Chapter 2 Logic Simulation
Logic Simulation

- **Purposes**
  - Verification
  - Debugging
  - Studying design alternative (cost/speed)
  - Computing expected behavior for tests

- **Simulation-based design verification**
  - To check correct operations:
    - e.g. delays of critical paths
    - free of critical races & oscillation
  - Problem is that tests are hand crafted; Very hard to prove that a test is complete.
  - Formal and assertion-based verification required
Modeling for Circuit Simulation

- Circuit models
  - Modeling levels
    - Behavioral, logic, switch, timing, circuit
  - Modeling description (languages)

- Signal models
  - Logic value models
  - Timing value models

- Choices of models determine the complexity and accuracy of simulation
Level of Circuit Modeling (1/2)

- Electronic system level
  - Software+hardware
  - Transaction/cycle-accurate functions
  - C/C++, SystemC, SystemVerilog, etc.

- Register-Transfer-Level (RTL)
  - Define bit and timing (almost) accurate architecture for sign-off
  - VHDL and Verilog

- Logic/cell/gate level
  - Interconnected Boolean gates
    - AND, OR, NOR, NAND, NOT, XOR, Flip-flops, Transmission gates, buses, etc.
  - Suitable for logic design, verification and test
Level of Circuit Modeling (2/2)

- **Switch level**
  - Interconnects of ideal transistor switch
  - Need transistor size, node R and C to determine logic value
    - Zero delay in timing
  - Suitable for full-custom high-performance ASIC

- **Timing level**
  - Use transistors with detailed device models
  - Calculate charge/discharge current with transistor’s voltage-current model and obtain node voltage as a function of time
  - Mainly for post-PR timing verification, e.g., Timemill™

- **Circuit level**
  - Lowest level, ultimate in simulation accuracy
  - Obtain timing by solving the equations relating branch/loop current and node voltage
  - Critical timing analysis for digital circuits
  - Mixed-signal circuit simulation
Logic States for Simulation

- Two states (0, 1) for combinational and sequential circuits with known initial states.
- Three states (0, 1, X) for sequential circuits with unknown initial states
  - X (unknown state) for cases when the logic value cannot be determined
  - X can be either 0 or 1.
  - Sources: uninitialized FF, bus, memory, multi-cycle paths, etc.
Logic Operations with X

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>
Problems with 3-Valued Logic

- May cause information loss
  - Fail determining the logic value even though that value can be easily determined
  - Example:
    The output is evaluated as x even though it should be 1 regardless of the actual value of x
Symbolic Simulation of “x”

- Treat “x” as a signal
  - NOT x = x’
  - x + x’ = 1

- Problem
  - There can be multiple sources of X, e.g., Flip-flops
  - One “x” for each unknown value (x1, x2, …)
  - Impractical for large circuits, e.g., x1 + x2 = ?
High-Impedence State Z

- Floating state: a node w/o a path conducting to either Vdd or Gnd

- Logic state of Z is interpreted dynamically
  - Single floating node
    - Same as its driven value before becoming floating
  - A set of floating nodes get connected
    - Depends on charge sharing, may become uncertain
  - A floating node connected to Vdd/Gnd becomes 1/0
  - When multiple source drive a floating node, the value depends on the strength of the driving logics.

- Most MOS circuits containing dynamic logic require four states (0, 1, x, z) for simulation
An Example of High-Z Bus

\[ o_i = \begin{cases} 
  x_i & \text{if } e_i = 1 \\
  Z & \text{if } e_i = 0 
\end{cases} \]
Delay (Timing) Models

- Delay of a logic element
  - Time between an input change (cause) and the output change (effect), e.g. C->E or D->E
  - Called gate delay, pin-pin delay, or switching delay

- Interconnect delay
  - Time between the generation of a signal transition at a gate output and its arrival at the input of a fanout gate, e.g. A->C, or B->D
  - Or called switching delay
  - Consider R, C (L) effects
Terms for Cell Delay Models

- Zero and unit delay
- Rise (fall) delay
  - Gate delays of different final output states
- Inertia delay
  - Minimum pulse width to cause a transition
  - Used for filtering input/output pulse
  - Input inertia delay: minimum pulse width for input
  - Output inertia delay: minimum pulse width for output
- Min/Max Delay
  - The minimum or maximum bound of a gate delay
- Transition time
  - Time for a signal to transit from 0 to 1 or 1 to 0.
Delay Models Examples

transport delay = 2
(transition-independent)
rise delay = 1
fall delay = 3
min-max delay
(transition independent)
1 ≤ d ≤ 3
input inertia delay = 4
input inertia delay = 2 & transport delay = 2
Common Cell Delay Models

- **Table-based**
  - A pin-pin min/max rise/fall delay of a cell = f(CL, Tr)
  - CL=output load
  - Tr=input transition time

- **Current-source based**
  - A voltage-controlled current source I(Vi, Vo)
  - I: Vdd to Gnd current
  - Vi: input voltage
  - Vo: output voltage
  - More accurate in terms of noise, but more CPU intensive

- **Interconnect delays**
  - Elmore delay
# Modeling Levels and Signals

<table>
<thead>
<tr>
<th>Level</th>
<th>Circuit Description</th>
<th>Signal</th>
<th>Timing Resolution</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESL</td>
<td>SystemC</td>
<td>0,1</td>
<td>transaction</td>
<td>system</td>
</tr>
<tr>
<td>Behavior</td>
<td>HDL</td>
<td>0,1</td>
<td>cycle</td>
<td>architecture</td>
</tr>
<tr>
<td>Logic</td>
<td>gate-level HDL</td>
<td>0, 1, X, Z (with signal strength)</td>
<td>zero, unit, multiple cell delays</td>
<td>logic design and test</td>
</tr>
<tr>
<td>Switch</td>
<td>transistor, RC interconnects</td>
<td>0, 1, X (with signal strength)</td>
<td>zero, possible gross-grain timing</td>
<td>full-custom logic verification</td>
</tr>
<tr>
<td>Timing</td>
<td>same as above (SPICE)</td>
<td>Analog</td>
<td>fine-grain time</td>
<td>timing verification</td>
</tr>
<tr>
<td>Circuit</td>
<td>same as above (SPICE)</td>
<td>Analog</td>
<td>continuous time</td>
<td>timing/analog verification</td>
</tr>
</tbody>
</table>
Types of Logic Simulators

- Compiled-driven simulators
  - The compiled code is generated from an RTL or gate-level description of the circuit
  - Simulation is simply execution of the compiled code

- Event-driven Simulators
  - Simulate only those signals with value changes
  - Only propagate necessary events (value changes)
Compiled Simulation

- A circuit is simulated by executing a compiled code of the circuit.
- Levelization
  - to ensure that a signal is evaluated after all its sources are evaluated

Levelization
- Assign all PI’s level 0
- The level of gate G is
  \[ L_g = 1 + \max(L_1, L_2, \ldots) \]
  where Li’s are G’s input gates

- level 0: \( a, b, c, d \)
- level 1: \( e, f \)
- level 2: \( g, h \)
Flow of Levelization

1. Start
2. Assign level 0 to all PI's
3. Put all PI fanout gates in Q
4. While Q is not empty:
   a. Pop next gate g from Q
   b. If g is ready to levelize:
      i. Append g's fanout gates to Q
      1. l = maximum of g's driving gate levels
      2. Assign l+1 to g
   c. Append g to Q
5. End
Example of Levelization

- The following orders are produced
  - $G_1 \Rightarrow G_2 \Rightarrow G_3 \Rightarrow G_4$
  - $G_1 \Rightarrow G_3 \Rightarrow G_2 \Rightarrow G_4$

<table>
<thead>
<tr>
<th>step</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$G_1$</th>
<th>$G_2$</th>
<th>$G_3$</th>
<th>$G_4$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$&lt;G_2, G_1&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$&lt;G_1, G_2&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>$&lt;G_2, G_3&gt;$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>$&lt;G_3, G_4&gt;$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>$&lt;G_4&gt;$</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Compiled Simulation – cont’d

- Code generation & execution

while (1) {
    read_in(a, b, c, d);
    e = AND(a, b);
    f = NOR(b, c);
    g = OR(e, f);
    h = NAND(d, f);
    print_out(g, h);
}

- Very effective for 2-state (0,1) simulation
  - Can be compiled directly into machine codes

- Mainly for functional verification, where timing is irrelevant
Code generation in C

```c
#include <stdlib.h>
int main(){
    unsigned int a=0xF; //1111
    unsigned int b=0xA; //1010
    unsigned int c=0x8; //1000
    unsigned int d=0x7; //0111
    unsigned int e, f, g, h;
    e = a&b;
    f = ~(b|c);
    g = e|f;
    h = ~(d&f);
    printf("g,h=%X,%X", g, h);
}
```
Problems with Compiled Simulation

- Zero-delay model
  - Timing problems, e.g., glitches and races, cannot be modeled

- Simulation time could be long
  - Proportional to $\Omega(\text{input vectors} \times \text{number of gates})$
  - Entire circuit is evaluated even though typically only 1-10% of signals change at any time
  - Note RTL compiled simulation is different and fast, since branching can be used.
Event-Driven Simulation

- An event is a change in value of a signal line
- An event-driven simulator evaluates a gate (element) only if one or more events occur at its inputs
- Only does the necessary amount of work
- Follows the path of signal flow

Diagram of a simple circuit with logic gates indicating signal flow and changes.
Zero-Delay Event-Driven Simulation

1. Read in initial state information
2. More input vector?
   - Yes: Read in new i/p vector, and put the fanout gates of the PIs with events in event queue Q.
   - No: Stop
3. Is Q empty?
   - Yes: Evaluate next g in Q. If g changes state, put its fanout gates in event queue Q.
   - No: More input vector?
The most straightforward and easy way to implement
- For binary logic, $2^n$ entries for $n$-input logic element
- May use the input value as table index
- Table size increases exponentially with the number of inputs

Could be inefficient for multi-valued logic
- A $k$-symbol logic system requires a table of $2^{mn}$ entries for an $n$-input logic element
  - $m = \log_2 k$
  - Table indexed by $mn$-bit words
Gate Evaluation – Input Scanning

- Assume that only dealing with AND, OR, NAND, and NOR primitive gates.
- These gates can be characterized by controlling value $c$ and inversion $i$.
  - The value of an input is said to be controlling if it determines the gate output value regardless of the values of other inputs.

<table>
<thead>
<tr>
<th>Gate</th>
<th>$c$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NAND</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Input Scanning – cont’d

<table>
<thead>
<tr>
<th>I/P of a 3-input primitive gate</th>
<th>O/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>c  x  x</td>
<td>c\oplus i</td>
</tr>
<tr>
<td>x  c  x</td>
<td>c\oplus i</td>
</tr>
<tr>
<td>x  x  c</td>
<td>c\oplus i</td>
</tr>
<tr>
<td>c' c' c'</td>
<td>c'\oplus i</td>
</tr>
</tbody>
</table>

Evaluate(G,c,i){
    u_values = false;
    for every input value v of G{
        if (v == c) return c\oplus i;
        if (v == x) u_values = true;
    }
    if (u_values) return x;
    return c’\oplus i;
}
Gate Evaluation – Input Counting

- To evaluate the output, it’s sufficient to know
  - Whether any input equals c
  - If not, whether any input equals x
- Simply maintain c_count & x_count
- Example: AND gate
  - 0 => 1 at one input: c_count--
  - 0 => x at one input: c_count--, x_count++

```
Evaluate(G,c,i){
    if (c_count > 0) return c⊕i;
    if (x_count > 0) return x;
    return c’⊕i;
}
```
Event-Driven Simulation with Delays

- While (event list is not empty) {
  t = next time in list;
  process entries for time t;
}

- The key is to construct a queue entry for every time point
Time wheel

- Max units is the largest delay experienced by any event
  - All gates + interconnects
  - A total of max+1 slots
Flow of Simulation with Delays

Algorithm 1 (two-pass)

Pass 1
retrieves the entries from event list & determine the activated gates.

Pass 2
evaluates the activated gates and schedules their computed values.
Simulation with Delays

Algorithm 1

\( \text{Activated} = \emptyset \)

For every event \((g, v_g^+)\) at list of time \(t\) { // from \(L_E\)

if \((v_g^+ \neq v_g)\) // \(v_g\) is the original value at signal \(g\) {

\(v_g = v_g^+;\)

for every \(j\) on fanout list of \(g\) {

update input value of \(j\);

add \(j\) to \(L_A\) if \(j\) is not a member of \(L_A\);

}\ /* for */
}

/* if */
}
/* for */

For every \(g \in L_A\) {

\(v_g^+ = \text{evaluate } (g)\);

schedule \((g, v_g^+)\) for time \(t+\text{delay}(g)\);

}
Two-Pass Algorithm

**Example**

![Diagram of a logic circuit with gates G1, G2, G3, and G4, and inputs A, B, and C, and outputs H, J, and K.]

**Gate delay**
- G1: 8ns
- G2: 8ns
- G3: 4ns
- G4: 6ns

### Table 3.5: Two-pass event-driven simulation

<table>
<thead>
<tr>
<th>Time</th>
<th>$L_E$</th>
<th>$L_A$</th>
<th>Scheduled events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{(A,1)}</td>
<td>{G2}</td>
<td>{(H,1,8)}</td>
</tr>
<tr>
<td>2</td>
<td>{(C,0)}</td>
<td>{G1}</td>
<td>{(E,1,10)}</td>
</tr>
<tr>
<td>4</td>
<td>{(B,0)}</td>
<td>{G1}</td>
<td>{(E,0,12)}</td>
</tr>
<tr>
<td>8</td>
<td>{(A,0),(H,1)}</td>
<td>{G2,G4}</td>
<td>{(H,0,16),(K,0,14)}</td>
</tr>
<tr>
<td>10</td>
<td>{(E,1)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>{(E,0)}</td>
<td>{G2,G3}</td>
<td>{(H,0,20),(J,1,16)}</td>
</tr>
<tr>
<td>14</td>
<td>{(K,0)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>{(H,0),(J,1)}</td>
<td>{G4}</td>
<td>{(K,0,22)}</td>
</tr>
<tr>
<td>20</td>
<td>{(H,0)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>{(K,0)}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (Cont.)
Example of Algorithm 1 to schedule a Null Event

- Time 0 : event (a, 1)
  evaluate $z=1 \Rightarrow (z, 1)$ scheduled for time 8
- Time 2 : event (b, 0)
  evaluate $z=0 \Rightarrow (z, 0)$ scheduled for time 10
- Time 4 : event (a, 0)
  evaluate $z=0 \Rightarrow (z, 0)$ scheduled for time 12

The last scheduled event (at $t=12$) is not a real event!!
An Improved Algorithm

Change Pass 2 to:

For every $j \in \textbf{Activated}$ {
    $v_j' = \text{evaluate}(j)$;
    if ($v_j' \neq lsv(j)$) ($lsv$: last saved value)
    {
        schedule ($j$, $v_j'$) for time $t + d(j)$;
        $lsv(j) = v_j'$;
    }
}
Two Pass V.S. One Pass Algorithm

- Two-pass strategy performs the evaluations only after all the concurrent events have been retrieved
  - to avoid repeated evaluations of gates having multiple input changes.
- Experience shows, however, that most gates are evaluated as a result of only one input change.
- One-pass strategy:
  - Evaluates a gate as soon as it is activated
  - Avoids the overhead of building the Activated set
One Pass Algorithm

For every event \((g, v_g^+)\) pending at current time \(t\) { 
\[v_g = v_g^+;\]
for every \(j\) on the fanout list of \(g\) { 
update input values of \(j\);
\[v_j^+ = \text{evaluate} \ (j);\]
if \((v_j^+ \neq v_j)\) { 
schedule \((j, v_j^+)\) for time \(t+d(j);\)
\[v_j = v_j^+;\]
}
}
}
An Example of Hazards

Hazards

- Unwanted transient pulses or glitches
Type of Hazards

- Static or dynamic
  - A static hazard refers to the transient pulse on a signal line whose static value does not change
  - A dynamic hazard refers to the transient pulse during a 0-to-1 or 1-to-0 transition
- 1 or 0

Static 1-hazard | Static 0-hazard | Dynamic 1-hazard | Dynamic 0-hazard
Static Hazard Detection

- Extra encoding can be used to detect hazards during logic simulation.
  - Note that hazards only occur during signal transition
  - Two consecutive vectors are considered simultaneously
- The following is the 6-valued encoding for a pair of vectors.
  - For example, 0->1 transition (R) is encoded as 0X1.

<table>
<thead>
<tr>
<th>Value</th>
<th>Sequence(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>Static 0</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>Static 1</td>
</tr>
<tr>
<td>0/1, R</td>
<td>{001,011} = 0x1</td>
<td>Rise (0 to 1) transition</td>
</tr>
<tr>
<td>1/0, F</td>
<td>{110,100} = 1x0</td>
<td>Fall (1 to 0) transition</td>
</tr>
<tr>
<td>0*</td>
<td>{000,010} = 0x0</td>
<td>Static 0-hazard</td>
</tr>
<tr>
<td>1*</td>
<td>{111,101} = 1x1</td>
<td>Static 1-hazard</td>
</tr>
</tbody>
</table>
# 6-Valued Logic for Static Hazard Analysis

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
<th>R</th>
<th>F</th>
<th>0*</th>
<th>1*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>R</td>
<td>F</td>
<td>0*</td>
<td>1*</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>R</td>
<td>R</td>
<td>0*</td>
<td>0*</td>
<td>R</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>F</td>
<td>0*</td>
<td>F</td>
<td>0*</td>
<td>F</td>
</tr>
<tr>
<td>0*</td>
<td>0</td>
<td>0*</td>
<td>0*</td>
<td>0*</td>
<td>0*</td>
<td>0*</td>
</tr>
<tr>
<td>1*</td>
<td>0</td>
<td>1*</td>
<td>R</td>
<td>F</td>
<td>0*</td>
<td>1*</td>
</tr>
</tbody>
</table>
Oscillation

- Oscillating circuits will result in repeated scheduling & processing of the same sequence of events

- Oscillation control takes appropriate action upon detection of oscillation
Local Oscillation Control

- identify conditions that cause oscillations in specific sub-circuits, e.g., latches, flip-flops
  - For an oscillating latch, the appropriate corrective action is to set $y = y' = x$ (unknown)

- Oscillation control via modeling
  - Example: when $y=y'=0$ (oscillation condition, also implying $S=R=1$), $G = x$ causes $y = y' = x$ and stops oscillation
Global Oscillation Control

- Detection of global oscillation is computationally infeasible
  - Requires detecting cyclic sequences of values for any signal in the circuit
- A typical procedure is to count the number of events occurring after any primary input change
  - Oscillation is “assumed” if the number exceeds the specified limit
Simulation Engines

- **Motivation**
  - Logic simulation is time consuming.

- **Simulation engines are special-purpose hardware for speeding up logic simulation.**
  - Usually attached to a general-purpose host computer through, for example, VME/PCI bus.
  - FPGA-based logic emulation

- **Use parallel and/or distributed processing architectures.**