Reading Assignment

- This lecture: 3.5, 3.8
- Next lecture: 3.6
Outline

Compiled Simulation

Element Evaluation
Compiled Simulation

- Simulate the function of combinational circuits.
  - Inputs
    - An internal model (functional or structural) of the circuit to be simulated.
    - Multiple input vectors (stimuli/test cases). Each vector contains the same number of bits as circuit inputs.
  - Outputs
    - The corresponding output vector for each input vector, containing as many bits as circuit outputs.
    - Usually values of internal signals are also stored for debugging and other purposes.
Compiled Simulation

▸ Simulate the function of combinational circuits.
▸ Inputs
  ▸ An internal model (functional or structural) of the circuit to be simulated.
  ▸ Multiple input vectors (stimuli/test cases). Each vector contains the same number of bits as circuit inputs.
▸ Outputs
  ▸ The corresponding output vector for each input vector, containing as many bits as circuit outputs.
  ▸ Usually values of internal signals are also stored for debugging and other purposes.
Compiled Simulation

- Simulate the function of combinational circuits.
- Inputs
  - An internal model (functional or structural) of the circuit to be simulated.
  - Multiple input vectors (stimuli/test cases). Each vector contains the same number of bits as circuit inputs.
- Outputs
  - The corresponding output vector for each input vector, containing as many bits as circuit outputs.
  - Usually values of internal signals are also stored for debugging and other purposes.
The combinational part of a synchronous circuit is simulated.

- Inputs: A, B, Q. Outputs: F.
- Logic operations are mapped to host instructions.
  - Assume the host has a single register that serves as src/dest operands implicitly for various instructions.
- Logic signals are mapped to host variables (A, B, E, F, Q).
  - Assume an input vector is available in the corresponding variables before simulation starts.
The combinational part of a synchronous circuit is simulated. 
- Inputs: A, B, Q. Outputs: F.
- Logic operations are mapped to host instructions.
  - Assume the host has a single register that serves as src/dest operands implicitly for various instructions.
- Logic signals are mapped to host variables (A, B, E, F, Q).
  - Assume an input vector is available in the corresponding variables before simulation starts.
Example

▶ The combinational part of a synchronous circuit is simulated.
  ▶ Inputs: A, B, Q. Outputs: F.
▶ Logic operations are mapped to host instructions.
  ▶ Assume the host has a single register that serves as src/dest operands implicitly for various instructions.
▶ Logic signals are mapped to host variables (A, B, E, F, Q).
  ▶ Assume an input vector is available in the corresponding variables before simulation starts.
Problems to Be Solved

- How to generate host code from the internal model?
  - How to avoid large code size if the circuit is large?
- How to handle three-valued logic?
- How to improve efficiency?
- Let’s study the algorithm for compiled simulation.
Problems to Be Solved

- How to generate host code from the internal model?
  - How to avoid large code size if the circuit is large?
- How to handle three-valued logic?
  - How to improve efficiency?
  - Let’s study the algorithm for compiled simulation.
Problems to Be Solved

▶ How to generate host code from the internal model?
  ▶ How to avoid large code size if the circuit is large?
▶ How to handle three-valued logic?
▶ How to improve efficiency?
▶ Let’s study the algorithm for compiled simulation.
Problems to Be Solved

- How to generate host code from the internal model?
  - How to avoid large code size if the circuit is large?
- How to handle three-valued logic?
- How to improve efficiency?
- Let’s study the algorithm for compiled simulation.
Initialization

- Assume the netlist data structure is available.
- Assume the Element Table is sorted by the levels of the elements from the lowest to the highest.
  - Take an arbitrary order if there is a tie.
- SIGNAL_VALUE: an array
  - Store logic value for each signal.
- Assume an input vector is associated with the circuit inputs.
  - Since circuit inputs can be assigned a level of 0, one can move all of them to the front of the Element Table.
  - Then the input vector can be stored in an array and accessed using the same set of indices.
Assume the netlist data structure is available.
Assume the Element Table is sorted by the levels of the elements from the lowest to the highest.
  ▶ Take an arbitrary order if there is a tie.

SIGNAL_VALUE: an array
  ▶ Store logic value for each signal.

Assume an input vector is associated with the circuit inputs.
  ▶ Since circuit inputs can be assigned a level of 0, one can move all of them to the front of the Element Table.
  ▶ Then the input vector can be stored in an array and accessed using the same set of indices.
Initialization

- Assume the netlist data structure is available.
- Assume the Element Table is sorted by the levels of the elements from the lowest to the highest.
  - Take an arbitrary order if there is a tie.
- **SIGNAL_VALUE**: an array
  - Store logic value for each signal.
- Assume an input vector is associated with the circuit inputs.
  - Since circuit inputs can be assigned a level of 0, one can move all of them to the front of the Element Table.
  - Then the input vector can be stored in an array and accessed using the same set of indices.
Initialization

- Assume the netlist data structure is available.
- Assume the Element Table is sorted by the levels of the elements from the lowest to the highest.
  - Take an arbitrary order if there is a tie.
- SIGNAL_VALUE: an array
  - Store logic value for each signal.
- Assume an input vector is associated with the circuit inputs.
  - Since circuit inputs can be assigned a level of 0, one can move all of them to the front of the Element Table.
  - Then the input vector can be stored in an array and accessed using the same set of indices.
The Compiled Simulation Algorithm

for (i = 1 to # of elements) {
    x = the i’th element in the Element Table;

    if (x is a circuit input) {
        s = fanout signal of x; // x should only have one fanout
        SIGNAL_VALUE[s] = the bit corresponding to x in the input vector;
    }
    else if (x is a circuit output) {
        s = fanin signal of x; // x should only have one fanin
        report the output bit of x as SIGNAL_VALUE[s];
    }
    else {
        Locate all fanin signals of x and obtain their values from SIGNAL_VALUE;
        Compute output values of x depending on functionality of x;
        Locate all fanout signals of x and store output values to SIGNAL_VALUE;
    }
}

The algorithm may be executed directly or be used to generate code for simulation.
Correctness is guaranteed if,

- The fanin signals of an element are ready in SIGNAL_VALUE before the computation of output values.

Since we visit the elements in the order of their levels, their fanin signals must be ready at the time of visit.

- Level of an element > level of its fanins

Complexity

- Space: $O(n)$ for SIGNAL_VALUE.
- Time: $O(m + n)$, the algorithm visit each edge once.

Note that the algorithm does not need the Fanout Table (for signal fanouts).
Algorithm Analysis

- Correctness is guaranteed if,
  - The fanin signals of an element are ready in SIGNAL_VALUE before the computation of output values.
  - Since we visit the elements in the order of their levels, their fanin signals must be ready at the time of visit.
    - Level of an element $> \text{level of its fanins}$

- Complexity
  - Space: $O(n)$ for SIGNAL_VALUE.
  - Time: $O(m + n)$, the algorithm visit each edge once.
  - Note that the algorithm does not need the Fanout Table (for signal fanouts).
Algorithm Analysis

- Correctness is guaranteed if,
  - The fanin signals of an element are ready in SIGNAL_VALUE before the computation of output values.
- Since we visit the elements in the order of their levels, their fanin signals must be ready at the time of visit.
  - Level of an element > level of its fanins

- Complexity
  - Space: $O(n)$ for SIGNAL_VALUE.
  - Time: $O(m + n)$, the algorithm visit each edge once.

- Note that the algorithm does not need the Fanout Table (for signal fanouts).
Algorithm Analysis

- Correctness is guaranteed if,
  - The fanin signals of an element are ready in SIGNAL_VALUE before the computation of output values.
- Since we visit the elements in the order of their levels, their fanin signals must be ready at the time of visit.
  - Level of an element $\geq$ level of its fanins
- Complexity
  - Space: $O(n)$ for SIGNAL_VALUE.
  - Time: $O(m + n)$, the algorithm visit each edge once.
- Note that the algorithm does not need the Fanout Table (for signal fanouts).
For graph-based functional models, only the fanins of each operation node are available.

- We need to access fanout signals of elements because an element may generate multiple fanout signals.
- This is usually not necessary for functional models where each node has one fanout signal.
  - This signal could be used by multiple nodes as inputs.
- We can extend the algorithm to simulate graph-based functional models by storing each signal value with the node generating it.
Extensions I: Graph-Based Functional Models

- For graph-based functional models, only the fanins of each operation node are available.
- We need to access fanout signals of elements because an element may generate multiple fanout signals.
- This is usually not necessary for functional models where each node has one fanout signal.
  - This signal could be used by multiple nodes as inputs.
- We can extend the algorithm to simulate graph-based functional models by storing each signal value with the node generating it.
Extensions I: Graph-Based Functional Models

- For graph-based functional models, only the fanins of each operation node are available.
- We need to access fanout signals of elements because an element may generate multiple fanout signals.
- This is usually not necessary for functional models where each node has one fanout signal.
  - This signal could be used by multiple nodes as inputs.
- We can extend the algorithm to simulate graph-based functional models by storing each signal value with the node generating it.
For graph-based functional models, only the fanins of each operation node are available.

We need to access fanout signals of elements because an element may generate multiple fanout signals.

This is usually not necessary for functional models where each node has one fanout signal.
  - This signal could be used by multiple nodes as inputs.

We can extend the algorithm to simulate graph-based functional models by storing each signal value with the node generating it.
Most hosts support bit-wise logic operations.
  - e.g. \( \sim \) (NOT), \& (AND), \( \mid \) (OR), \( \sim \) (XOR) in C/C++/Java.

Instead of simulating one input vector using Boolean operations, we may simulate multiple input vectors utilizing those bit-wise logic operations.

We may simulate \( W \) input vectors in parallel for an operand of \( W \) bits.
  - Typical \( W \): 32, 64, 128, 256
Most hosts support bit-wise logic operations.
  e.g. ~ (NOT), & (AND), | (OR), ^ (XOR) in C/C++/Java.

Instead of simulating one input vector using Boolean operations, we may simulate multiple input vectors utilizing those bit-wise logic operations.

We may simulate $W$ input vectors in parallel for an operand of $W$ bits.
  Typical $W$: 32, 64, 128, 256
Most hosts support bit-wise logic operations.
  e.g. \(\sim\) (NOT), \& (AND), \(\mid\) (OR), \(\sim\) (XOR) in C/C++/Java.

Instead of simulating one input vector using Boolean operations, we may simulate multiple input vectors utilizing those bit-wise logic operations.

We may simulate \(W\) input vectors in parallel for an operand of \(W\) bits.
  Typical \(W\): 32, 64, 128, 256
The bit-wise operations also enable us to encode three-valued logic using multiple bits and to simulate the circuit as Boolean logic.

- A possible encoding:
  - 0 → 00
  - 1 → 11
  - u → 01

- Bit-wise operations give correct results for AND and OR
- We need to swap the two bits after the bit-wise NOT
  - You cannot assign 10 to u because if so, then
    \(\text{AND}(u, u) = 10 \& 01 = 00 = 0\).

- This is especially helpful for simulation of functional models where nodes are mostly basic logic operations.
The bit-wise operations also enable us to encode three-valued logic using multiple bits and to simulate the circuit as Boolean logic.

A possible encoding:
- 0 → 00
- 1 → 11
- u → 01

Bit-wise operations give correct results for AND and OR.

We need to swap the two bits after the bit-wise NOT.
- You cannot assign 10 to u because if so, then
  \[ \text{AND}(u, u) = 10 \& 01 = 00 = 0. \]

This is especially helpful for simulation of functional models where nodes are mostly basic logic operations.
The bit-wise operations also enable us to encode three-valued logic using multiple bits and to simulate the circuit as Boolean logic.

A possible encoding:
- 0 → 00
- 1 → 11
- u → 01

Bit-wise operations give correct results for AND and OR
- We need to swap the two bits after the bit-wise NOT
  - You cannot assign 10 to u because if so, then
    \[ AND(u, u) = 10 \& 01 = 00 = 0. \]
- This is especially helpful for simulation of functional models where nodes are mostly basic logic operations.
The bit-wise operations also enable us to encode three-valued logic using multiple bits and to simulate the circuit as Boolean logic.

A possible encoding:
- 0 → 00
- 1 → 11
- u → 01

Bit-wise operations give correct results for AND and OR.

We need to swap the two bits after the bit-wise NOT.
- You cannot assign 10 to u because if so, then
  \[ \text{AND}(u, u) = 10 \& 01 = 00 = 0. \]

This is especially helpful for simulation of functional models where nodes are mostly basic logic operations.
Extensions III: Three-Valued Logic

- The bit-wise operations also enable us to encode three-valued logic using multiple bits and to simulate the circuit as Boolean logic.
- A possible encoding:
  - 0 → 00
  - 1 → 11
  - u → 01
- Bit-wise operations give correct results for AND and OR
- We need to swap the two bits after the bit-wise NOT
  - You cannot assign 10 to u because if so, then \( AND(u, u) = 10 \& 01 = 00 = 0 \). This is especially helpful for simulation of functional models where nodes are mostly basic logic operations.
Since there is no notion of time for compiled simulation, only cycle-to-cycle behavior can be simulated.

An initial state should be assigned.

- Can make use of three-valued logic to cover a large set of actual physical states.

An input vector is assumed to be ready at the primary inputs when the clock cycle begins.

An output vector including the primary outputs and the next state bits is obtained by one run of compiled simulation.

- Timing is verified separately via STA.

Need to run compiled simulation $C$ times for $C$ cycles.

- No, you cannot parallelize those $C$ simulations.
Extensions IV: Synchronous Circuits

► Since there is no notion of time for compiled simulation, only cycle-to-cycle behavior can be simulated.
► An initial state should be assigned.
  ▶ Can make use of three-valued logic to cover a large set of actual physical states.
► An input vector is assumed to be ready at the primary inputs when the clock cycle begins.
► An output vector including the primary outputs and the next state bits is obtained by one run of compiled simulation.
  ▶ Timing is verified separately via STA.
► Need to run compiled simulation $C$ times for $C$ cycles.
  ▶ No, you cannot parallelize those $C$ simulations.
Since there is no notion of time for compiled simulation, only cycle-to-cycle behavior can be simulated.

An initial state should be assigned.
  - Can make use of three-valued logic to cover a large set of actual physical states.

An input vector is assumed to be ready at the primary inputs when the clock cycle begins.

An output vector including the primary outputs and the next state bits is obtained by one run of compiled simulation.
  - Timing is verified separately via STA.

Need to run compiled simulation $C$ times for $C$ cycles.
  - No, you cannot parallelize those $C$ simulations.
Extensions IV: Synchronous Circuits

- Since there is no notion of time for compiled simulation, only cycle-to-cycle behavior can be simulated.
- An initial state should be assigned.
  - Can make use of three-valued logic to cover a large set of actual physical states.
- An input vector is assumed to be ready at the primary inputs when the clock cycle begins.
- An output vector including the primary outputs and the next state bits is obtained by one run of compiled simulation.
  - Timing is verified separately via STA.
- Need to run compiled simulation $C$ times for $C$ cycles.
  - No, you cannot parallelize those $C$ simulations.
Since there is no notion of time for compiled simulation, only cycle-to-cycle behavior can be simulated.

An initial state should be assigned.
  - Can make use of three-valued logic to cover a large set of actual physical states.

An input vector is assumed to be ready at the primary inputs when the clock cycle begins.

An output vector including the primary outputs and the next state bits is obtained by one run of compiled simulation.
  - Timing is verified separately via STA.

Need to run compiled simulation $C$ times for $C$ cycles.
  - No, you cannot parallelize those $C$ simulations.
Asynchronous circuits can be simulated if the timing behavior can be separated from how signal changes.

Figure 3.10  (a) Circuit used as pulse generator (b) Correct model for compiled simulation (Abramovici et al., 1990)
Outline

Compiled Simulation

Element Evaluation
Input Value Storage

In addition to storing one value per signal, we may store values at inputs of elements. Need $O(m)$ extra storage.

(Abramovici et al., 1990)
Discussions

- To support input value storage per element, the Fanout Table (for signal fanouts) need to be used.
  - Though the Fanin Table (for element fanins) is not needed any more.
- The passing of signals to element inputs are explicitly modeled.
  - Useful for fault modeling.
- More efficient evaluation for elements can be supported.
  - e.g. via the use of truth tables and apply to multi-valued logics.
To support input value storage per element, the Fanout Table (for signal fanouts) need to be used.
  - Though the Fanin Table (for element fanins) is not needed any more.

The passing of signals to element inputs are explicitly modeled.
  - Useful for fault modeling.

More efficient evaluation for elements can be supported.
  - e.g. via the use of truth tables and apply to multi-valued logics.
Discussions

- To support input value storage per element, the Fanout Table (for signal fanouts) need to be used.
  - Though the Fanin Table (for element fanins) is not needed any more.
- The passing of signals to element inputs are explicitly modeled.
  - Useful for fault modeling.
- More efficient evaluation for elements can be supported.
  - e.g. via the use of truth tables and apply to multi-valued logics.
Input Scanning

- Apply to elements of types AND/OR/NAND/NOR
  - If any input is $c$, then the output is $c \oplus i$.
  - Otherwise, the output is $\overline{c} \oplus i$.
- Better performance on modern processors than approaches using different code pieces to evaluate different types.
  - e.g. a switch that need multiple branches,
  - Or function pointers (virtual functions for OOP) that need additional memory visits.

(Fig. 3.20, Abramovici et al., 1990)
Input Scanning

- Apply to elements of types AND/OR/NAND/NOR
  - If any input is $c$, then the output is $c \oplus i$.
  - Otherwise, the output is $\overline{c} \oplus i$.
- Better performance on modern processors than approaches using different code pieces to evaluate different types.
  - e.g. a switch that need multiple branches,
  - Or function pointers (virtual functions for OOP) that need additional memory visits.

(Fig. 3.20, Abramovici et al., 1990)
Summary

- Compiled simulation evaluates elements from the lowest level to the highest level, and computes circuit outputs from circuit inputs.