Simulation

EE5375 – ADD II
Prof. MacDonald
Logic Simulation

● Simulation is primarily used to verify design:
  – function and / or
  – timing

● Simulation also used to verify:
  – operation regardless of power-up state (initial X’s)
  – operation across processing variation (delay min-max)
  – the absence of race conditions, hang-ups, etc.

● Also used for:
  – evaluation of design alternatives
  – evaluation of ECO’s
  – documentation – demonstrate operation
Sim Process

control stimuli → sim program → results: text or waves

library → sim program

internal model

automated checking
Prototypes

- Originally used instead of simulation
  - prototype were built of discrete components
  - impossible today based on complexity

- Emulation popular today
  - FPGA systems used to model design
  - runs at 1 to 2 MHz – 100 times slower than actual
  - logic simulation – run ~10,000 slower
  - logic simulation w timing – run ~100,000 slower
  - transistor simulation – runs ~1,000,000,000 slower
  - difficult to generate input stimuli
  - difficult to debug and “see” internals
Sim vs. Emulation

- Can change delays (temp/process/voltage)
- Can detect contention (look for x’s)
- Self checking easier
- Can start from any arbitrary state (well maybe)
- Can see any signal – internal visibility
- Can generate input stimuli
  - precise control of asynch inputs (IRQ)
- Can arbitrarily force signals behave differently
- BUT MUCH SLOWER
Verification vs. Test

- Verification used to detect design errors
- Test generation for detecting physical defects
- Defects are fairly well understood
  - can be modeled and enumerated
  - fault coverage can be calculated
  - simple model like stuck-at works reasonably well
- Design errors are not well defined or enumerated
  - determining quality of verification is nightmare
  - how much time do you have? keep adding testcases
  - some use bug detection rate to measure quality
Problems with Simulation

- How to generate tests
  - Why we pay verification engineers?
  - Verification is harder than design
    - good verification engineers worth weight in gold
    - Example,
      - designer adds a new mode bit in 10 minutes
      - verification now has twice the possible state space
  - Bug rate
    - blocks, modes, and features provide categories
      - can still miss a specific combination of modes and stimuli
      - relies on engineers intuition
    - “Done at tape-out”
Problems with Simulation

- Are the results correct?
  - Thousands of testcases with millions of signals
  - Eye-ball method
    - necessary for the first round of simulations
    - not reliable as testcase list grows and
    - design changes lead to re-running all again and again
    - not good a year later after personnel changes
  - Monitors and checkers
    - have a testbench that checks for specific conditions
    - need tests with teeth
    - false positives
    - false negatives
Formal Verification

- **Model Checking**
  - make rules and use formal methods to prove correct
  - rigorous
  - number of rules required is overwhelming
  - state explosion – can’t handle big designs

- **Boolean Equivalence**
  - Referred to as FV also (confusing)
  - Very commonly used in ASIC design methodology
  - Compares two designs
    - required now for first synthesis and final layout netlist
Simulator Definitions

- Compiler-driven Simulator
  - model is based on code
- Table-driven Simulator
  - model based on data structure
- Activity and Event-driven Simulation
- Events
- Evaluation
- Level of simulation
Unknown Logic Value

- Logic level may be unknown
  - sequential elements at power up
  - contention
  - timing violation

- Textbook uses symbol u, normally x is used.

- These tables are pessimistic

<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>0</td>
<td>1</td>
</tr>
<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>

<table>
<thead>
<tr>
<th>not</th>
<th>0</th>
<th>1</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Pessimism in U evaluation
Pessimism in synch reset flops
Unknowns and feedback

- Logic simulations often require no feedback
- All feedback is buried inside of primitives
- Feedback also kills static timing analysis
Compiled Simulation

- No good sense of timing
  - cycle based.
  - sequential circuits only
- Evaluate circuit every cycles and store state
- Very fast and can simulate $W$ vectors for a $W$-wide memory
- Good for evaluation of test vectors on comb. circuits.
Compiled Simulation

\[ \text{lda } b \]
\[ \text{inv} \]
\[ \text{or } q \]
\[ \text{and } a \]
\[ \text{sta } g \]
\[ \text{stg } q \]

\[ a \]
\[ b \rightarrow e \rightarrow f \rightarrow g \rightarrow D \rightarrow Q \]
\[ q \rightarrow e \rightarrow f \rightarrow g \rightarrow D \rightarrow Q' \]
Event Driven Simulation

- Evaluations only occur for activated gates.
- Inactive circuits require no computation.
- Simulation time is advanced to next event – thus skipping periods of inactivity.
- Can easily introduce control events
  - display signal values
  - check values and report mismatches
  - end simulation
Event Driven Simulation

- Event list starts with input stimulus
- Each event activates one or more gates
- Activated gates are evaluated
- Output changes result in scheduling of new events in the event list – merging these internally generated events with original.
- Simulation time continues until no more events or stop event is encountered.
Delay Models

- Every gate introduces delay affected by:
  - Supply voltage
  - Temperature
  - Rise vs. fall delay
  - Input to output path
  - Input transition time
  - Output capacitance

- Transport Delay
- Inertial Delay
Delay Modes

- **Unit Delay Model**
  - all delays reduced to 1
  - gives indication of critical path
  - avoids hold time violations

- **Ambiguity Delay Model**
  - uses ambiguity interval – delay is between 3 and 6

- **Nominal Delay Model**
  - spice generated delays are used

- **Zero Delay Model** – possible race condition
Delay Modes in Verilog

- Zero Delay Mode – possible race condition
- Unit Delay Mode
  - all delays reduced to 1
  - gives indication of critical path
  - avoids hold time violations
- Distributed Delay Mode
- Path Delay Mode
Wire Delay Models and back annotation

- In deep submicron technology, device delay is decreasing while wire delay is increasing
- Prior to place / route, use an estimated model
  - based on silicon area
  - uses statistical approach
- After place and route, back-annotate delay
  - for timing-based simulation
  - use in static timing analysis
Element Evaluation

- Determining output values based on inputs
- Input storage
  - parallel with signals – memory optimized
  - for each gate redundantly
    - faster but more memory used
    - good for fault modeling

- Should include generation of timing
Element Eval – Truth tables

- number of inputs and states = n
- number of logic values = k
- number of bits to represent up to k values = q
  where k <= 2^q
- table must have 2^(qn) entries
- fast evaluation
- limited by number of states/inputs
- Binary flag – indicates if any input is non-binary
  - if not set, use binary truth table
  - otherwise use a slower function to evaluate
  - good trade-off
Element Eval – Zoom tables

- Combine truth tables of all gates in library
- Concatenate gate type and gate inputs for index
- Main problem – size
- Each type must use $S$ entries

![Diagram showing truth tables for types 0, 1, and $t-1$.]
Element Eval – input counting

- Each gate is defined by C and I
  - C is controlling value which always determines output
  - I is inversion and determines inversion parity
- XOR?
- Gates with inverted inputs

<table>
<thead>
<tr>
<th></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>
<tr>
<td>XOR</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
function evaluate (G, c, i)
    u_values = 0;
    for every input value v of G
        if v=c then return c xor i
        if v=u then u_value = 1
    if u_values then return u
    else return c’ xor i
Element Eval – input counters

If counters are used for number of inputs that are controlling and unknown, then evaluate algorithm is simplified and speed-up

function evaluate (G, c, i)
    if c_count > 0 return c xor l
    else if u_count > 0 return u
    else c’ xor l
Hazards

- Hazards are a subset of glitches
  - not necessarily bad on data signals
    - assuming timing is met – glitch is ignored
    - does result in gratuitous power consumption
  - very bad on clocks and asynch resets and sets
  - Static hazards
  - Dynamic hazards

- Multi-valued logics to detect and simulate hazards
  - 6 level logic for static
  - 8 level logic for dynamic
# Hazards – 6 valued logic

<table>
<thead>
<tr>
<th>Value</th>
<th>Sequence</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 transition</td>
</tr>
<tr>
<td>1/0, F</td>
<td>{100, 110} = 1X0</td>
<td>Fall transition</td>
</tr>
<tr>
<td>0*</td>
<td>{000,010} = 0X0</td>
<td>Static 0 hazard</td>
</tr>
<tr>
<td>1*</td>
<td>{101,111} = 1X1</td>
<td>Static 1 Hazard</td>
</tr>
</tbody>
</table>
**Hazards – 6 valued logic - AND**

<table>
<thead>
<tr>
<th></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>
# Hazards – 8 valued logic

<table>
<thead>
<tr>
<th>Value</th>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>Static 0</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>Static 1</td>
</tr>
<tr>
<td>0/1, R</td>
<td>0XX1 excluding</td>
<td>Rise</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>1/0, F</td>
<td>1XX0 excluding</td>
<td>Fall</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0XX0</td>
<td>Static 0 hazard</td>
</tr>
<tr>
<td>1*</td>
<td>1XX1</td>
<td>Static 1 hazard</td>
</tr>
<tr>
<td>R*</td>
<td>0XX1</td>
<td>Dynamic hazard</td>
</tr>
<tr>
<td>F*</td>
<td>1XX0</td>
<td></td>
</tr>
</tbody>
</table>
Event List

Tp

Tq

Tr

done
General algorithm for event-driven sim

While (event list not empty)
    begin
        t = next time in list (follow pointer)
        process all entries
    end

Processing all entries entails evaluating all gates with an input that has changed and scheduling events for the output of the gate later.
Two pass algorithm

- Two pass Algorithm
  - For each time, create list of activated gates by looking at the fanout of changing nets
  - Evaluate all activated gates

- Avoids evaluating gates 2 or more times
Only True Event Algorithm

- Algorithm
  - For each time, create list of activated gates by looking at the fanout of changing nets
  - Evaluate all activated gates
  - Only add new events if an output actually changes

- Avoids events that don’t really occur
- Requires storing previous state of all signals