A USD$ 475 Million Bug

• Intel’s Pentium Bug in 1994
  – 3.3 million transistors on chip
  – 1 trillion ($10^{12}$) cycles were simulated before tape-out
  – a bug in floating-point division hardware was revealed
  – fixed by adding 5 more transistors

• Intel offered replacements to users
  – $475 Million USD penalty at Q4’94
Verification Challenge (1/3)

- Designs grow as Moore’s Law predicted
- Design productivity has risen 10x since 1990 due to synthesis
- Effort for verification doubles every 6-9 months

The challenge of functional verification is getting bigger
Verification Challenge (2/3)

- 50% of chips require 1 or more re-spins
- 74% of re-spins are due to functional defects

![Bar Chart showing reasons for re-spins]

Colleit International 2000
Verification Challenge (3/3)

• Average 1.5 functional bugs per 200 lines in RTL
  – 150 bugs in a typical 20K-line design

• Average 1.5 man-day required to find a bug
  – 225 man-day required for a 20K-line design

• *Do you find all bugs before tape-out?*

• *How do you find bugs quicker and easier?*
Glossary

- Verification environment
  - commonly referred as testbench (environment)

- Definition of testbench
  - a verification environment containing a set of components - such as bus functional models (BFMs), bus monitors, memory models - and the interconnect of such components with the design-under-test

- Verification (Test) suites (stimuli, patterns, vectors)
  - test signals and the expected response under given testbenches
Verification Reuse

• Verification goal is *defect-free*
• The macro-level testbenches and suites should be reusable
  – the macro may be redesigned later
• The integration team must be able to reuse the macro-level testbenches
  – the macro must be verified both as a standalone unit and in the context of the final application
• Testbenches must be compatible with system-level verification tools
  – testbenches may be used in system verification
Verification Plan

• Verification plan is part of the design report
• Verification takes over 70% of development time
• Contents
  – verification strategy for both subblock- and top-level
  – testbench components - BFM, bus monitors, …
  – required verification tools
  – simulation environment including block diagrams
  – key features needed to be verified in both levels
  – regression test environment and procedure
  – clear criteria to determine whether the verification is successfully complete
Benefits of Verification Plan

• Verification plan enables
  – developing the testbench environment early
  – developing the test suites early
  – focusing the verification effort to meet the shipment criteria
  – forcing designers to think through the time-consuming activities before performing them
  – a separate verification support team creates a verification environment in parallel with the primary design task
Verification Strategy

• Three major phases
  – subblock verification
    • thorough and exhaustive functionality verification
    • simulation, code coverage, TB automation
  – macro verification
    • interface verification between subblocks
    • simulation, hardware accelerator, TB automation, code coverage
  – prototyping
    • real prototype runs real software in the real application
    • emulation, FPGA, ASIC test chip

• **Bottom-up** approach
  – locality
  – catching bugs is *easier* and *faster* in the lower level
Types of Verification

- Compliance verification
- Corner case verification
- Assertion-based verification (ABV)
  - either static(formal) or dynamic
- Random verification
  - create scenarios that engineers do not anticipate
- Real code verification
  - avoid misunderstanding the spec
- Regression verification
  - ensure that fixing a bug will not introduce another bugs
  - be performed on a regular basis
Assertion

• An assertion
  – is a statement about a design’s intended behavior
  – is also named as property; used interchangeably
  – usually does not map to the real HW (not synthesizable)

• An assertion is used
  – to ensure consistency between the designer’s intention and what he created
  – to guard the design assumptions/properties

“Input should range from 0 to 240.”;
“After req raises, gnt is expected within 10 clock cycles.”
Neither New nor Innovative Idea

- The assertion concept has been adopted to software engineering for many many years

```c
#include <assert.h>

char hex_to_char(unsigned int num)
{
    assert(num <= 15);
    if(num < 10)
        return num + '0';
    else
        return num - 10 + 'A';
}
```

“Writing Solid Code” or “如何撰寫 0 錯誤程式”
Assertion-Based Verification

• Assertion
  – to preserve design assumptions and properties
  – break the simulation when assertion fails
  – both spatial and temporal relationship can be asserted
  – help designers to locate bugs at right place and time

• Open Verification Library (OVL)
  – www.verificationlib.org

• Property Specification Language (PSL)
  – derived from IBM’s Sugar, adopted by Accellera 2003

• Concept extended to function monitors and functional coverage
ABV Flow

- More thorough block-level verification
  - find bug easier
  - simulate effectively
- Build SoC from more robust IPs
  - assertions can be reused at system-level
  - system-level verification focuses on interfaces among IPs
Taxonomy

Functional Verification

Dynamic
- simulation based
- input vectors required
- no 100% guarantee (false positive issue)

Formal/Static
- mathematical-proof
- no input vectors required
- 100% guarantee

Semi-Formal
- still simulation based
- check properties formally at each simulation step

Model Checking
- verify properties formally
- capacity issue
- improvement expected

Equivalence Checking
- pretty mature

STA in timing verification
Verification Tools (1/2)

• Simulation
  – even-driven: good debug environment
  – cycle-based: fast simulation time

• Code coverage
  – coverage on RTL structure
  – Verification Navigator, CoverMeter

• Hardware Verification Language (HVL)
  – a language providing powerful constructs for generating stimulus and checking response
  – OpenVera in Vera, e in Specman Elite, SystemVerilog, (SCV in SystemC, former TestBuilder)
Verification Tools (2/2)

- Functional coverage
  - coverage on functionality
- Formal property checking
- Verification IPs (VIPs)
  - BFM}s and bus monitors for standard protocols
- Hardware modeling
- Hardware acceleration and emulation
- Prototyping
  - FPGA
  - test chip in silicon
Verification IPs

- A package including well-designed and well-verified BFM/monitor for a specific (interface) protocol
  - AMBA, Ethernet, SONET, UTOPIA, PCI, USB, I²C, UART, CAN, …
  - avoid reinvent-the-wheel
  - accelerate the verification
Code Coverage (1/2)

- Indicate how much of design has been exercised
- Point out what areas need additional verification
- Optimize regression suite runs
- Minimizes the use of valuable simulation resources
- Quantitative stopping criterion
- *Verify more but simulate less*
Code Coverage (2/2)

• Types
  – statement  – branch
  – condition  – path
  – toggle     – triggering
  – FSM

  – 100% coverage in statement, branch, condition
  – others can be used as secondary metrics

• Redundancy removal
• Minimize regression test suites
Verification with Code Coverage

![Graph showing the comparison between design verified with and without coverage over time. The graph indicates that verification with coverage results in higher percent design verified compared to verification without coverage. Feedback from coverage results is also shown.](image-url)
Bug Detection Curve

Rate of bugs found

Functional Testing

with coverage

without coverage

purgatory

release

Time

Source: Verification Methodology Manual
Inspection

- Finding bugs by careful inspection is faster than that by simulation
- Inspection process
  - design (specification, architecture) review
  - code (implementation) review
    - line-by-line fashion
    - at the subblock-level
- Lint-like tool can help spot defects w/o simulation
Adversarial Testing

• For original designers
  – focus on proving that the design works correctly

• Separate verification team
  – focus on trying to prove the design is broken
  – keep up with the latest tools and methodologies

• The combination of the two gives the best results
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
  – reducing the risk of supporting problems
Testbench Design

- Auto or semi-auto stimulus generation is preferred
- **Automatic response checking** is a must
- Powerful pattern generators and function monitors help
- Macro-level testbench
  - will be shipped along with the macro so that the customer can verify the macro in the system design (**reusable testbench**)
Example: USB Device Controller
Automatic Response Checking

Input Stimulus

Device Under Verification

Reference Model
(C/C++, HDL/HVL, Hardware Modeler)

Automatic Response Checking

Typical use: datapath elements and processors
Functional Models (1/2)

• Functional (Behavioral) model is required for all IPs
  – fast simulation and integration in system-level designs
  – a very secure model for customer’s evaluation
• BFM model is required particularly for interface IPs
  – PCI, USB, IEEE1394, ...
• ISA model is required for processor IPs
Functional Models (2/2)

• Functional models of different abstractions
  – provide *tradeoffs* between accuracy and simulation speed
  – meet various needs of the H/W and S/W teams in various design phases

• Model security
  – functional models usually developed by C/C++ or HDL
  – protection should be applied if security is a concern
  – compiled (encrypted) models are usually required
Timing Verification

- STA is the fastest approach for timing verification of synchronous designs
  - avoiding any timing exceptions is extremely important

- Gate-level simulation
  - most useful in verifying timing of asynchronous logic, multi-cycle paths and false paths
  - much slower run-time performance
  - gate-level sign-off
  - translate functional test patterns into tester patterns
Case Study 1:
16-Bit Embedded DSP Core
RTL Verification Environment

Assembly code
(*.dsp)

Execution file
(*.exe)

Memory Image files
(*.hexg)

S/W Tools

Verification Environment

BDMA I/F

ISA I/F

Core

PM
(64Kx24)

BDMA I/F

PLL

DM
(64Kx16)

BIST testing

CM
(64Kx16)

Interrupt SPORT GPIO I/O

Coverage measured by VN

copyright © 2004
Deterministic Verification

100+ deterministic verification suites

- Special functions
  - MAC, ALU, SHIFTER, DAG, SPORTs, Timer and Interrupts
- Pipeline control
  - data dependency, control dependency, and loop handling, ...
- Common DSP algorithms
  - FFT, IFFT, FIR, DEMEAN_BUFFER, ...
- Real code
  - DCT, PCM, modem, GSM codec, G.722, ...

Completeness assured by code coverage tool
Verification for External Interrupts

• One of the hard-to-verification cases
• Generating a complicated external interrupt sequence is not easy
  – scenarios such as nested interrupts, …
• Use an external interrupt generator
  – self-timing
  – highly configurable
  – help create non-trivial interrupt sequences
Instruction Set Simulator (ISS)

- Implement in C
- Cycle-by-cycle accuracy
  - all registers (internal registers, memory-mapped register,...)
  - important internal signals (program counter, stall signals, pipeline control signals, ...)

Verification Pattern → Hardware → H.log → ISS → I.log → Compare
IP Verification
Juinn-Dar Huang     jdhuang@mail.nctu.edu.tw

37
copyright © 2004

FPGA Prototyping Board
Application for FPGA Prototyping

- Bass enhancement

AD1847

UDSP-FPGA

7-SEGMENT

BOOT ROM
ASIC Prototyping

• More difficult to debug
  – lack of observability – limited external pins
  – lack of controllability – free-run mode only
MP3 Demo System
MP3 Player Demo System

UDSP-1600

AC’97 Codec

CF Card

Flash ROM
Development Kit

- 7-Seg. LEDs
- EPP Port
- Connectors
- Flash ROM
- UDSP-1600
- Audio I/F
Case Study 2:
32-Bit Embedded CPU Core
Design Verification

- Rigorous RTL coding style
  - assured by a linting tool
- Verifiable RTL coding style
  - friendly to simulation, equivalence checker, ...
- Deterministic verification
  - basic feature verification
  - corner case verification
  - code coverage analysis
- Random code verification
- Sophisticated regression environment
Deterministic Verification

- Data processing (ALU, shifter and multiplier)
- Memory access (load, store and swap)
- Branch and interrupt
- Pipeline data and control dependency
- Different CPU models
- Coprocessor interface

Completeness assured by code coverage tool
Random Code Verification (1/2)

Massive constrained random patterns

Random Code Generator

Random seed

Constraint file

Golden Model

CPU RTL

Result Comparator

Mismatch assertion

Information dump
Random Code Verification (2/2)

• Random code generator
  – constrained random instructions generation
    • instruction type
    • ratio among types
  – constrained random exception(interrupt) injection
    • exception type
    • probability for each exception
  – capable to rebuild failure scenarios for debugging

• Result comparator
  – automatically compare on the fly
  – all output and register values are compared
  – dump mismatch information for debugging
  – Over 100 million cycles verified at least
Must-Do List for My Next CPU Project
Must-Do List (1/2)

- Rigorous RTL coding and lint check
- Thorough verification plan in advance
- Deterministic verification
  - predefined must-verify features
  - corner-case verification
- Complete code coverage
- Constraint-driven random verification
  - constraint-driven randomized stimulus generator
  - use Vera or e
  - functional coverage
- Untimed and Timed ISA models
Must-Do List (2/2)

• Dynamic ABV
  – use assertions intensively throughout the simulation
  – use OVL and PSL

• Optional formal verification
  – property checking (model checking)
  – equivalence checking

• Optional HW accelerator

• FPGA prototyping
  – choose an appropriate application

• ASIC-based demo system
Untimed and Timed ISA Models (1/2)

- Untimed ISA model
  - instruction-level accuracy
  - no accurate timing information
  - for behavioral simulation
- Timed ISA model
  - cycle-accurate ISA model
  - functionality + cycle timing information
Untimed and Timed ISA Models (2/2)

- System C is a good candidate for above models
  - native and trivial for system designers
  - good encryption
  - can co-simulate with HDL
  - seamlessly integrate with GUI wrapper
- Timed model should be based on the untimed one
- ISS should be just a GUI wrapper + an ISA model
RTL Verification Flow

1. Linting
2. Simulation
   - Code Coverage
   - Functional Coverage
   - Dynamic ABV, ...
3. Property Checking
4. Synthesis
5. Equivalence Checking

Inputs:
- Verification Vectors
- Rules Database

Outputs:
- Gate-Level Netlist