System verification

What is verification?

• A process used to demonstrate the functional correctness of a design
• To make sure that you are indeed implementing what you want
• To make sure that the result of some transformations is as expected
Testing vs. verification

- **Testing verifies manufacturing**
  - Verify that the design was manufactured correctly

What is Driving Functional Verification?

- Verification requirements grows at a multiple of Moore's Law
  - 10X for ASICs
  - 100X for ASIC-based systems and SOCs which include embedded software
- Verification Complexity = \( f(\text{Architectural Complexity, Amount of Reuse, Clock Frequency, System Software Worst-Case}) \)
Verification bottleneck

Typical verification experience
Outline

- Conventional design and verification flow review

- Verification Techniques
  - Simulation
  - Formal Verification
  - Static Timing Analysis

- Emerging verification paradigms

Conventional Design Flow
Verification at different levels of abstraction

**Goal:** Ensure the design meets its functional and timing requirements at each of these levels of abstraction

In general this process consists of the following conceptual steps:

1. Creating the design at a higher level of abstraction
2. Verifying the design at that level of abstraction
3. Translating the design to a lower level of abstraction
4. Verifying the consistency between steps 1 and 3
5. Steps 2, 3, and 4 are repeated until tapeout
Verification Techniques

Goal: Ensure the design meets its functional and timing requirements at each of these levels of abstraction

- Simulation *(functional and timing)*
  - Behavioral
  - RTL
  - Gate-level (pre-layout and post-layout)
  - Switch-level
  - Transistor-level
- Formal Verification *(functional)*
- Static Timing Analysis *(timing)*
**Classification of Simulators**

- **Logic Simulators**
  - HDL-based
  - Event-driven
  - Cycle-based
  - Schematic-based
    - Gate
    - System

**HDL-based**: Design and testbench described using HDL
- Event-driven
- Cycle-based

**Schematic-based**: Design is entered graphically using a schematic editor

**Emulators**: Design is mapped into FPGA hardware for prototype simulation. Used to perform hardware/software co-simulation.
Event-driven Simulation

- Event: change in logic value at a node, at a certain instant of time $\rightarrow (V,T)$
- Event-driven: only considers active nodes
  - Efficient
- Performs both timing and functional verification
  - All nodes are visible
  - Glitches are detected
- Most heavily used and well-suited for all types of designs

Events:
- Input: $b(1)=1$
- Output: none

Events:
- Input: $b(1)=1$
- Output: $c(3)=0$
Event-driven Simulation

- Uses a timewheel to manage the relationship between components
- **Timewheel** = list of all events not processed yet, sorted in time (complete ordering)
- When event is generated, it is put in the appropriate point in the timewheel to ensure causality

Event-driven simulation flowchart
Event-driven Simulation

D=2

D=1

a
b
c
d
e

D=1

D=2

b(1)=1
d(5)=1

c(3)=0
d(5)=1
e(6)=1

Cycle-based Simulation

• Take advantage of the fact that most digital designs are largely synchronous

• Synchronous circuit: state elements change value on active edge of clock

• Only boundary nodes are evaluated
Cycle-based Simulation

• Compute steady-state response of the circuit
  – at each clock cycle
  – at each boundary node

Cycle-based versus Event-driven

• **Cycle-based:**
  – Only boundary nodes
  – No delay information

• **Event-driven:**
  – Each internal node
  – Need scheduling and functions may be evaluated multiple times

• Cycle-based is 10x-100x faster than event-driven (and less memory usage)
• Cycle-based does not detect glitches and setup/hold time violations, while event-driven does
(Some) EDA Tools and Vendors

• Logic Simulation
  – Scirocco (VHDL) → Synopsys
  – Verilog-XL (Verilog) → Cadence Design Systems
  – Leapfrog (VHDL) → Cadence Design Systems
  – VCS (Verilog) → Chronologic (Synopsys)

• Cycle-based simulation
  – SpeedSim (VHDL) → Quickturn
  – PureSpeed (Verilog) → Viewlogic (Synopsys)
  – Cobra → Cadence Design Systems
  – Cyclone → Synopsys

Simulation Testplan

• Simulation
  – Write test vectors
  – Run simulation
  – Inspect results

• About test vectors
  – HDL code coverage
Simulation-based verification

Consistency: same testbench at each level of abstraction

Testbench | Simulation |
---|---|
| Behavioral |
| RTL Design |
| Gate-level Design (Pre-layout) |
| Gate-level Design (Post-layout) |

Some Terminology

- Verification environment
  - Commonly referred as **testbench** (environment)
- Definition of a testbench
  - A verification environment containing a set of components [such as bus functional models (BFMs), bus monitors, memory modules] and the interconnect of such components with the design under-verification (DUV)
- Verification (test) suites (stimuli, patterns, vectors)
  - Test signals and the expected response under given testbenches
Coverage

- What is simulation coverage?
  - code coverage, FSM coverage, path coverage
  - not just a percentage number
- Coverage closes the verification loop
  - feedback on random simulation effectiveness
- Coverage tool should
  - report uncovered cases
  - consider dynamic behaviors in designs

Simulation Verification Flow

checker

RTL design

test bench

coverage

Yes/No
Coverage analysis helps

Coverage Pitfalls

- 100% coverage → verification done?
  - Code coverage only tells if a line is reached
- One good coverage tool is enough?
  - No coverage tool covers everything
- Coverage is only useful in regression?
  - coverage is useful in every stage
Coverage Analysis Tools

• Dedicated tools are required besides the simulator
• Several commercial tools for measuring Verilog and VHDL code coverage are available
  – VCS (Synopsys)
  – NC-Sim (Cadence)
  – Verification navigator (TransEDA)
• Basic idea is to monitor the actions during simulation
• Requires support from the simulator
  – PLI (programming language interface)
  – VCD (value change dump) files

Testbench automation

• Require both generator and predictor in an integrated environment
• Generator: constrained random patterns
  – Ex: keep A in [10 … 100]; keep A + B == 120;
  – Pure random data is useless
  – Variations can be directed by weighting options
  – Ex: 60% fetch, 30% data read, 10% write
• Predictor: generate the estimated outputs
  – Require a behavioral model of the system
  – Not designed by same designers to avoid containing the same errors
Conventional Simulation Methodology Limitations

- Increase in size of design significantly impact the verification methodology in general
  - Simulation requires a very large number of test vectors for reasonable coverage of functionality
  - Test vector generation is a significant effort
  - Simulation run-time starts becoming a bottleneck

- New techniques:
  - Static Timing Analysis
  - Formal Verification

New Verification Paradigm

- **Functional**: cycle-based simulation and/or formal verification
- **Timing**: Static Timing Analysis
Types of Specifications

- **Requirements**
  - design should satisfy
  - Requirements are precise: a must for formal verification

**Specification**

- **Informal**
  - Equivalence
    - Is one design equivalent to another?

- **Formal**
  - Properties
    - Design has certain good properties?

---

Formal vs Informal Specifications

- **Formal requirement**
  - No ambiguity
  - Mathematically precise
  - Might be executable

- **A specification can have both formal and informal requirements**
  - Processor multiplies integers correctly (formal)
  - Lossy image compression does not look too bad (informal)
Formal Verification

- Can be used to verify a design against a reference design as it progresses through the different levels of abstraction
- Verifies functionality without test vectors
- Three main categories:
  - Model Checking: compare a design to an existing set of logical properties (that are a direct representation of the specifications of the design). Properties have to be specified by the user (far from a “push-button” methodology)
  - Theorem Proving: requires that the design is represented using a “formal” specification language. Present-day HDL are not suitable for this purpose.
  - Equivalence Checking: it is the most widely used. It performs an exhaustive check on the two designs to ensure they behave identically under all possible conditions.

Formal Verification vs Informal Verification

Formal Verification
- Complete coverage
- Effectively exhaustive simulation
- Cover all possible sequences of inputs
- Check all corner cases
- No test vectors are needed

Informal Verification
- Incomplete coverage
- Limited amount of simulation
- Spot check a limited number of input seq’s
- Some (many) corner cases not checked
- Designer provides test vectors (with help from tools)
Complete Coverage Example

- For these two circuits:
  \[ f = ab(c+d) \]
  \[ = abc + abd \]
  \[ = g \]
  - So the circuits are equivalent for all inputs
- Such a proof can be found automatically
  - No simulation needed

Using Formal Verification

- No test vectors
- Equivalent to exhaustive simulation over all possible sequences of vectors (complete coverage)
Symbolic simulation

- Simulate with Boolean formulas, not 0/1/X
- Example system:

\[ a \oplus b \oplus c \]

- Example property: \[ x = a \oplus b \oplus c \]

Verification engine: Boolean equivalence (hard!)

Why is this formal verification?

Simulating sequential circuits

Property:

if \( r_0 = a \), \( z_0 = b \), \( z_1 = c \)
then \( r_2 = a \oplus b \oplus c \)

Symbolic evaluation:

\[
\begin{align*}
r_0 &= a \\
r_1 &= a \oplus b \\
r_2 &= (a \oplus b) \oplus c
\end{align*}
\]

Limitation: can only specify a fixed finite sequence
Model checking

Verification engine: state space search (even harder!)

Advantage: greater expressiveness
(but model must still be finite-state)

First order decision procedures

- Handles even non-finite-state systems
- Used to verify pipeline equivalence
- Cannot handle temporal properties
Increasing automation

• Handle larger, more complex systems
• Boolean case
  – Binary decision diagrams
    • Boolean equivalence in symbolic simulation
    • Symbolic model checking
  – SAT solvers
• State space reduction techniques
  – partial order, symmetry, etc.
• Fast decision procedures
  
  Very hot research topics in last decade, but still do not scale to large systems.

Scaling up

• The compositional approach:
  – Break large verification problems into smaller, localized problems.
  – Verify the smaller problems using automated methods.
  – Verify that smaller problems together imply larger problem.
Example -- equivalence checkers

- Identify corresponding registers
- Show corresponding logic “cones” equivalent
  - Note: logic equivalence ≈ symbolic simulation
- Infer sequential circuits equivalent
  That is, local properties ⇒ global property

Abstraction

- Hide details not necessary to prove property
- Two basic approaches
  - Build abstract models manually
    abstract model ⇒ property
    abstraction relation
    system ⇒ property
  - Use abstract interpretation of original model
Examples of abstraction

- Hiding some components of system
- Using X value in symbolic simulation
- One-address/data abstractions
- Instruction-set architecture models

All are meant to reduce the complexity of the system so that we can simplify the verification problem for automatic tools.

Decomposition and abstraction

- Abstractions are relative to property
- Decomposition means we can hide more information.
- Decomposed properties are often relative to abstract reference models.
Equivalence Checking tools

- Structure of the designs is important
  - If the designs have similar structure,
  - then equivalence checking is much easier
- More structural similarity at low levels of abstraction

Degree of Similarity: State Encoding

- Two designs have the same state encoding if
  - Same number of registers
  - Corresponding registers always hold the equal values
- Register correspondence a.k.a. register mapping
  - Designs have the same state encoding if and only if
  - there exists a register mapping
- Greatly simplifies verification
  - If same state encoding,
  - then combinational equivalence algorithms can be used
Producing the Register Mapping

- By hand
  - Time consuming
  - Error prone
  - Can cause misleading verification results
- Side-effect of methodology
  - Mapping maintained as part of design database
- Automatically produced by the verification tool
  - Minimizes manual effort
  - Depends on heuristics

Degree of Similarity: Combinational Nets

- Corresponding nets within a combinational block
  - Corresponding nets compute equivalent functions
- With more corresponding nets
  - Similar circuit structure
  - Easier combinational verification
- Strong similarity
  - If each and every net has a corresponding net in the other circuit,
  - then structural matching algorithms can be used
Degree of Similarity: Summary

Weak Similarity

- Different state encodings
  - General **sequential equivalence** problem
  - Expert user, or only works for small designs
- Same state encoding, but combinational blocks have different structure
  - IBM’s BoolsEye
  - Compass’ VFormal
- Same state encoding and similar combinational structure
  - Chrysalis (but weak when register mapping is not provided by user)
- Nearly identical structure: **structural matching**
  - Compare gate level netlists (PBS, Chrysalis)
  - Checking layout vs schematic (LVS)

Strong Similarity

Capacity of a Comb. Equiv. Checker

- Matching pairs of fanin cones can be verified separately
  - How often a gate is processed is equal to the number of registers it affects
  - Unlike synthesis, natural subproblems arise without manual partitioning
  - “Does it handle the same size blocks as synthesis?” is the wrong question
  - “Is it robust for my pairs of fanin cones?” is a better question
- Structural matching is easier
  - Blocks split further (automatically)
  - Each gate processed just once
Main engine: combinational equivalence

• For these two circuits:
  \[ f = ab(c+d) \]
  \[ = abc + abd \]
  \[ = g \]

In practice:
1. Expression size blowup
2. Expressions are not canonical

Binary Decision Diagrams

• Binary Decision Diagrams are a popular data structure for representing Boolean functions
  – Compact representation
  – Simple and efficient manipulation

[Bry86]
Example:

BDD construction for $F=(a+b)c$

$F = a' \cdot F_{a=0}(b,c) + a \cdot F_{a=1}(b,c)$

$= a' \cdot (bc) + a \cdot (c)$

$(bc) = b' \cdot (0) + b \cdot (c)$

Two construction rules

• ORDERED
  variables must appear in the same order
  along all paths from root to leaves

• REDUCED
  1. Only one copy for each isomorphic sub-
     graph
  2. Nodes with identical children are not
     allowed
Reduction rule 1.: Only one copy for each isomorphic sub-graph

Reduction rule 2.: Nodes with identical children are not allowed

(We built it reduced from the beginning)
Nice implications of the construction rules

Reduced, Ordered BDDs are **canonical** that is, some important problems can be solved in constant time:

1. **Identity checking**
   
   \((a+b)c\) and \(ac+bc\) produce the same identical BDD

1. **Tautology checking**
   
   just check if BDD is identical to function

1. **Satisfiability**
   
   look for a path from root to the leaf

BDD summary

- Compact representation for Boolean functions
- Canonical form
- Boolean manipulation is simple
- Widely used
Equivalence Checking: Research

- Early academic research into tautology checking
  - A formula is a tautology if it is always true
  - Equivalence checking: \( f \) equals \( g \) when \( (f = g) \) is a tautology
  - Used case splitting
  - Ignored structural similarity often found in real world
- OBDDs [Bryant 1986]
  - Big improvement for tautology checking [Malik et. al 1988, Fujita et. al 1988, Coudert and Madre et. al 1989]
  - Still did not use structural similarity
- Using structural similarity
  - Combine with ATPG methods [Brand 1993, Kunz 1993]
  - Continuing research on combining OBDDs with use of structural similarity

What is Static Timing Analysis?

- \( \text{STA} = \text{static timing analysis} \)
- \( \text{STA} \) is a method for determining if a circuit meets timing constraints without having to simulate
- No input patterns are required
  - 100% coverage if applicable
Static Timing Analysis

• Suitable for synchronous design

• Verify timing without testvectors

• Conservative with respect to dynamic timing analysis

Static Timing Analysis

• Inputs:
  – Netlist, library models of the cells and constraints (clock period, skew, setup and hold time…)

• Outputs:
  – Delay through the combinational logic

• Basic concepts:
  – Look for the longest topological path
  – Discard it if it is false
Timing Analysis - Delay Models

- **Simple model 1:**
  \[ A_k = \text{arrival time} = \max(A_1, A_2, A_3) + D_k \]
  
  \( D_k \) is the delay at node \( k \), parameterized according to function \( f_k \) and fanout node \( k \).

- **Simple model 2:**
  \[ A_k = \max\{A_1 + D_{k1}, A_2 + D_{k2}, A_3 + D_{k3}\} \]

- Can also have different times for rise and fall time.

---

Static delay analysis

```plaintext
// level of PI nodes initialized to 0, // the others are set to -1.
// Invoke LEVEL from PO
Algorithm LEVEL(k) { // levelize nodes
    if( k.level != -1)
        return(k.level)
    else
        k.level = 1+\max\{LEVEL(k_i)|k_i \in \text{fanin}(k)\}
    return(k.level)
}

// Compute arrival times:
// Given arrival times on PI’s
Algorithm ARRIVAL() {
    for L = 0 to MAXLEVEL
        for \{k|k.level = L\}
            A_k = \max\{A_{k_i}\} + D_k
}
```
An example of static timing analysis

(Some) EDA Tools and Vendors

• **Formal Verification**
  – Formality → Synopsys
  – FormalCheck → Cadence Design Systems
  – DesignVerifier → Chrysalis

• **Static Timing Analysis**
  – PrimeTime → Synopsys (gate-level)
  – PathMill → Synopsys (transistor-level)
  – Pearl → Cadence Design Systems
The ASIC Verification Process

Overview Emerging Challenges

- Conventional design flow → Emerging design flow
  - Higher level of abstraction
  - More accurate interconnect model
  - Interaction between front-end and back-end
- Signal Integrity
- Reliability
- Power
- Manufacturability

Paradigm: Issues must be addressed early in the design flow – no more clear logical/physical dichotomy
⇒ New generation of design methodologies/tools needed
Emerging issues

• **Signal Integrity** (SI) : Ensure signals travel from source to destination without significant degradation
  – Crosstalk: noise due to interference with neighboring signals
  – Reflections from impedance discontinuity
  – Substrate and supply grid noise

More emerging issues

• **Reliability**
  – Electromigration
  – Electrostatic Discharge (ESD)

• **Manufacturability**
  – Parametric yield
  – Defect-related yield
More emerging issues

• Power
  – Power reduction at RTL level and at gate level
    • Library-level: use of specially designed low-power cells
    • Design technique
  – It is critical that power issues be addressed early in the design process (as opposed to late in the design flow)
  – Power tools:
    • Power estimation: (Design Power - Synopsys)
    • Power optimization: take into consideration power just as synthesis uses timing and area (Power Compiler - Synopsys)

SoC verification

• Large-scale
  – Build with a number of components (HW & SW)
• Not only hardware
  – HW
  – SW
  – Their interaction
SoC verification flow

- Verify the leaf IPs
- Verify the interface among IPs
- Run a set of complex applications
- Prototype the full chip and run the application software
- Decide when to release for mass production

Finding/fixing bugs costs

- Design integration stage
- Time to fix a bug

• Chip NREs increasing making respins an unaffordable proposition
  - Average ASIC NRE ~$122,000
  - SOC NREs range from $300,000 to $1,000,000
The usefulness of IP verification

- 90% of ASICs work at the first silicon but only 50% work in the target system
  - Problem with system level verification (many components)
- If a SoC design consisting of 10 blocks
  - $P(\text{work}) = .9^{10} = .35$
- If a SoC design consisting of 2 new blocks and 8 pre-verified robust blocks
  - $P(\text{work}) = .9^2 \times .99^8 = .69$
- To achieve 90% of first-silicon success SoC
  - $P(\text{work}) = .99^{10} = .90$

Checking functionality

- Verify the whole system by using full functional models
- Test the system as it will be used in the real world
- Running real application codes (such as boot OS) for higher design confidence

RTL simulation is not fast enough to execute real applications
Dealing with complexity

- Solutions
  - Move to a higher level of abstraction for system functional verification
- Formal verification
- Use assistant hardware for simulation speedup:
  - Hardware accelerator
  - ASIC emulator
  - Rapid-prototyping (FPGA)

Hardware-Software Cosimulation

- Couple a software execution environment with a hardware simulator
- Simulate the system at higher levels
  - Software normally executed on an Instruction Set Simulator (ISS)
  - A Bus Interface Model (BIM) converts software operations into detailed pin operations
- Allows two engineering groups to talk together
- Allows earlier integration
- Provide a significant performance improvement for system verification
  - Has gained popularity
Co-simulation

Homogenous/Heterogenous

Product SW

ISS (optional)

Product SW

Hardware Implementation

VHDL  Verilog

Simulation algorithm

Event  Cycle  Dataflow

Simulation Engine

Emulator

HW-level cosimulation

- Detailed Processor Model:
  - processor components (memory, datapath, bus, instruction decoder etc) are discrete event models as they execute the embedded software.
  - Interaction between processor and other components is captured using native event-driven simulation capability of hardware simulator.
  - Gate level simulation is extremely slow (~tens of clock cycles/second), behavioral model is ~hundred times faster. Most accurate and simple model

Gate-Level HDL software

ASIC Model (VHDL Simulation)

(Backplane)
ISS+Bus model

• Bus Model (Cycle based simulator):
  – Discrete-event shells that only simulate activities of bus interface without executing the software associated with the processor. Useful for low level interactions such as bus and memory interaction.
  – Software executed on ISA model and provide timing information in clock cycles for given sequence of instructions between pairs of IO operation.
  – Less accurate but faster simulation model.

Compiled ISS

• Compiled Model:
  – very fast processor models are achievable in principle by translating the executable embedded software specification into native code for processor doing simulation. (Ex: Code for programmable DSP can be translated into Sparc assembly code for execution on a workstation)
  – No hardware, software execution provides timing details on interface to cosimulation.
  – Fastest alternative, accuracy depends on interface information.
HW-assisted cosimulation

- Hardware Model:
  - If processor exists in hardware form, the physical hardware can often be used to model the processor in simulation. Alternatively, processor could be modeled using FPGA prototype. (say using Quickturn)
  - Advantage: simulation speed
  - Disadvantage: Physical processor available.

- ASIC Model (VHDL Simulation)

FPGA Processor

ASIC Model (VHDL Simulation)

(Backplane)

Cosimulation engines: Master slave cosimulation

- One master simulator and one or more slave simulators: slave is invoked from master by procedure call.
- The language must have provision for interface with different language
- Difficulties:
  - No concurrent simulation possible
  - C procedures are reorganized as C functions to accommodate calls

Master
HDL
HDL Interface

Slave
C simulator
Distributed cosimulation

- Software bus transfers data between simulators using a procedure calls based on some protocol.
- Implementation of System Bus is based on system facilities (Unix IPC or socket). It is only a component of the simulation tool.
- Allows concurrency between simulators.

Alternative approaches to co-verification

- Static analysis of SW
  - Worst-case execution time (WCET) analysis
  - WCET with hardware effects
- Software verification
Static analysis for SW: WCET

• Dynamic WCET analysis
  – Measure through running a program on a target machine with all possible inputs
  – Not feasible - the input is for worst case??
• Static WCET analysis
  – Derive approximate WCET from source code by predict the value and behavior of program that might occur in run time

Approximate WCET

• Not easy to get the exact value
  – Trade-off for exactness and complexity
• But, must be safe and had better be tight
Static WCET analysis

• Basic approach
  – Step1: Build the graph of basic blocks of a program
  – Step2: Determine the time of each basic block by adding up the execution time of the machine instructions
  – Step3: Determine the WCET of a whole program by using Timing Schema
    - $\text{WCET}(S_1; S_2) = \text{WCET}(S_1) + \text{WCET}(S_2)$
    - $\text{WCET}(\text{if } E \text{ then } S_1 \text{ else } S_2) = \text{WCET}(E) + \max(\text{WCET}(S_1), \text{WCET}(S_2))$
    - $\text{WCET}(\text{for}(E) S;) = (n+1)\text{WCET}(E) + n\text{WCET}(S_1)$ where $n$ is loop bound

Example with a simple program

<Source Code>

```
While (I<10){
    If(I<5) B
    j=j+2;
    Else
        k=k+10; D
    If(I>50) E
    m++; F
    I++; G
}
```

Diagram:

```
A --> B --> C
     |       |
     v       v
D --> E --> F --> G
```

feasible??  Loop bound??
Static WCET analysis

- High-level (program flow) analysis
  - To analyze possible program flows from the program source
    - Paths identification, loop bound, infeasible path etc.
  - Manual annotation – compiler optimization
  - Automatic derivation
- Low-level (machine level) analysis
  - Determine the timing effect of architectural features such as pipeline, cache, branch prediction etc.

Low-level analysis

- The instructions’ execution time in RISC processor varies depending on factors such as pipeline stall or cache miss/hit due to the pipelined execution and cache memory.
  - In the pipelined execution, an instruction's execution time varies depending on surrounding instructions.
  - With cache, the execution time of a program construct differ depending on which execution path was taken prior to the program construct.

[S.-S. Lim et al., An accurate worst case timing analysis for RISC processors, IEEE Transactions on Software Engineering, vol. 21, Nr. 7, July 1995]
Pipeline and cache analysis

- Program construct keeps timing information of every worst case execution path of the program construct.
  - the factors that may affect the timing of the succeeding program construct
  - the information that is needed to refine WCET when the timing information of preceding construct is known.

Difficulty of Static WCET analysis

- WCET research, Active research area but not yet practical in industry
  - Limits of automatic path analysis
  - Too complex analysis
- Bytecode analysis
- Writing predictable code? a single path program whose behavior is independent of input data
  - No more path analysis
  - Gain WCET time by exhaustive measurement

[Peter Puschner, Alan Burns, Writing Temporally Predictable Code, IEEE International Workshop on Object-Oriented Real-Time Dependable Systems]
Debugging embedded systems

• Challenges:
  – target system may be hard to observe;
  – target may be hard to control;
  – may be hard to generate realistic inputs;
  – setup sequence may be complex.

Software debuggers

• A monitor program residing on the target provides basic debugger functions.
• Debugger should have a minimal footprint in memory.
• User program must be careful not to destroy debugger program, but should be able to recover from some damage caused by user code.
Breakpoints

- A breakpoint allows the user to stop execution, examine system state, and change state.
- Replace the breakpointed instruction with a subroutine call to the monitor program.

ARM breakpoints

- Uninstrumented code:
  - 0x400  MUL r4,r6,r6
  - 0x404  ADD r2,r2,r4
  - 0x408  ADD r0,r0,#1
  - 0x40c  B loop

- Code with breakpoint:
  - 0x400  MUL r4,r6,r6
  - 0x404  ADD r2,r2,r4
  - 0x408  ADD r0,r0,#1
  - 0x40c  BL bkpoint
Breakpoint handler actions

• Save registers.
• Allow user to examine machine.
• Before returning, restore system state.
  – Safest way to execute the instruction is to replace it and execute in place.
  – Put another breakpoint after the replaced breakpoint to allow restoring the original breakpoint.

In-circuit emulators

• A microprocessor in-circuit emulator is a specially-instrumented microprocessor.
• Allows you to stop execution, examine CPU state, modify registers.
Testing and Debugging

- ISS
  - Gives us control over time – set breakpoints, look at register values, set values, step-by-step execution, ...
  - But, doesn’t interact with real environment
- Download to board
  - Use device programmer
  - Runs in real environment, but not controllable
- Compromise: emulator
  - Runs in real environment, at speed or near
  - Supports some controllability from the

Logic analyzers

- Debugging on final target
- A logic analyzer is an array of low-grade oscilloscopes:
Logic analyzer architecture

How to exercise code

• Run on host system.
• Run on target system.
• Run in instruction-level simulator.
• Run on cycle-accurate simulator.
• Run in hardware/software co-simulation environment.
Trace-driven performance analysis

- **Trace**: a record of the execution path of a program.
- Trace gives execution path for performance analysis.
- A useful trace:
  - requires proper input values;
  - is large (gigabytes).

Trace generation

- **Hardware capture**:
  - logic analyzer;
  - hardware assist in CPU.
- **Software**:
  - PC sampling.
  - Instrumentation instructions.
  - Simulation.