Topic VIII

Formal Hardware Verification (I)
Circuit Modeling

系統晶片驗證
SoC Verification

Sep, 2004

Administrative Information

- Homework #1 due next Monday (11/22)
  - Q&A available on TA's website
- Please pay attention to class website from time to time
  - Newly-added Q&A (tips)
  - Your grades
- There will be quiz… as bonus points

What we will learn in this topic…

1. Definition of formal (hardware) verification
2. Modeling of a hardware system
3. Practical modeling issues

It's a warm-up.

Hope you can totally understand it

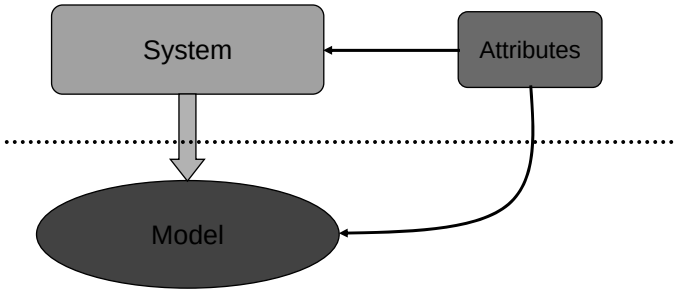
so that we can continue to the detailed formal verification engine algorithms later…

What is formal verification?

"The expression 'formal verification', as it appears in the literature, refers to a variety of (often quite different) methods used to prove that a model of a system has certain specified attributes. What distinguishes _formal_ verification from other undertakings also called 'verification', is that _formal_ verification conveys a promise of mathematical certainty. The certainty is that if a model is formally verified to have a given attribute, then no behavior or execution of the model ever can be found to be contradict this"

Robert Kurshan, "Computer-Aided Verification of Coordinating Processe"

In other words, formal verification is…



- Variety of verification methods (often quite different)
- What distinguishes formal?
  - Conveys a promise of mathematical certainty
    ➔ Guarantees that no behavior or execution of the model can ever be found to contradict the attributes

Some key phrases from the previous slide…

1. A system
   - Continuous / discrete time
   - Finite / infinite states
   - Hardware / software
2. Model of a system
3. Attributes
4. Mathematical certainty

Model of a System

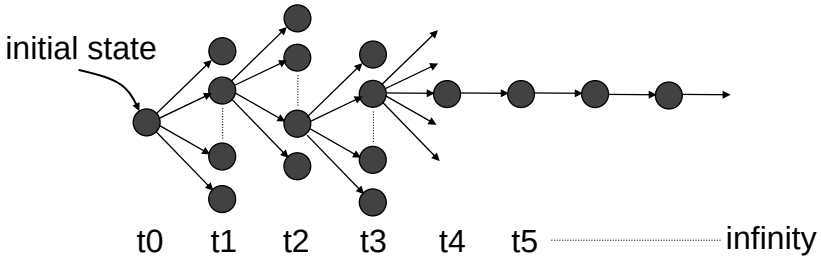
- A mathematical representation of a system, which can usually be represented in certain data structures and analyzed by computer-aided algorithms
  - Simplification and assumption are usually needed
  - Usually in the form of a state machine or an automata

Attributes of a System

- Some characteristics, behavior of a system, which can be summarized with rules, and these rules can usually be represented as mathematical formulae
  - Rule = formula of variables
  - Variables in the model of a system
    - Input / output variables
    - Internal variables (signals)
    - State variables
    - Time variables

"Mathematical Certainty" in Formal Verification

- Space exhaustiveness
  - Verify all input combinations of the system
- Time exhaustiveness
  - Verify system behavior from initial state to time infinity



We will revisit these key phrases later.

As we mentioned earlier, there are different verification methods, which are specifically applied to various models of systems.

However, almost all of them are intractable.

But "formal hardware verification" is the one that at least we can see *a shred of light* in practical applications

What distinguishes hardware verification?

(Well, we are talking about *digital* hardware verification here; it's indeed a quite different story in *analog* verification)

- Finite state space
  - Number of possible states is confined by the number of registers
  - "Reachable states" are subset of all the possible states
- Finite number system
  - Modular 2^n number system in general
  - Special case: Boolean logic

Well, in a practical modern design,
the number of registers can be as high as 1M+
➔ Number of possible states = 2^{1M}
(Mission impossible)

How can a commercial tool handle such a big design?

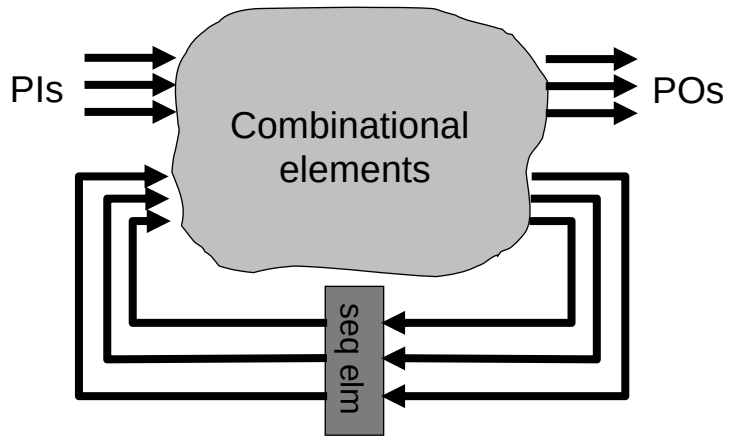
First, let's see how the circuits are modeled.

Model for a Hardware System

- Circuit = combinational + sequential

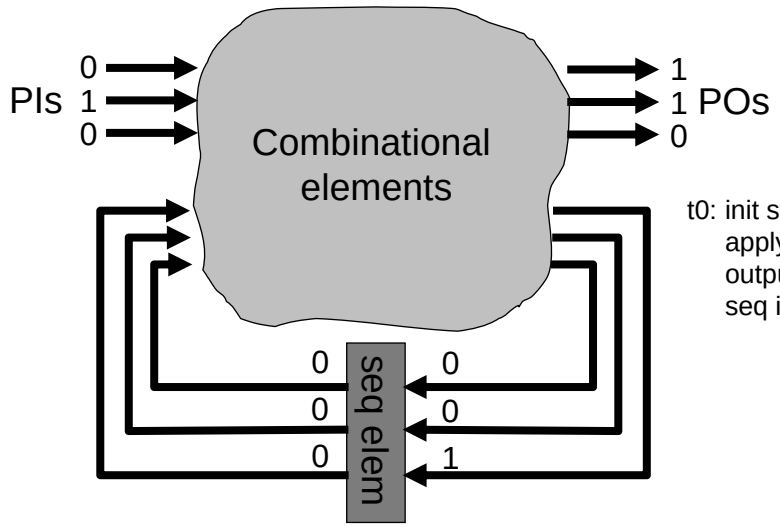
1. Combinational elements
   (output and inputs are in the same cycle)
   - Primary inputs (PIs), primary outputs (POs)
   - Primitive gates (and, or, nand, nor, inv, buf)
   - Complex gates (xor, xnor, mux, aoi, oai, adder, etc)
   - Tri-state cells
   - Interconnection
   - Others
2. Sequential elements
   (output gets the value at the next cycle)
   - Flip-flops (edge-triggered)
   - Latches (level sensitive)
   - Complex sequential elements (memory, etc)

Hoffman Model

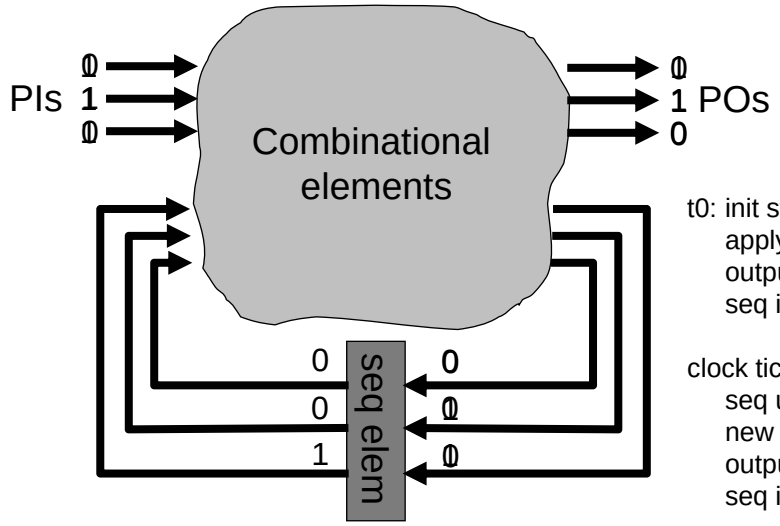


Note: use 0-delay model

Hoffman Model --- Simulation



Hoffman Model --- Simulation



How to define data structures for a circuit?

- Graph-based data structure
  - Graph → circuit
  - Nodes → gates (and/or pins, ports, etc)
  - Edges → wires (nets)
- For front-end applications (e.g. synthesis, function verification)
  - (circuit + gates) are enough
- For back-end applications (e.g. layout, timing analysis)
  - Pins, ports, nets, etc are usually necessary

class Circuit should at least contain…

1. The circuit netlist (graph)
   - A parser routine to transform the design into the circuit/gate data structures
   - Manage the construction and destruction of the gates
2. Circuit operating routines
   - Netlist traversal (DFS/BFS)
   - Netlist transformation (add/delete/transform)
3. Interface to other packages
   - Simulation, timing/clock analysis, etc
   - Formal engines (ATPG, BDD, SAT, etc)

class Circuit --- a simple example

```
class Circuit
{
public:
   Circuit(const string& inFile);
   ~Circuit();

   bool simulate(const string& vectorFile);
   friend ostream& operator << (ostream&, const
       Circuit&);

private:
   vector<Gate *>      _piList;
   vector<Gate *>      _poList;
   vector<Gate *>      _seqElmList;
};
```


class Gate should at least contain…

1. Identity
   - Name, id, type, level, value, etc.
2. Connections
   - Fanins, fanouts
3. Traversal, debugging utilities
   - Marks, temporary values, etc.

class Gate --- a simple example

```
class Gate
{
public:
   Gate(unsigned numFanins = 0);
   virtual ~Gate();

   const string& getName() const;
   virtual bool simulate() = 0;
   friend ostream& operator << (ostream&, const Gate*);

protected:
   string             _name;
   unsigned           _id;
   GateValue          _value;
   GateFlag           _flag;
   vector<Gate *>     _faninList;
   vector<Gate *>     _fanoutList;
};
```


Inherent classes for Gate --- example

```
class DffGate : public Gate
{
public:
   DffGate(unsigned numFanins = 0);
   ~DffGate();

   bool simulate();

private:
   GateValue        _prevDinValue;
   GateValue        _prevClkValue;
};
```

Using “enum” for “GateFlag”

```
class Gate
{
public:
   enum GateFlag {
      MARK_1    = 0x0001,
      MARK_2    = 0x0010,
      // meta enum
      SCHEDULED = MARK_1,
      VISITED   = MARK_1,
      ACTIVE    = MARK_2,
      // dummy end
      DUMMY_END
   };
};

void Gate::setFlag(Gate::GateFlag f) { _flag |= f; }
bool Gate::isFlagSet(Gate::GateFlag f) const
{ return (_flag & f); }
```


Sounds simple??

But there are many issues to consider…

1. Gate types vs. inherent classes

◆ Gate types

```
class Gate
{
   GateType       _type;
};
```

◆ Inherence is more C++ like

◆ How many types?

- I/Os, Primitive, complex, tri-states,… could be many
- (suggestion) Reducing number of types
  - Converted to equivalent gates (e.g. OR → (not, AND))
  - (Pros) Easier to implement and maintain
  - (Cons) More difficult to debug
  - (???) May have impact on performance

2. Primitive or complex gates?

◆ [Extreme 1] Decompose all gates into simple primitives (and limit the number of inputs)

e.g. decompose all gates to 2-input NANDs

- (Pros) Further simplify the implementation

◆ [Extreme 2] Keep all the original types in the design

e.g. xor, mux, tri-state, adder, etc

- May need to use generic functional representation (e.g. BDDs)
- (Pros) Closer to original circuit → easier to debug

How to choose a balance between these 2 extreame?

➔ Depending on your applications

3. Single-output or multi-output gate?

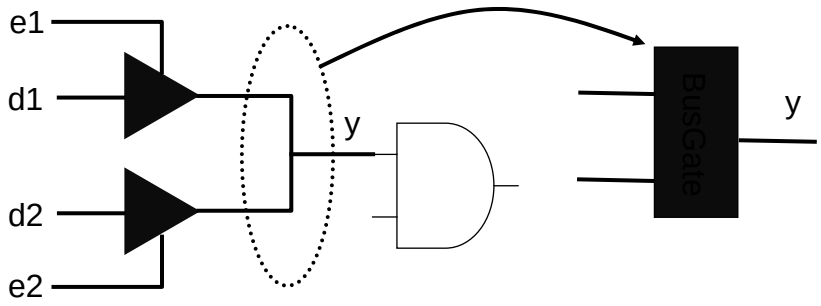
- In the exemplary classes above, we assume each gate represents a single function (i.e. single output)
- However, some gates may have multiple funcitons
  - e.g. Adder → { sum, carry }
  - e.g. DFF → { Q, Q_bar }
- [note] Different from _fanoutList;
- Multi-output gates
  - Data structure will be different from the above classes
  - How??
- Single-output gates
  - Adder - sum, carry?

4. Multi-driven gates?

- e.g.

  assign y = e1? d1: 1'bz;

  assign y = e2? d2: 1'bz;



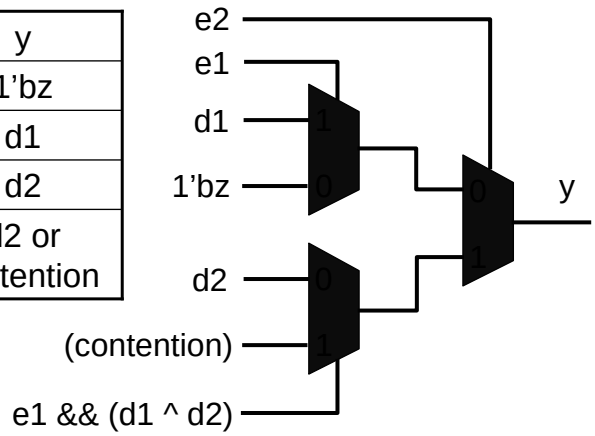
BusGate can be very complicated…

- For example,
  - How to build the BDD for the BusGate?
  - What is the CNF for the BusGate?
  - What if e1 = e2 = 1, d1 = 1, d2 = 0? (bus contention) What's the y value?

Use simpler gates to represent BusGate?

- For example, use Muxes

e1	e2	y
0	0	1'bz
1	0	d1
0	1	d2
1	1	d2 or contention



5. Combinational loops

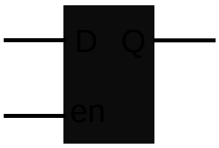
- If a circuit has combinational loops…
  - Feedbacks in topological ordering
    - Impact on cycle-based (0-delay) simulation
  - What if the loop oscillates?
  - Holding values → like sequential elements
- Can we just cut the loop and add a pseudo PI?
  - Extra behavior introduced
  - How about adding a constraint (PPI == loopHead)?
    - OK, but what about sequential effect?

Too complicated if we don't cut the loops…

- For some applications, it is OK
  - For example, in equivalence checking, if both golden and revised circuits have the same combinational loops ---
    - ➔ cut and match the loops
    - ➔ Treat those pseudo PIs as free variables
- For others, it is NOT OK but…
  - Need to watch out the false negative and false positive problems carefully

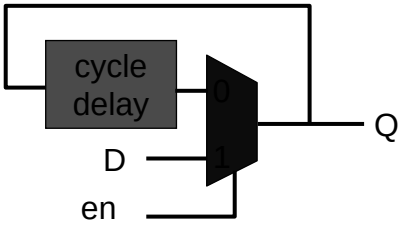
6. Latches --- sequential or combinational?

- D-Latch

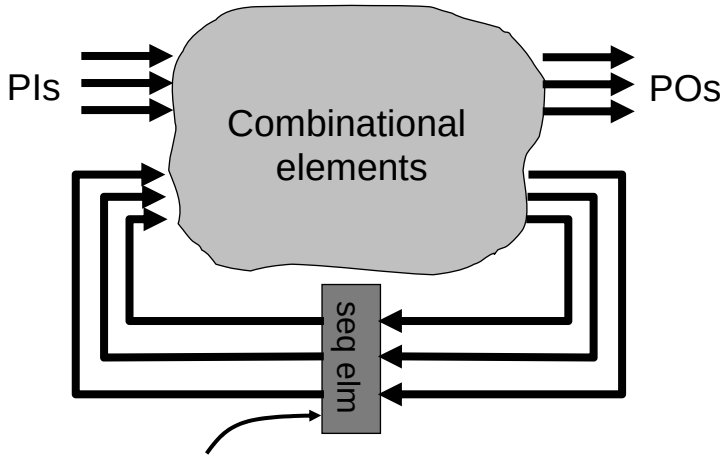


if (en == 1) Q = D; (buffer)
if (en == 0) Q(t) = Q(t-1); (hold)

- Equivalent to ---

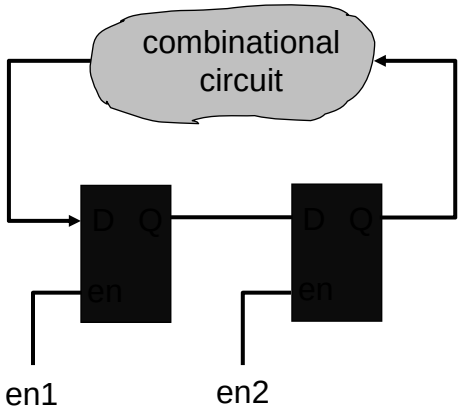


Remember, in Hoffman Model ---



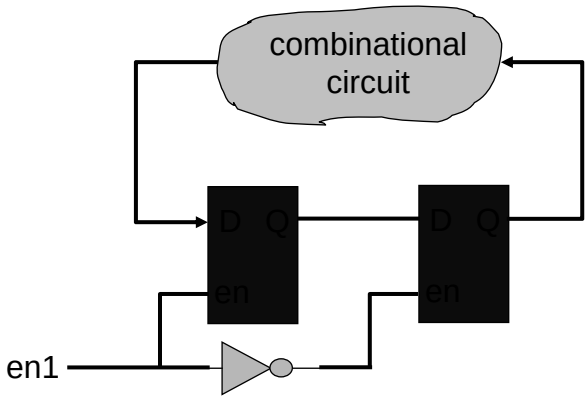
What if they are latches?
➔May have combinational loops

In other words…

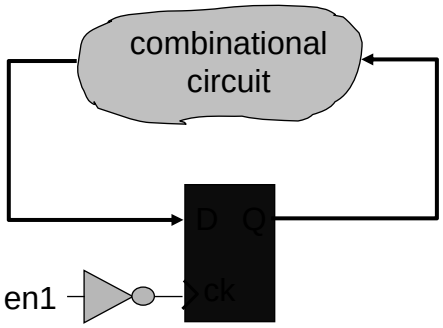


When en1 = en2 = 1, it's a combinational loop

However, very often it's a false loop…



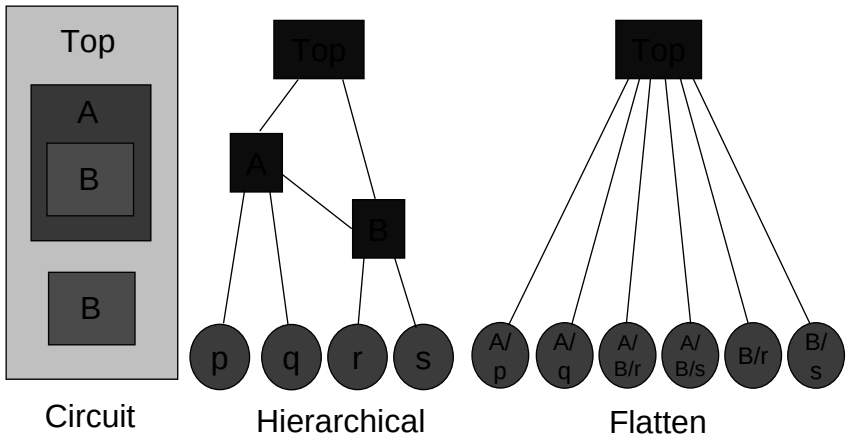
Latch Folding



Note: Perform latch folding first!!

7. Hierarchical vs. flatten

◆ The data structures mentioned above are for flatten netlist



Hierarchical Circuit Structure

- Pros
  - Save space
  - Properties in the same module won't be repeated in proof
- Cons
  - Difficult for evaluation (simulation, formal, etc)

8. Bit vs. word-level

- Note: original RTL designs are word-level
  - Adder, multiplier, data register, data mux, etc
- Word-level data structure
  - Pros
    - Closer to design intent (good for logic reasoning)
    - Can apply arithmetic techniques other than Boolean
  - Cons
    - More complicated in implementation

9. Other misc issues

- Design vs. cell library
  - Complicate parsing routine
- How to specify initial state?
  - A single state or set of states?
  - Init sequence?
  - Init constraints
- How to specify (multiple) clocks?