Review: Design Modeling

- **Behavior model**
  - functional model
  - limited timing information
  - describe I/O behavior
- **Timing**
  - given inputs, when outputs change

- **RTL Model**
  - functional + timing + latches
  - \( c = a + b \)

- **Structural RTL Model**
  - functional + timing + latches + primitives
  - primitive: and, or, MUX, nmos, pmos, etc.
  - \( c = a + b \)
    - ripple-carry adder
    - carry-look-ahead adder
Sequential model

- We need: functional model for combinational logic + a state transition diagram
  - $s = a \oplus b \oplus L$
  - $c = ab + aL + bL$

$# of states = 2^M$

How to represent all reachable states?

Recall: verification demand

- Functional verification cost grows faster than design complexity

A simple example of why

- If you have a design with 10-bit flipflops, the maximum number of states is 1024
- If it is 11-bit, the number of states is 2048
- So in theory, the verification space grows exponentially while the design grows linearly

(Note: this view can be misleading, why?)
Resource spending

- Functional Verification
- Post-Silicon Validation

Verification approaches

- Vector (test bench) Simulation
  - Define correctness
  - Test case generation
  - Coverage metric
    - Assertion-based properties
  - Hardware emulation
- Formal Verification
  - Logic Equivalent Checking
    - RTL to Gate, RTL to Schematics
  - Model checking
- "Semi-formal" Approach
  - Symbolic simulation
  - Effective for array verification

Challenges in verification

- Specification
  - Correctness may not be well-defined
  - This is especially true at unit level
- Size and Complexity
  - Vector simulation remains the only effective way for full-chip functional verification
  - Model checking remains limited
  - Symbolic simulation remains limited
- Verification re-use
- Quality Concerns
  - Never enough
  - Depends on experience and time-to-market
Verification relies on simulation

- Full-chip functional verification relies on vector simulation
  - Formal method is not powerful enough yet
- Test vectors (testbench) are prepared in a semi-random fashion
  - To test an adder:
    - (normal flow) supply random numbers
    - (boundary conditions) also make sure to supply the maximum and minimum numbers
      - to test the carry output

Level of simulation

- RTL
  - low complexity
  - verify function
  - zero delay model
- Gate
  - zero/unit delay model
  - fault injection and simulation
- Transistor
  - unit switching delay model
  - transistor strengths/sizes included
  - verify custom circuit designs

Verification by simulation

- Checking for error conditions (bus contention)
- Ability to change delays to test worst-case timing
- Checking user-specified expected values
- Starting simulation at any stage
Logic simulation

- **Cycle-based simulation**
  - simulate from latches to latches
  - provide cycle-accurate information
  - suitable for clock-based synchronous design
  - used for processor verification
  - involve zero delay

- **Event-driven simulation**
  - simulate only newly-generated events
  - "input changes" => "output changes"
  - involve unit delay
  - more general but slower

---

**Event-Driven Simulation**

---

**Simulation with unit delay**
TestBench in Verilog

```verilog
module Nand_Latch_1 (q, qbar, preset, clear);
output q, qbar;
input preset, clear;

nand G1 (q, preset, bar),
G2 (qbar, clear, q);
endmodule

module test_Nand_Latch_1; // Design Unit Testbench

reg preset, clear;
wire q, qbar;
Nand_Latch_1 M1 (q, qbar, preset, clear); // Instantiate UUT

initial // Create DUTB response monitor
begin
$monitor ($time, "preset = %b clear = %b q = %b qbar = %b", preset, clear, q, qbar);
end

initial begin
// Create DUTB stimulus generator
#10 preset =0; clear = 1;
#10 preset =1;
$stop; // Enter to proceed
#10 clear =0;
#10 clear =1;
#10 preset =0;
end

initial #60 $finish; // Stop watch
endmodule
```

Simulation Results

<table>
<thead>
<tr>
<th>tsim</th>
<th>preset</th>
<th>clear</th>
<th>q</th>
<th>qbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

3-Value simulation

- 0, 1, X

![3-Value simulation diagram]

- X and X-bar

- Don't care and Don't know
  - at inputs - don't care
    - if simulation results are right, X inputs are "don't care"
  - inside - don't know
Detailed simulation/emulation

- Switch-level simulation
  - to check individual custom blocks
  - provide more accurate information
    - transistor strengths
    - timing
    - dynamic logic behavior
- Spice simulation
  - to check cells and interconnects
  - very time-consuming
- Hardware emulation
  - use FPGA
  - not a cheap solution

Considerations in simulation

- Test stimuli
  - manual generation
  - pseudo-random
- Definition of correctness
  - manual observation
  - manual construction of monitor program
- Effectiveness of tests
  - try to exhaust all interesting cases
  - rely on empirical data
- Length of simulation
  - billions of cycles

For processors - RTPG

- For processor verification
- RTPG – Random Test Program Generation
- Basic Ideas
  - Symbolic execution
  - Biasing inputs
- RTPG’s job is to ensure the instruction sequences are meaningful
  - Hide many details from user
  - Allow user to specify constraints and biases
Observability is a key issue
- Decide where you want and you can check the results of simulation
  - You need to know the expected behavior first
- Logical registers may not be the same as physical registers
  - Register renaming depends on the implementation
- Logical cache may not be the same as physical cache structure
  - 1 level cache vs. 3 level cache
- Memory is the only “safe” observation point

RTPG Methodology
- Billions of cycles are simulated
- Verification coverages are systematically measured based on a collection of known events
- A subset of tests are selected for testing defects as well

When to stop?
- Types of Biasing Inputs
  - Single instruction set; Instruction pairs; 3-instruction set; Selected sequences (experience + new ideas); Biased random sequences
- Coverage Measurement
  - Statement coverage; Toggle coverage; error discovery rate
- Tape-out Criteria
  - An empirical decision
Recall: Pre-silicon Validation cycles

Verification crisis

- More than 50% of the project budget already goes to verification
- Simulation and testbench preparation time already drags the time-to-market
- Design complexity grows tremendously with the use of IP cores
- Cost of chip re-spin is high
  - > $100K for ASIC
  - > $1M for a complex SOC

Easy vs. hard bugs

- Engineers spend tremendous amount of time to develop the environment, building the models
  - Need good software environment, API, etc.
- Engineers waste time to capture "easy" errors
  - Some automation is desperately needed
- They should focus on capturing "hard" ones
Divide and conquer

- As design becomes extremely complex, fixing a bug becomes more tedious
- If we start RTPG directly on the whole chip,
  - We spend a lot of time to fix bugs in some units
  - Yet some other units receive less attention

Unit level verification

- A chip is usually divided into 6-7 units
- Goal: detect 95-99% of the design errors at unit level and the remaining 1-5% at the full-chip level
- Uni-Sim allows individual unit to be verified independently from the completion of other units
Question

- Unit-level verification makes sense for a microprocessor design
  > Where the functionality of each unit is somewhat well defined
  > The unit-level verification environment can be re-used for the next generation

- What if we are verifying an SOC?
  > Do we want to invest in building a unit-level verification environment for every unit and maybe we can use them once?

Additional questions

- What is our unit-level verification strategy?
- How do we generate functional tests?
- How do we check for correctness?
- When we move to full chip of an SOC, what is to be verified?
  > What is our verification space, different from unit-level verification?
  > It should be simpler, isn’t it?
- How to re-use verification results?

Example: Motorola MPC8540

[Diagram of Motorola MPC8540 chip]
Unified Or Disparate Methodology

- A Unified methodology involves a single test generator for a project
  - Users do not know how tests are generated
  - Same test generator covers different configurations

- A Disparate methodology involves a dedicated test generator for each system configuration
  - End users require in-depth knowledge
  - Software maintenance is difficult

TestBench

- Dedicated testbench for each individual unit

- A testbench consists of:
  - Drivers: Translating unit-to-unit transaction-level commands to pin signals
  - Monitors: Monitoring responses on the interfaces and reporting violations
  - Behaviors: Monitoring system operations via monitors and report system failures

- Allow maximal testbench re-use

A Conceptual Example
Verification of Unit C

Verification of Unit D

Verification of C and D w.r.t. A and B
Constrained Verification

Correctness is usually monitored through *Assertions* in the design.

A Test Template Example

```c
Trans[2] {
    Master = range(0, 2)
    Slave = namedbias(2 => "A", 1 => "B")
    Dir = equalbias("read", "write")
    Burst = bias(1, 2)
    Inhibit = bias(2, 1)
    if (P(".Dir") eq "read") {
        Delay = range(0, 4)
    }
} - P("Trans[*].Burst") == 0 && 
  P("Trans[*].Inhibit") == 1
+ P("Trans[*].Slave") eq "A"
```

Success Stories

- A SoC based on the Motorola PowerPC architecture
- Initial approach was a C++ based test generator
- Backtracking was very difficult to implement, which severely restricted coverage
- Transgen overcomes these problems

<table>
<thead>
<tr>
<th></th>
<th>C++ impl.</th>
<th>Transgen impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines (NBNC)</td>
<td>38697</td>
<td>3703</td>
</tr>
<tr>
<td>Development Effort</td>
<td>4 p-years</td>
<td>2 p-months</td>
</tr>
<tr>
<td>Generation speed</td>
<td>5-6 trans/sec</td>
<td>2-3 trans/sec.</td>
</tr>
<tr>
<td>Platform-independent</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

MTV 2002, Nodine, et. al.
Assertion-Based Verification

- Support for assertion-based verification
  - Open Vera Assertion (OVA) – Synopsys
  - System Verilog
  - Sugar – IBM formal property language
  - Etc.

- Vendors often provide assertion libraries
  - OVA
  - Open Verilog Library (OVL):
    - [http://www.verificationlib.org](http://www.verificationlib.org)

- Assertion monitoring does help to quickly locate design bugs (everybody says so)

---

An assertion example (OVL)

```verilog
// ASSERT_NO_OVERFLOW
// NAME: ASSERT_NO_OVERFLOW - An invariant concurrent assertion to ensure
// an expression (or variable) does not exceed
// a MAX value or reach a value <= MIN limit.
// inputs:
// ----------------------
// paraseters:
// ----------------------
// module assert_no_overflow (clk, reset_n, test_expr);
parameter severity_level = 0;
parameter width=1;
parameter min=0;
parameter max= ((1<<width)-1);
parameter options = 0;
parameter msg="VIOLATION";
input clk, reset_n;
input [width-1:0] test_expr;
`ifdef ASSERT_ON
`ifdef ASSERT_OVL_VERILOG
`else
`define ASSERT_OVL_VERILOG
`endif
`ifdef ASSERT_GLOBAL_RESET
`define ASSERT_RESET_SIGNAL `ASSERT_GLOBAL_RESET
`else
`define ASSERT_RESET_SIGNAL reset_n
`endif
`ifdef ASSERT_OVL_PSL
/* psl
property ASSERT_NO_OVERFLOW =
always ((`ASSERT_RESET_SIGNAL  != 1'b0) ->
(never {test_expr == max;(test_expr > max) ||
(test_expr <= min)})
abort (`ASSERT_RESET_SIGNAL  == 1'b0))
@posedge clk);
assert ASSERT_NO_OVERFLOW;
*/
`endif  // ASSERT_OVL_PSL
//synopsys translate_off
`ifdef ASSERT_OVL_VERILOG
// local parameters used as defines
parameter OVERFLOW_START = 1'b0;
parameter OVERFLOW_CHECK = 1'b1;
reg r_state;
initial r_state=OVERFLOW_START;
parameter assert_name = "ASSERT_NO_OVERFLOW";
integer error_count;
initial error_count = 0;
`include "ovl_task.h"
```
```
Assertion Example (OVL)

```verilog
//ifdef ASSERT_INIT_MSG
initial
begin
    // Call the User Defined Init Message Routine
    $display("Message: ");
end
always @(posedge clk) begin
    if (`ASSERT_RESET_SIGNAL != 1'b0) begin
        case (r_state)
            OVERFLOW_START:
                if (test_expr == max) begin
                    r_state <= OVERFLOW_CHECK;
                end
            OVERFLOW_CHECK:
                if (test_expr != max) begin
                    r_state <= OVERFLOW_START;
                    if (test_expr <= min || test_expr > max) begin
                        ovl_error("\$display("'");
                    end
                end
            endcase
        end
    end // always
    else begin
        r_state <= OVERFLOW_START;
    end
endmodule
```

Assertion Example (OVA)

```verilog
// define basic events
bool read_setup: sel & enable & write
bool read_enable: sel & enable & read
bool write_setup: sel & enable & read
bool write_enable: sel & enable & write
bool idle: ~sel & ~enable

clock posedge clk {
    event read: if (read_setup) #2 read_enable;
    event write: if (write_setup) #1 write_enable;
}
assert read_a : check(read);
assert write_a : check(write);
```

Transaction-Based Verification

- **RapidIO design**
  - RTL model to describe the hardware
  - Transaction definitions to describe the transactions allowed
  - Assertions in place to describe design properties and to monitor simulation
- All of above are part of the design

<table>
<thead>
<tr>
<th>Read(data, address)</th>
<th>Write(data, address)</th>
<th>Transaction Layer of abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101 1000010100</td>
<td>01010101 111010100</td>
<td>0000101 10001010 1010101 111010100</td>
</tr>
</tbody>
</table>
Verification Re-use

- Re-use of test templates
  - Especially true in microprocessor verification
- Verification IPs
  - Built-in assertion with IP cores
  - Pre-built verification environment for IPs

Verification IP - Re-use

- For an IP core, its verification environment can be re-used as well
  - Assertion re-use
  - Transaction definition re-use
  - Testbench re-use
- IP users only need to deal with application specific transactions to develop their own testbench

[Diagram showing IP users interacting with application, transaction test sequences, transaction test sequences, hardware signals, and simulation engine]

Coverage report

Application specific transaction test