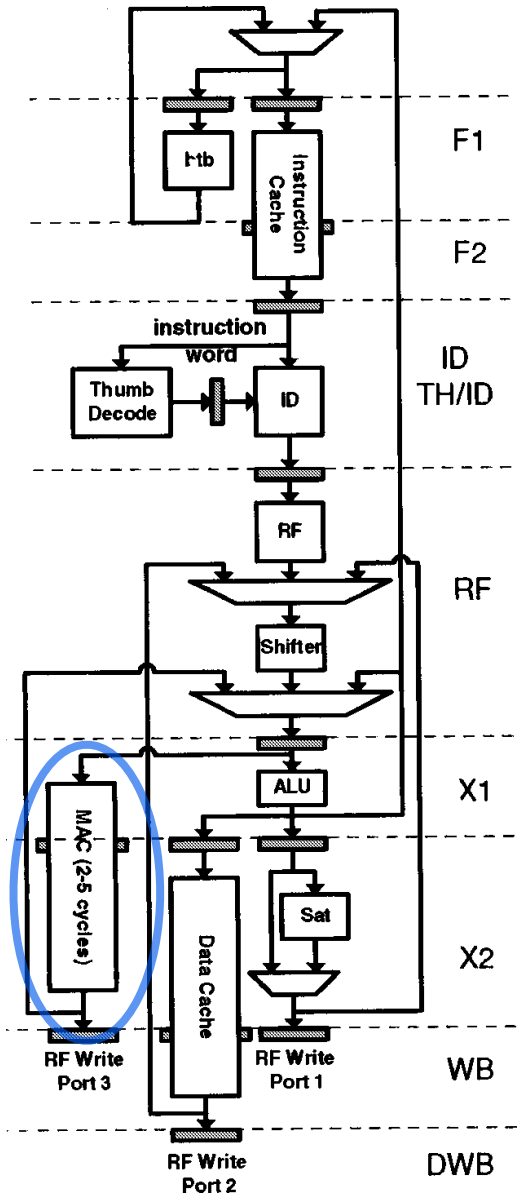
Bottom-up testing

## how it works

**Remove a block from the design.**

**Test it in isolation against specification.**

Requires writing a bug-free "contract model" for the unit.



# Climbing the Hierarchy: Multi-unit Testing

Top-down testing

complete processor testing

- multi-unit testing

unit testing

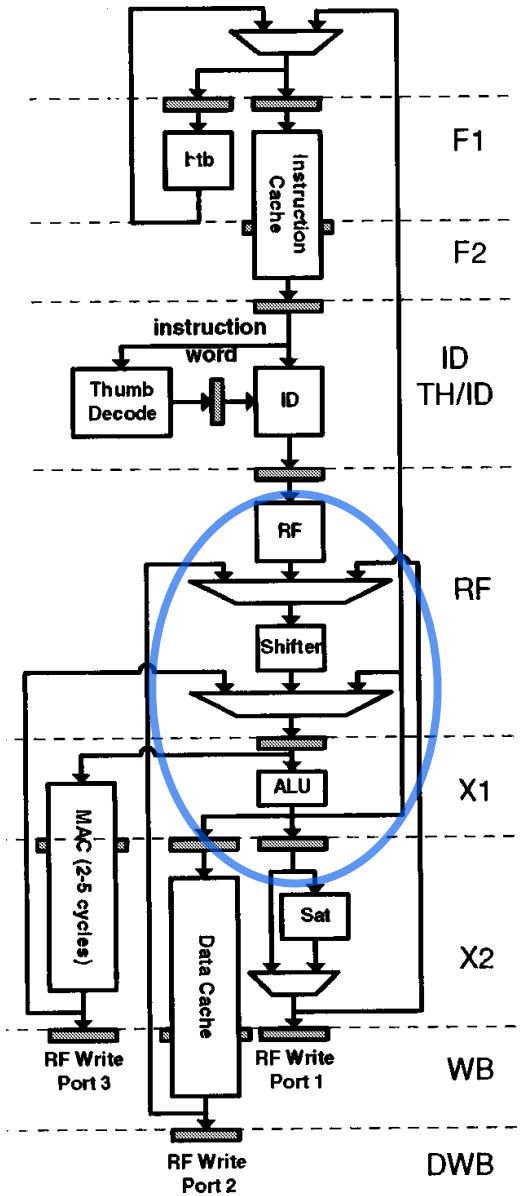
Bottom-up testing

## how it works

**Remove connected blocks from design.**

**Test in isolation against specification.**

*Choice of partition determines if test is "eye-opening" or a "waste of time"*



# Processor Testing with Self-Checking Units

Top-down testing

complete processor testing

- processor testing with self-checks

multi-unit testing

unit testing

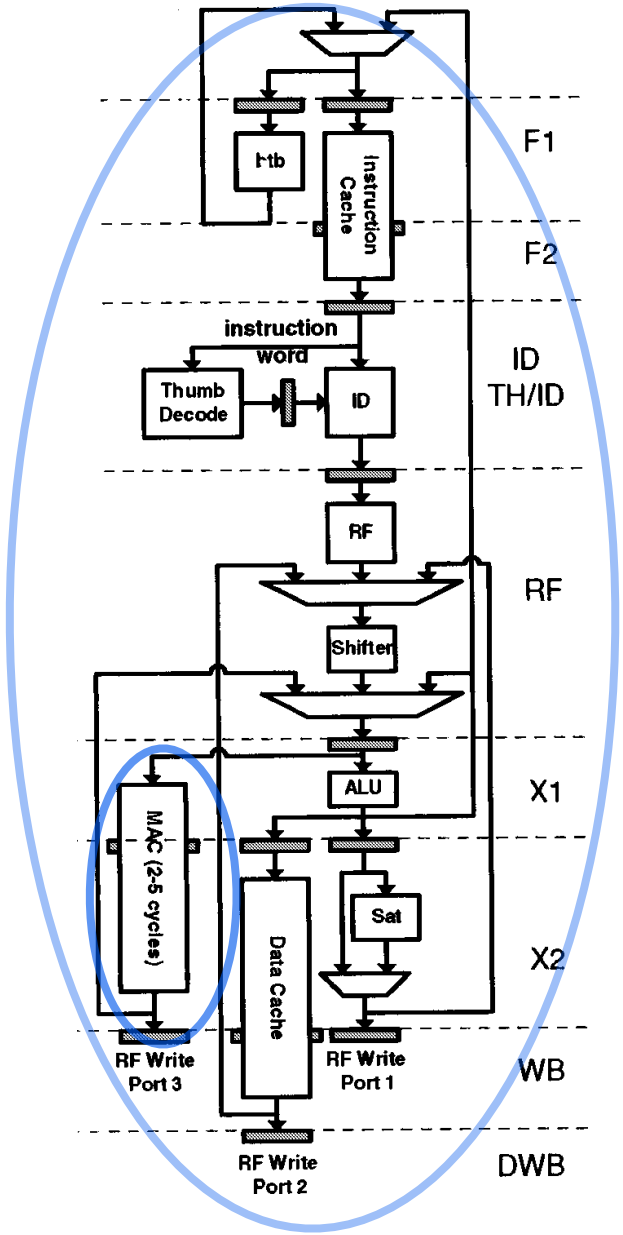
Bottom-up testing

## how it works

**Add self-checking to units**

**Perform complete processor testing**

Self-checks are unit tests built into CPU, that generate the "right answer" on the fly. Slower to simulate.



# Testing: Verification vs. Diagnostics

Top-down testing

- complete processor testing
- processor testing with self-checks
- multi-unit testing
- unit testing

Bottom-up testing

- **Verification:**

  A yes/no answer to the question "Does the processor have one more bug?"

- **Diagnostics:**

  Clues to help find and fix the bug.

Diagnosis of bugs found during "complete processor" testing is hard ...

# "CPU program" diagnosis is tricky ...

**Observation**: On a buggy CPU model, the correctness of every executed instruction is suspect.

**Consequence**: One needs to verify the correctness of instructions that surround the suspected buggy instruction.

Depends on: (1) number of "instructions in flight" in the machine, and (2) lifetime of non-architected state (may be "indefinite").

# State observability and controllability

Top-down testing

complete processor testing

processor testing with self-checks

multi-unit testing

unit testing

Bottom-up testing

- **Observability:**

  Does my model expose the state I need to diagnose the bug?

- **Controllability:**

  Does my model support changing the state value I need to change to diagnose the bug?

Support != "yes, just rewrite the model code"!

# Writing a Test Plan

# The testing timeline ...

Top-down testing

complete processor testing

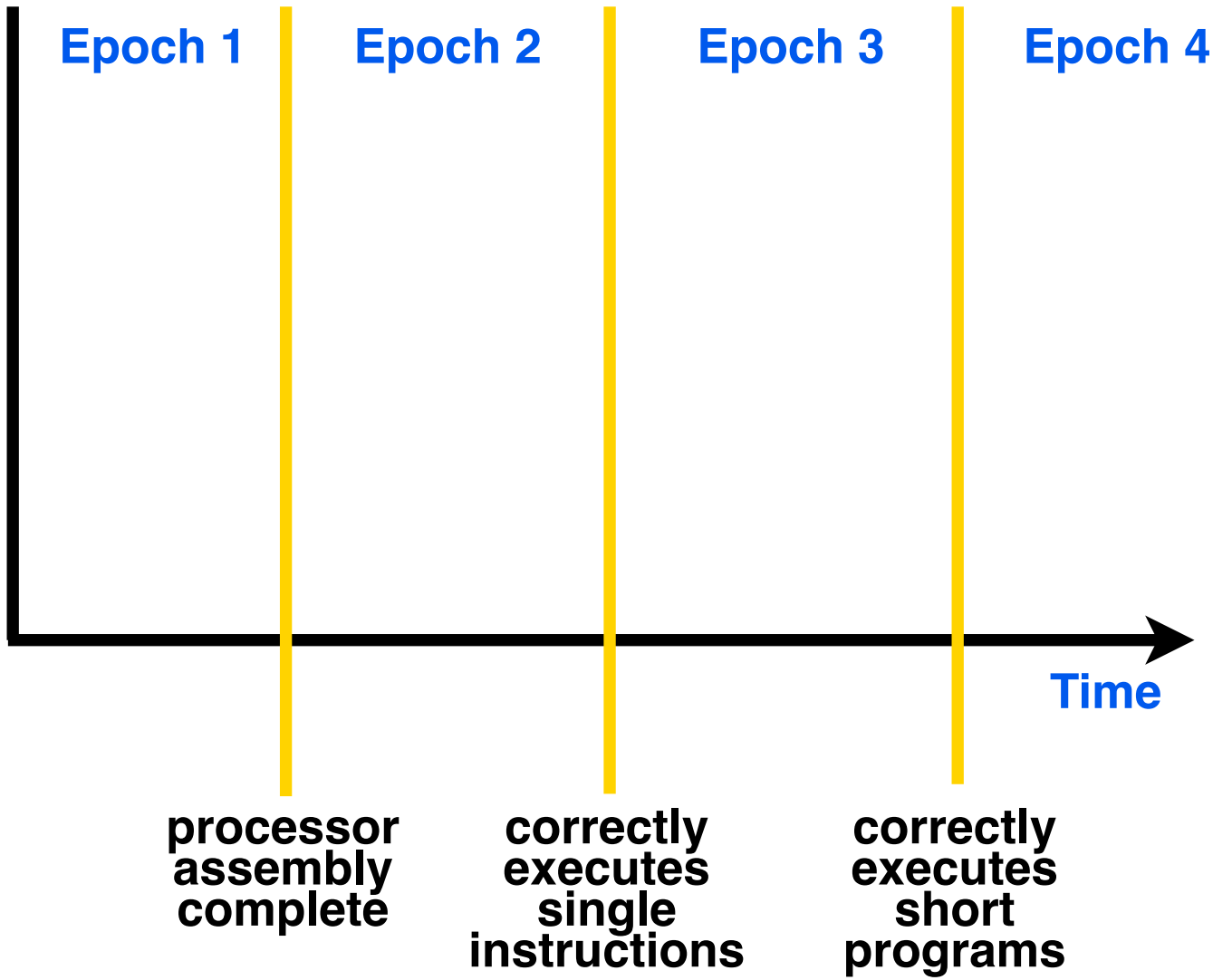
processor testing with self-checks

multi-unit testing

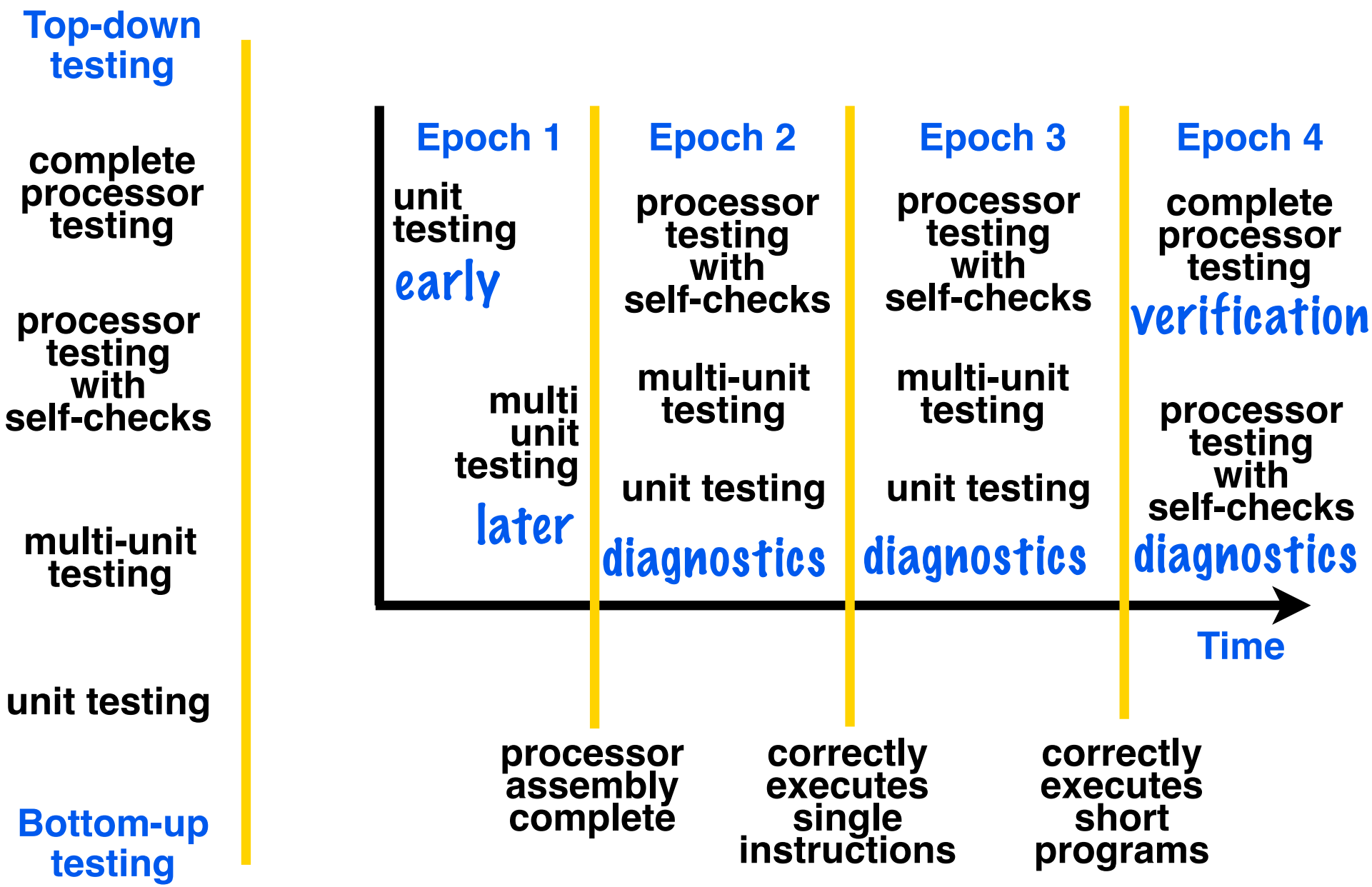
unit testing

Bottom-up testing

## Plan in advance what tests to do when ...

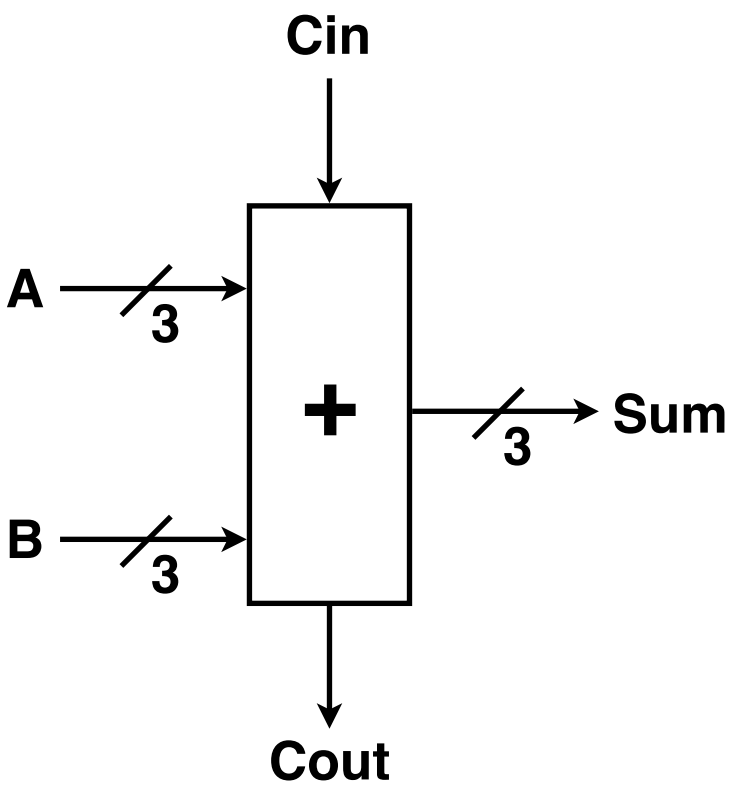


# An example test plan ...



# Unit Testing

# Combinational Unit Testing: 3-bit Adder




Number of input bits ? 7

Total number of possible input values?

$2^7 = 128$

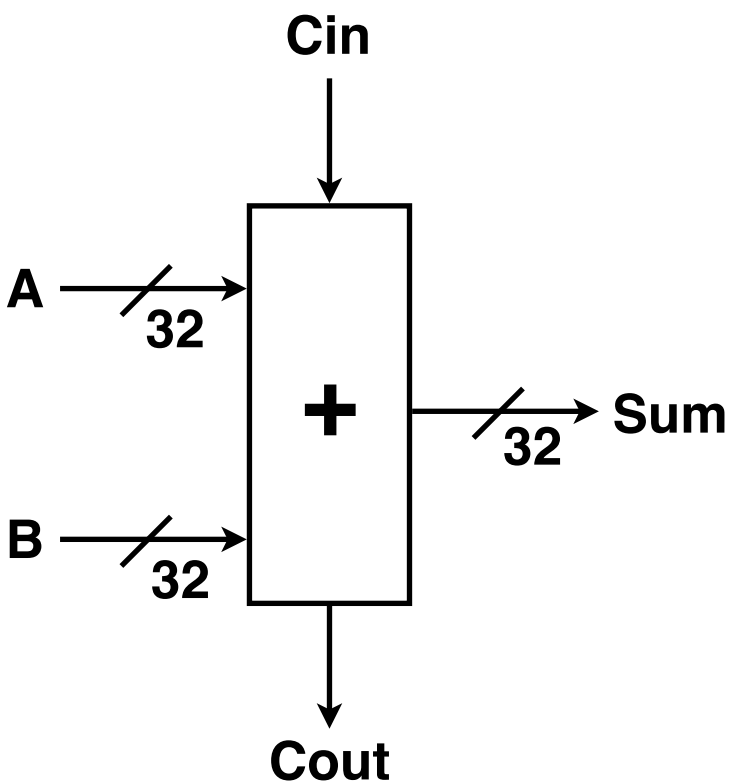
**Just test them all ...**

**Apply "test vectors" 0,1,2 ... 127 to inputs.**

**100% input space "coverage"**

**"Exhaustive testing"**

# Combinational Unit Testing: 32-bit Adder




Number of input bits ? 65

Total number of possible input values?

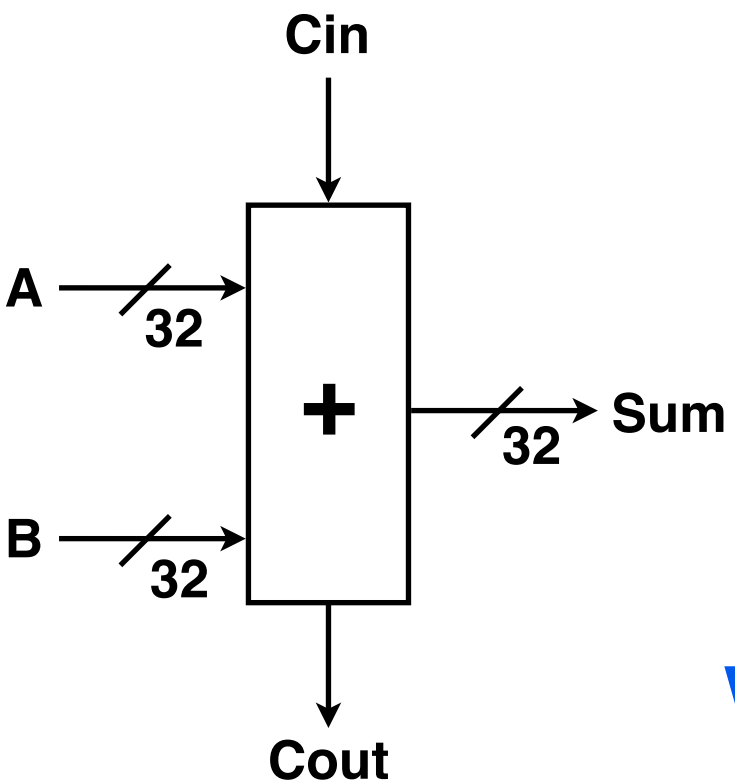
$2^{65} = 3.689e+19$

**Just test them all?**

**Exhaustive testing does not "scale".**

"Combinatorial explosion!"

# Test Approach 1: Random Vectors

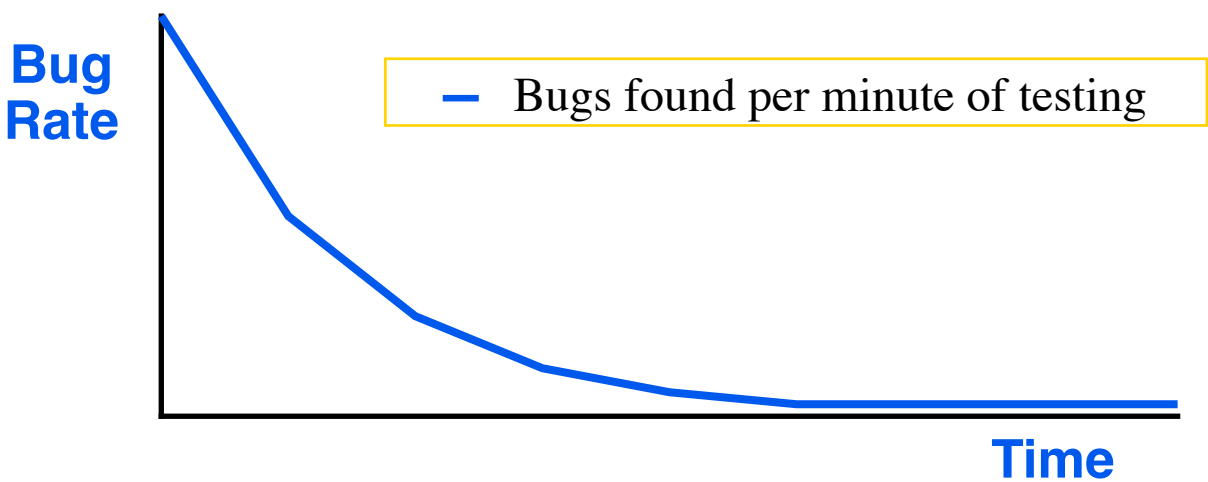



## how it works

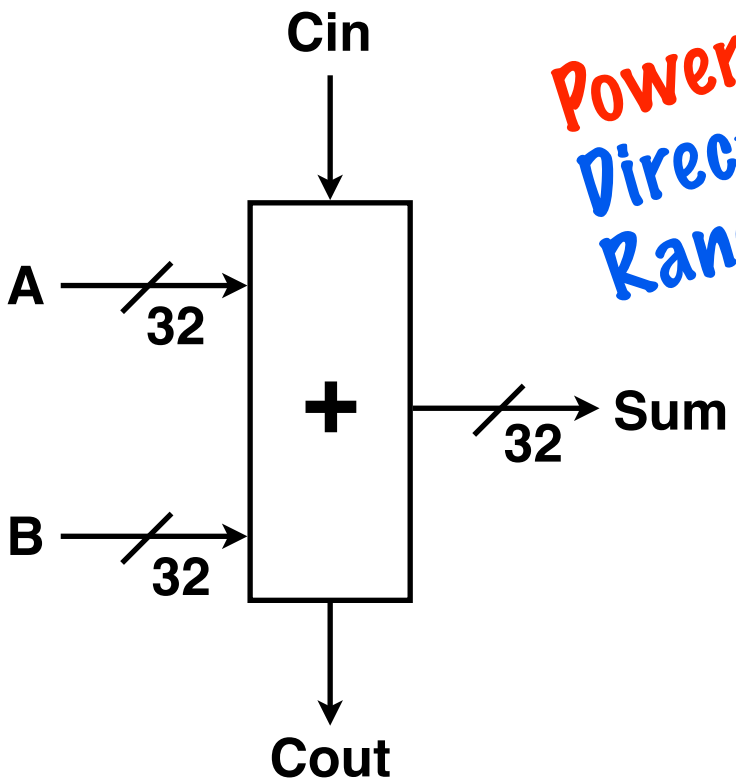
**Apply random A, B, Cin to adder.**

**Check Sum, Cout.**

## When to stop testing? Bug curve.



# Test Approach 2: Directed Vectors




Power Tool: Directed Random

how it works

**Hand-craft test vectors to cover "corner cases"**

**A == B == Cin == 0**

**"Black-box": Corner cases based on functional properties.**

**"Clear-box": Corner cases based on unit internal structure.**

# State Machine Testing

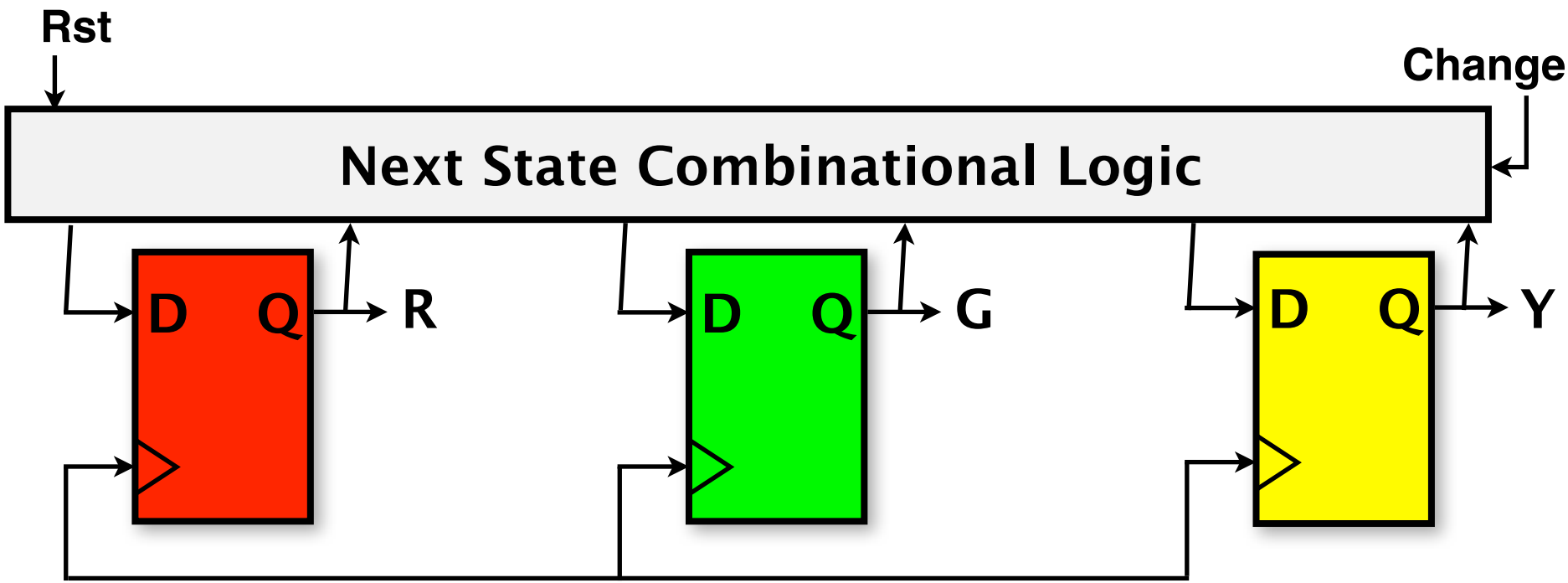
CPU design examples

DRAM controller state machines

Cache control state machines

Branch prediction state machines

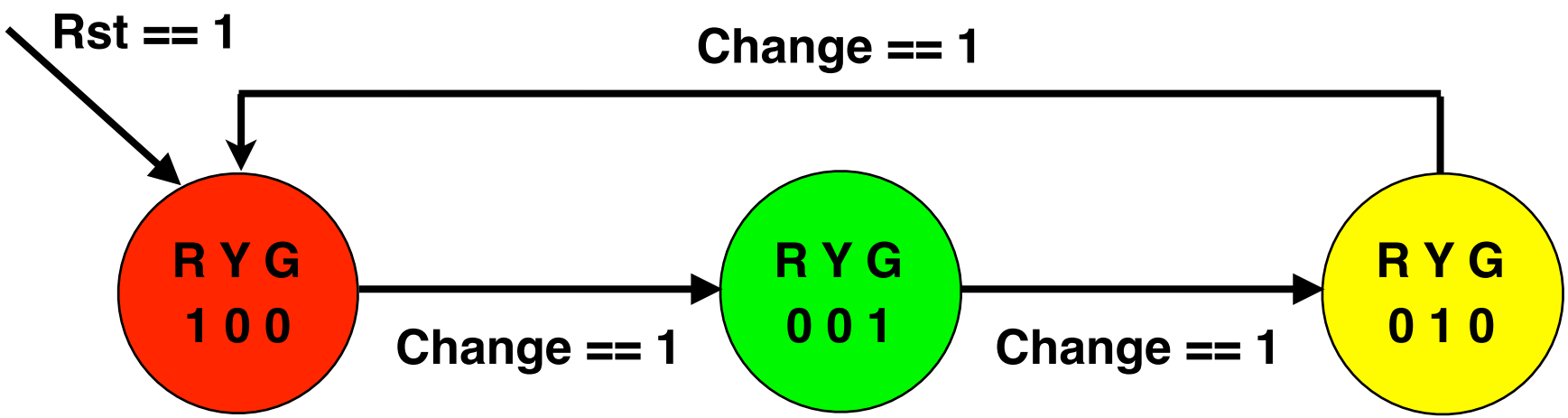
# Testing State Machines: Break Feedback




**Isolate "Next State" logic.**
**Test as a combinational unit.**

Easier with certain HDL coding styles ...

# Testing State Machines: Arc Coverage




**Force machine into each state.**
**Test behavior of each arc.**

Intractable for state machines with high edge density ...

# Regression Testing

Or, how to find the last bug ...

# Writing "complete CPU" test programs

Top-down testing

complete processor testing

processor testing with self-checks

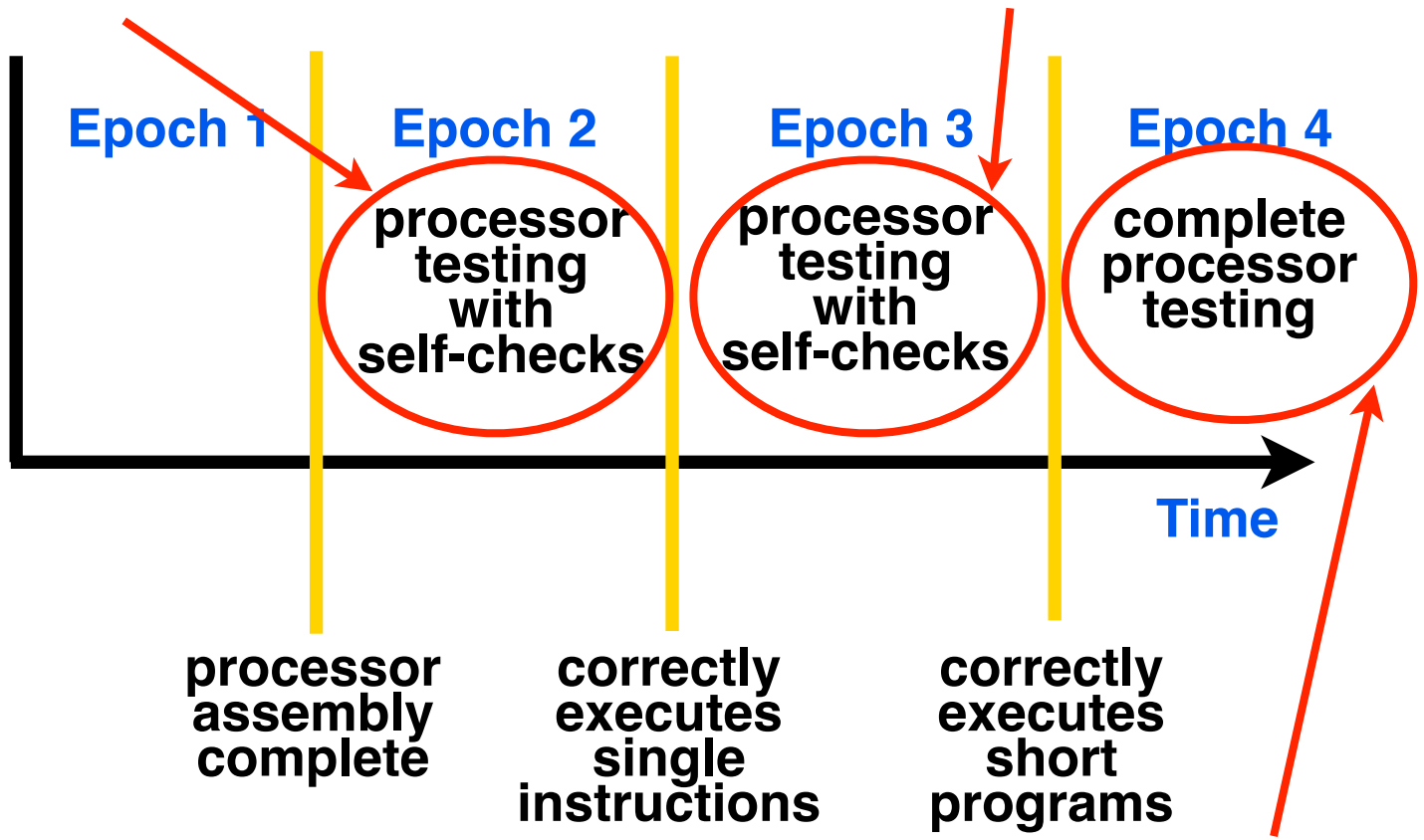
multi-unit testing

unit testing

Bottom-up testing

Single instructions with directed-random field values.

White-box "Instructions-in-flight" sized programs that stress design.




Tests that stress long-lived non-architected state.

Regression testing: re-run subsets of the test library, and then the entire library, after a fix.

# Conclusion -- Testing Processors

- Bottom-up test for diagnosis, top-down test for verification.
- Unit testing: avoiding combinatorial explosions.
- Complete CPU tests: write programs that stress the hard parts of the design.
- Make your testing plan early!

After the break: Chisel testing tips.