LOGIC SIMULATION

• Suppose we have implemented a combinational or sequential circuit

• We would like to know the logical values of the outputs of the circuit for a given (sequence of) input(s).

• Sometimes, while performing logic simulation, we would like (coarse) timing information of the circuit as well

• What is the usefulness of logic simulation?
  – Logic simulation is often used as a means to determine if the implemented circuit is correct (with respect to a high level specification).
  – By logically simulating a series of test vectors, and testing that the outputs are correct, we can verify circuit functionality
  – This is also referred to as functional verification.
  – It can help answer the question “Is what I implemented what I had functionally specified?” Contrast this with ATPG, which answers the question “Is the manufactured IC consistent with what I implemented?”
The confidence with which we can answer the above question after such functional verification depends on how comprehensive our set of test vectors is. Obviously, we can’t apply all possible input test vectors in general (even for combinational circuits), so there is always a possibility that functional errors are still present in the design.

Another technique to perform functional verification is *formal verification*.

* Model-checking is a powerful technique for this, and is covered in ECEN 5139.

* In such techniques, the designer specifies a temporal property of the system, using a *temporal logic*.

* The model checking tool implicitly verifies that this property holds for *all possible execution traces* of the design. If the property fails, the model checker reports a witness trace.

* The factors which slow the acceptance of formal verification techniques are
  
  - Industrial designers often feel that temporal properties are not easy to write
  
  - Also, it is not clear how many properties are sufficient to “fully” test the design (current research is beginning to address this).
• Formal verification can effectively handle relatively small designs realistically (abstraction-refinement based techniques address this fact).

— As a result, logic simulation based functional verification is still very prevalent in industry.

• There are several “levels” at which we may perform logic simulation. For each such style of logic simulation, timing information may be associated with each circuit element.

— Behavioral level:
  * The circuit is typically described in a HDL
  * The resulting description is much like a C-program, with specialized syntax to handle clocks, concurrency, signal drive strengths, etc.

— Gate level:
  * The circuit is described as a gate level netlist.
  * Gates may have timing information associated with them

— Switch level:
  * The circuit is described as a transistor-level netlist
  * The simulation outputs are logical signals
* This requires special handling of gate strengths, node capacitances etc.

- We will focus on gate-level logic simulation.

- The core gate-level simulation techniques are applicable to behavioral or switch level simulation, with modifications to address the specialized features of designs described at these levels.
LOGIC SIMULATION - OVERVIEW

- There are several algorithms for logic simulation:
  - Event-driven
    * Assume we are given the circuit description, input vector and logical description of all gates.
    * Simulation events stored in a queue of temporally sorted pending events.
    * Say the first event causes the output of a gate $G$ to change at time $t_G$.
    * Schedule an event at gate $G$, at time $t_G$.
    * Process the next event, proceeding until the event queue becomes empty.
  - Compiled code
    * In a setup phase, we compile the circuit description into a series of machine language instructions.
    * For example, if our circuit is a AND gate, then the compiled machine level instruction will be a logical AND. Assuming 32 bit operands, we will be able to simulate 32 input vectors at once.
* Significant speedup over event-driven, but compilation overhead can be large

— Hardware emulation
  * Build circuit in hardware.
  * Earlier approaches took long (several months) to develop hardware prototype. Several companies offered such products
  * Newer approaches involve FPGA-based prototype.
  * Significant speedup over compiled code
EVENT DRIVEN SIMULATION

- Consider the example below:

- Assume that \{A, B, C, D, E, F, G\} = <1000101> at \(t = 100\), CLK=0 at \(t = 125\), CLK=1 at \(t = 175\) and \(C = 1\) at \(t = 140\).

- Assume that the logic states modeled are 0, 1, X.

- Let the logic functionality of the AND gate be as shown below:

<table>
<thead>
<tr>
<th></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>
• Suppose the AND and OR gate delay is 10 (we could have different rising/falling delays, min-max delays, or input-dependent delays as well)

• The main simulation structure is a **timing queue** or **timing wheel**. This queue lists pending temporally sorted **events**.

• The algorithm iterates on the followign steps
  – Take the first event on the queue
  – Find its consequence (signals that change in the future). This is done by a table-lookup.
  – Insert such signals in the queue

• We proceed until the queue becomes empty of pending events, or if a user-specified simulation time limit is reached.

• This table evolves as below:
time | event
--- | ---
125 | CLK=0
140 | C=1
175 | CLK=1

Shaded event causes B=1, G=0 at 135.

Shaded event causes D=1 at 145.

Shaded event causes no new events

Shaded event causes F=1 at 155

Events shaded as represent the current event
Past events are removed (they are shown here to make the figure easier to follow)
• From an implementation perspective, we need a fast way to add/remove events to our queue.
  
  — If we use a linked list, insertion takes $O(N)$ time (assuming $N$ elements in the list already). On average this would take $N/2$ time, but in practice the list is cluttered since most gates have about the same delay. Hence insertion time is close to worst-case.
  
  — If we use a heap (priority tree), then insert is done in $O(\log N)$ time, and deletion in constant time. A (min) heap is a binary tree with the property that the value stored at any node is less than or equal to the value of its children.
  
  — In practice we usually use an event wheel. This is an array whose indices are time units. Size of the ring must be guessed (larger if delays have fine granularity).
Slots are indexed with the time they correspond to

- In this case, events can be added in constant time. When searching for the next event, we simply skip empty time slots until we find a time slot with an event attached to it.

- There may be several events at a given time slot. So they could be represented as a linked list.
EVENT Driven Simulation - VARIANTS

- One variant is **min-max** simulation. In this case, gates are assigned a minimum and a maximum delay.

- If the minimum falling delay is 3, and the maximum falling delay is 7, then all we can say for times between 3 and 7 is that the output is falling. Thus *regions of ambiguity* result.

- If gates are in series, these regions of ambiguity can rapidly grow in size.

```
0   3-7   6-14   9-21
```

```
\[ \text{min delay} = 3, \text{max delay} = 7 \]
```

- The advantages of min-max simulation is that it can help model variations in delay due to processing, temperature etc.

- The disadvantages include more events (twice as many) to process and somewhat more complex implementation and display of results.
• Another disadvantage is that ambiguity is often amplified as the above example shows.

• In reality, the delays of the inverters in the above figure are usually correlated. If one inverter has a delay close to the max delay (due to temperature or process factors), the other inverters also have a similar delay.

• Therefore min-max simulation is more relevant for board-level simulations, where speed variation of the different parts is independent. On an IC, gate delays are usually correlated (especially for gates that are physically near each other)
EVENT DRIVEN SIMULATION - SWITCHES AND LOGIC STRENGTH

- In many circuits, the output “strength” needs to be modeled.

- For example, it is important to know if a signal is resistively driven to some logic value or capacitively held at that value.

- This is typically required for switch-level circuits, or for circuits implemented with tristate-able drivers.

- In the figure above, signal B is high-impedance (capacitively held at its value) when TS=1. When TS=0, it is resistively driven to the value of A.
• In the transparent latch, it is important to simulate the relative strength of the two inverters.

• In the second example, the relative strengths of X and Y are important in determining Z (in case C1 and C2 are on simultaneously). Actually, if C1 and C2 are mutually exclusive, this circuit is a Mux.

• In the third example, the internal capacitances of the nodes Y and Z are important in determining the output Z (which is computed by applying charge conservation).

• In all the examples above, we need to model signal strengths in addition to their logical values.

• To simulate these features, the simulator must have a richer set of logic values (perhaps 0, 1, X, R1, R0, C1, C0 or more values).

• Also, the simulator must handle charge sharing.

• Terman in 1983 came up with such a simulator. In this event driven simulator, 3 values were used. X values are scheduled as soon as possible. Logic 0 or 1 values are scheduled later.
SPEEDING UP LOGIC SIMULATION

• Typical speed for logic simulator is 1000 events/sec/MIP.

• So a 1 MIP processor simulates 1000 events/sec. In other words, it takes about 1000 instructions to process an event.

• For a circuit with 10 million gates with 10% activity per clock cycle (i.e. the number of gate outputs with events on them) we would require 1000 seconds to simulate the circuit per clock cycle per MIP.

• Activity would be higher for DSP ICs, and is typically about 10% for processor-like circuits.

• For an IC with 10 million gates (10% activity), whose clock speed is 1GHz, simulated on a 1000 MIPS processor, we would require 1 second to simulate a clock period. This is 9 orders of magnitude slower than the speed of the IC. Ouch.

• So we would like to speed up simulation. The candidate techniques are:
  – Software techniques (levelized code simulation, compiled code simulation)
- Hardware techniques (hardware simulators, hardware emulators, hardware models)

- Let us look into each of these techniques next
LEVELIZED CODE

- Consider the circuit fragment below. Assume all gates have a delay of 10.

- If we perform event driven simulation on this circuit, we can potentially have more than one event on a node during the simulation.

- Let $abc$ change from $<000>$ to $<110>$. In that case, we have a rising event on $d$ at 10. When we process this, we get a rising event on $e$ at 20, and a rising event on $f$ at 20. Suppose we process the event on $f$ first. Then we process the event on $e$ which causes a falling event on $f$ at 30.

- In other words, we processed two events on $f$.

- Can we avoid this?

- Yes. We avoid this by **levelizing** the circuit beforehand.
• The level of a node is 0 if it is a PI node. Otherwise, it is one greater than the maximum level of all its fanin nodes. In our example, $d$ has a level 1, $e$ has a level 2 and $f$ has a level 3.

• Levelization can be performed in linear time with a simple recursive algorithm.

• Once the circuit is levelized, we simulate gates strictly in level order. In this way we avoid processing two events on gate $f$.

• The advantages:
  – We need no special scheduling (no timing wheel).
  – Levelization allows us to simulate the circuit with the smallest possible number of events.

• The disadvantages:
  – No loops are allowed (i.e. we cannot handle sequential circuits since they cannot be levelized).

• Levelized code is used extensively for large designs.
COMPiled code

• In event driven simulation, a large number of CPU instructions are devoted to the following steps
  – Looking up the values of signals
  – Table lookup to compute output
  – Generating the output, and scheduling it

• Compiled code eliminates this by translating the circuit to be simulated into a compiled set of machine language instructions.

• Consider the circuit below

• We can write this as a computer program in a high-level language, compile it, and run it to get the simulation output.
• In this example, the program would be:
  
c = a && b;
  if(m) {
    e = d;
  }
  else {
    e = c;
  }
  f = e && c;

• Advantage:
  
  – Once this is compiled, each line requires typically 1-2 machine language instructions, since most typical gates map directly to machine language instructions. This constitutes a speedup of roughly 500× over event driven simulation.

  – Also, when we compute a logical AND instruction, we can actually AND 32 pairs of signals (assuming a 32-bit-wide AND instruction).

• The disadvantage is that we have some up-front configuration time (for high level language code generation, compilation, linking etc). But this is not significant if we have a large numbers of vectors and large circuits.

• This type of simulator is heavily used in industry
HARDWARE ASSISTED TECHNIQUES

• Several such techniques have been devised:
  – Hardware implementation of conventional simulator:
    * Create event queue, gate lookup tables etc. in hardware.
    * Typically requires a year to build, but in the meanwhile software solutions get faster as well.
    * As a result, such techniques have become mostly obsolete.
    * Companies like ZYCAD and VALID were built around these ideas
  – Hardware implementation of levelized code simulator:
    * Specialized hardware architecture, with levelized list of gates in memory
    * Evaluate one gate per clock, with gate lookup tables in memory
    * Large circuits needed to be partitioned
    * Different partitions need to communicate information, which was a major bottleneck.
* YSE (Yorktown Simulation Engine) from IBM is the best example

- Hardware emulation:
  * Emulate the behavior of hardware using FPGA
  * An FPGA is a programmable device which is composed of many Lookup Blocks, which can be programmably wired.
  * Each logic block can implement any logic function of 3-5 variables, with memory as well.
  * So the task is to decompose the logic being implemented, into a network of such lookup blocks. FPGA vendors already provide such solutions.
  * Each FPGA can be programmed directly based on the HDL description of the circuit.
  * Typically a board consists of several FPGAs which can together implement large circuits.
  * Typical speed is 1/50 to 1/100 of real circuit speed. This is very high compared to any software solution (since software solutions essentially serialize the simulation of the gates in the circuit, which the emulator does not do)
  * FPGAs are typically expensive, plus such a scheme requires large startup time (due to
the programming that is required for each FPGA).

* Partitioning of functionality across FPGAs needs to be addressed as well.

* In spite of these disadvantages, this solution nevertheless is favored due to its high speed.