Brief Introduction to Logic Simulation

Kinds 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
  – During a single simulation of the circuit, typically only a small percentage of the signal lines change values
  – Simulate only those signals with value changes
Logic States for Simulation

• Combinational and synchronous circuits with a known initial state
  – Two logic states (0, 1)
    • Good to compute logic behavior
    • Not enough for detecting hazards

• Sequential circuits with an unknown initial state
  – Three states (0, 1 & u)
    • u (unknown state) represents
      – Initial state of flip-flops & RAMs
      – Interpreted as either 0 or 1

Truth Table for 3-Valued Logic

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

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>u</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>1</td>
<td>u</td>
</tr>
</tbody>
</table>
**Information Loss of 3-Valued Logic**

Related to unknown:

\[
\begin{align*}
(a b c) &= (1u1) \quad \text{should produce } f = 1 \quad \text{instead of } f = u \\
\Rightarrow \quad \text{The use of } u \text{ may cause information loss}
\end{align*}
\]

**Compiled Simulation**

- Given circuit C:
  1. Levelize circuit
  2. Generate code

- Level 0: a, b, c, d
- Level 1: e, f
- Level 2: g
- Level 3: i
- Level 4: j
• Code:

```c
while(1) {
    Read_in ( a, b, c, d );
    e = NAND ( a, b );
    f = INV ( c );
    g = NOR ( b, f );
    i = AND ( e, i );
    j = NAND ( i, d );
    Print ( e, i, j );
}
```

• Levelizing circuit:
  - Assign all PI lines level 0
  - The level of a gate g is:
    \[ L_g = 1 + \max ( L_{i1}, L_{i2}, \ldots L_{in} ) \]
    \( i_1, i_2 \ldots i_n \) are inputs of gate g.

• Elements are simulated in ascending order of logic level
• Time required to simulate a vector = \( t \times N \)
  - \( t \): time required to simulate an element
  - \( N \): number of elements

**Problems with compiled simulators**
• 0-delay model: timing cannot be properly modeled \( \Rightarrow \) no hazard or signal propagation is predicted
• Simulation time = \( t \times N \) could be slow
Event-Driven Simulation

- For each new input vector, the ratio of lines which change values to the total number of lines in the circuit is called the activity A.
  - Typically A = 2 - 10 %
- An event is a change in value of a signal line
- The output of a gate $i$ will only change value when one or more of its inputs change value.
  - Element $i$ need only be simulated when an event occurs at one of its inputs

When an event does occur, then every element to which this line fans out to, is called potentially active.

Simulating these potentially active lines alone can result in significant savings in computation time.

Example:

0000 $\rightarrow$ 0001
Only simulated NOR & NAND gates.
**Algorithm - a simplified version**

- 0-delay assumption

```
• Read in initial state information

• More input Vector?
  - Yes
    - Is event queue empty?
      - No
        - Evaluate element $i$, $i \in Q_1$. If element $i$ changes state, put Fanout($i$) in $Q_1$
      - Read in (new) input vector
        - For each line $i$ such that $V_{new}(i) \neq V_{old}(i)$
          - Put $(i, V_{new}(i))$ on event queue $Q_1$
        - Stop
  - Yes
```
**Delay Models**

- **Zero delay:** Output changes instantly in response to an input
- **Unit delay:** All gates have one unit delay
- **Transport delay:**
  - transition independent
  - transition dependent: rise & fall delays are specified separately
- **Ambiguous delay or Minmax delay**

**Inertial delay:** All circuits require energy to switch states. The minimum duration for which an input change should persist in order for the device to switch states is called inertial delay

- **Input inertial delay** $d_I$
  - An input pulse whose duration is less than $d_I$ is filtered (or suppressed) by the gate.
- **Output inertial delay** $d_O$
  - The gate output cannot generate a pulse whose duration is less than $d_O$
Event-Driven Simulation with Delay

• 0-delay model: Only verifies functionality

0 \rightarrow 1

\begin{align*}
0 \ (t = 0) & \quad 1 \ (t = 1) \\
1 \ (t = 1) & \quad 0 \ (t = 2)
\end{align*}

• Unit - delay model:

\begin{align*}
1 \ (t = 1) & \quad 0 \ (t = 2) \\
1 \ (t = 1) & \quad 1 \ (t = 3)
\end{align*}

Need a "time-flow" mechanism.
**Time Flow Mechanism & Event Scheduling**

- When an element $i$ is simulated and it is determined that its output signal $j$ has changed state, i.e. $V_{\text{new}}(j) \neq V_{\text{old}}(j)$, then signal $j$ must be scheduled to change to $V_{\text{new}}(j)$ at time $t + \Delta i$
  where $t$: the current simulation time
  $\Delta i$: the delay of element $i$
  The event $(j, V_{\text{new}}(j), t + \Delta i)$ is scheduled to occur in the future.
- Need an event queue for each time of simulation

**Time wheel -**

- An array of headers
- Events scheduled for time $t$ appear in array $[t \mod M]$.
- $M$ must be larger than the longest delay of any element.
Event Driven Simulation

- While (event list is not empty)
  
  \{
  \begin{align*}
  t &= \text{next time in list;} \\
  \text{process entries for time } t \\
  \end{align*}
  \}

- Process_entries: Algorithm I
  - Two passes
    - **Pass 1**: retrieves the entries from event list & determine the activated gates.
    - **Pass 2**: evaluates the activated gates and schedules their computed values.

Algorithm I

 Activated = φ

For every entry \((i, v'_i)\) at list time \(t\) \{ 
  \begin{align*}
  &\text{if } (v'_i \neq v(i)) \{ \\
  &\quad \text{update } v(i) = v'_i; \\
  &\quad \text{for every } j \text{ on fanout list of } i \{ \\
  &\quad \quad \text{update input value of } j; \\
  &\quad \quad \text{add } j \text{ to } \text{Activated}; \\
  &\quad \} \text{ /* for */} \\
  &\} \text{ /* if */} \\
  &\text{ /* for */}
  \end{align*}
\}

For every \(j \in \text{Activated}\) 
  \begin{align*}
  &v'_j = \text{evaluate } (j); \\
  &\text{schedule } (j, v'_j) \text{ for time } t+d(j);
  \end{align*}
\}
Example:

\[
\begin{array}{ccc}
\text{a} & \text{8} & \text{z} \\
\end{array}
\]

- 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 \( j \in \text{Activated} \) {

\[
\begin{align*}
\nu_j' &= \text{evaluate (j)}; \\
\text{if} \ (\nu_j' \neq lsv(j)) \\
\{ \\
&\text{schedule (j, } \nu_j' \text{) for time } t+d(j); \\
lsv(j) &= \nu_j'; \\
\}
\}
\end{align*}
\]
• 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 \((i, v_i')\) pending at current time \(t\) {
\[
v(i) = v_i';
\]
for every \(j\) on the fanout list of \(i\) {
  update input values of \(j\);
  \(v_j' = \text{evaluate} (j)\);
  if \((v_j' \neq lsv(j))\) {
    schedule \((j, v_j')\) for time \(t+d(j)\);
    \(lsv(j) = v_j'\);
  }
}

Simulation Engines

• Logic simulation is time-consuming

• Simulation engines (Simulator hardware accelerators) are special-purpose hardware for speeding up logic simulation.
  – Usually attached to a general-purpose host computer.
  – For example, the hardware accelerator can be implemented in a board & communicated with a workstation through VME/PCI bus.

• Use parallel and/or distributed processing architectures

Logic Emulation

• Implementing the netlist in re-programmable hardware.

• Re-programmable hardware (Emulation hardware):
  – Field- programmable gate arrays (FPGAs) are hardwired together on large PC boards- called emulation-module boards
  – All emulation-module boards are connected through backplane connectors into a large array of FPGAs.
The emulation hardware can be operated at 100 kHz to several MHz.
- Simulating 100k to several mega vectors per second
- 10,000 to 1,000,000 times faster than software simulators

Synthesis process:

1. ASIC netlist
2. Compilation
   - (design translation, partitioning, placement & routing)
3. bit streams for reprogramming FPGAs
4. Configuration
   - (reprogram the FPGAs)

Emulation hardware

FPGAs

Backplane connector

Emulation modules