Sequential Synthesis

History: Combinational Logic → single FSM → Hierarchy of FSM's

SIS
Sequential Circuit Optimization (single machine)
Facilities for handling latches

MISII
Facilities for managing networks of FSMs

VIS ("handles" hierarchy)
Sequential Circuit Partitioning
Sequential Synthesis

Original -> Subckt 1
<table>
<thead>
<tr>
<th></th>
<th>Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Partition</td>
</tr>
<tr>
<td></td>
<td>Combine/ Flatten</td>
</tr>
<tr>
<td>Final -&gt; Subckt n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Optimize</td>
</tr>
<tr>
<td></td>
<td>Interface logic (asynchronous?)</td>
</tr>
</tbody>
</table>

Partition for layout...
What are Combinational Circuits?

Definition: A circuit is combinational if it computes a function which depends only on the current inputs applied to the circuit; for every input set of values, there is a unique output set of values.

- Acyclic circuits are necessarily combinational
- Cyclic circuits can be combinational,
  - in fact, there are combinational circuits whose minimal implementation must have cycles [Kautz 1970]
- Recent work on checking if circuit is combinational [Malik ‘94, Shiple ‘95]. These are based on X-valued simulation.
What are Sequential Circuits?

Some sequential circuits have memory elements.
- Synchronous circuits have clocked latches.
- Asynchronous circuits may or may not have latches (e.g. C-elements), but these are not clocked.

Feedback (cyclic) is a necessary, but not sufficient condition for a circuit to be sequential.

Synthesis of sequential circuits is not as well developed as combinational. (only small circuits)

Sequential synthesis techniques are not really used in commercial software (except maybe retiming).

Sequential verification is a problem.
Example

Registers and Latches (Netlist) \(\rightarrow\) State Transition Graph (STG)

The above circuit is sequential since primary output depends on the state and primary inputs.
Representations of Sequential Circuits

Transition relation is $T(pi, ps, ns)$ or $T(pi, ps, ns, po)$. It is the characteristic function of all edges of the STG.

$$T(pi, ps, ns) = \Pi(ns_i \oplus f_i(pi, ps))$$
Representation of Sequential Circuits

Each representation has its advantages and disadvantages.

- **STG** is like a two-level description
  - can blow up.
- **Netlist** only way for large circuits.
- **Transition relation** T usually represented by BDD’s.
  - Can blow up, but
  - can also express it as separate relations for each latch which are implicitly conjoined:

\[
T = \prod T_i(\pi_i,\psi_i,\eta_i)
\]
Example - Highway Light (Verilog)

```
module highway_control(
  input clk, car_present, enable_hwy, short_time, long_timer, hwy_light, hwy_start_timer, enable_farm);

input clk, car_present, enable_hwy, short_time, long_timer;
output hwy_light, hwy_start_timer, enable_farm;
wire wire car_present;
wire wire short_time, long_time, hwy_start_timer, enable_farm;

color reg hwy_light;

initial hwy_light = GREEN;
assign hwy_start_timer = (hwy_light == GREEN) && (car_present == YES) && long_timer || (hwy_light == RED) && enable_hwy;
assign enable_farm = (hwy_light == YELLOW) && short_time;

always @(posedge clk)
begin
  case (hwy_light)
    GREEN: if(car_present == YES) && long_timer
      hwy_light = YELLOW;
    YELLOW: if(short_time)
      hwy_light = RED;
    RED: if(enable_hwy)
      hwy_light = GREEN;
  endcase
end
```

Finite State Machines

Finite State Machines in STG or transition relation form are a behavioral view of sequential circuits.
- They describe their transitional behavior.
- They can distinguish among a finite number of classes of input sequence histories:
  - These classes are the internal states of the machine.

Moore Machine: is a quintuple: \( M(S, I, O, \delta, \lambda) \)
- \( S \): finite non-empty set of states
- \( I \): finite non-empty set of inputs
- \( O \): finite non-empty set of outputs
- \( \delta \): \( S \times I \rightarrow S \) transition (or next state) function
- \( \lambda \): \( S \rightarrow O \) output function (note: output only a function of present state)
FSM’s (continued)

Mealy Machine: $M(S, I, O, \delta, \lambda)$ but
- $\lambda: S \times I \to O$ (i.e. output depends on both present state and present input)
- for digital circuits, typically $I = \{0,1\}^m$ and $O = \{0,1\}^n$

In addition, (for both Moore and Mealy machines)
certain states are classified as reset or initial states

Finite automata are similar to FSM’s, but
- they do not produce any outputs,
- they just accept input sequences (an accepting set of states is given).
Representing State Machines

State Transition Graph
Example: Traffic Light Controller - Mealy Machine

Input predicate/outputs

STG

- HG: not(c and t1)/
  hl=GREEN; fl= RED; st=0
  ts/  
  hl=RED; fl=YELLOW; st=1
- FY: not(ts)/
  hl=RED; fl=YELLOW; st=0
  not(c or t1)/
  hl=RED; fl=GREEN; st=1

- HY: c and t1/
  hl=GREEN; fl= RED; st=1
  not(ts)/
  hl=RED; fl=RED; st=0

- FG: not(not(c or t1)/
  hl=RED; fl=GREEN; st=0
  ts/  
  hl=RED; fl=GREEN; st=1
Representing State Machines

State Transition Table
Example: Traffic Light Controller - Mealy Machine

<table>
<thead>
<tr>
<th>PS</th>
<th>IN</th>
<th>NS</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>HG</td>
<td>not(c and t1)</td>
<td>HG</td>
<td>hl=GREEN; fl=RED; st=1</td>
</tr>
<tr>
<td>HG</td>
<td>c and t1</td>
<td>HY</td>
<td>hl=GREEN; fl=RED; st=1</td>
</tr>
<tr>
<td>HY</td>
<td>not(ts)</td>
<td>HY</td>
<td>hl=YELLOW; fl=RED; st=0</td>
</tr>
<tr>
<td>HY</td>
<td>ts</td>
<td>FG</td>
<td>hl=YELLOW; fl=RED; st=1</td>
</tr>
<tr>
<td>FG</td>
<td>not(not(c) or t1)</td>
<td>FG</td>
<td>hl=RED; fl=GREEN; st=0</td>
</tr>
<tr>
<td>FG</td>
<td>not(c) or t1</td>
<td>FY</td>
<td>hl=RED; fl=GREEN; st=1</td>
</tr>
<tr>
<td>FY</td>
<td>not(ts)</td>
<td>FY</td>
<td>hl=RED; fl=YELLOW; st=0</td>
</tr>
<tr>
<td>FY</td>
<td>ts</td>
<td>HG</td>
<td>hl=RED; fl=YELLOW; st=1</td>
</tr>
</tbody>
</table>
Transition and Output Relations

- \( R \subseteq I \times S \times S \times O \) is the transition and output relation
- \( r = (in, s_{ps}, s_{ns}, out) \in R \) iff input \( in \) causes a transition from \( s_{ps} \) to \( s_{ns} \) and produces output \( out \).
- Since \( R \) is a set, it can be represented by its characteristic function (and hence as a BDD).
- Depending on the application, it may be preferable to keep the transition and output relation separate:
  - Transition Relation: \( R_\delta \subseteq I \times S \times S \)
  - Output Relation: \( R_\lambda \subseteq I \times S \times O \)
Non-Determinism and Incomplete Specification

In automata theory, non-determinism is associated with many transitions:
- From a given current state and under the same input conditions we may go to different states and have different outputs.
- Each behavior is considered valid.
- Non-determinism provides a compact way to describe a set of valid behaviors.

In classical sequential function theory, transition functions and output functions can be incompletely specified (functions can have don’t cares), i.e. defined only on a proper subset of their input space.
- where it is undefined, we consider it to allow any behavior.

Both methods describe sets of valid behaviors.
Non-Determinism versus Incomplete Specification

Given an input and present state:

• Non-determinism: some next states and outputs may be ruled out.
  - Result is that only a subset of next states and outputs are admissible for a transition.

• Don’t cares: all next states and outputs are allowed.
  - these may be because the given state is unreachable, so will never occur,
  - or the state is a binary code not used during the state assignment.
Non-Determinism versus Incomplete Specification

Incomplete transition structure: It may be that no next state is allowed.
- If this is because that input will never occur at that state we can “complete” the description by adding transitions to all states and allowing all outputs.
- On the other hand, we may want the machine to do nothing (e.g. as an automaton).
  - Sometimes we “complete” the transition structure by adding a dummy state and calling it a non-accepting state or a state whose output is an error signal.

All describe a set of behaviors. These are used to describe
- flexibility for the implementation during synthesis, and/or
- to describe a subset of acceptable behaviors.
Non-Determinism versus Incomplete Specification

- Optimization tools for logic synthesis and verification exploit, in various fashions, incomplete specification to achieve optimization objectives.
- Methods to exploit flexibility given by non-determinism have been devised [Kim and Newborn, Somenzi, Wang, Wanatabe, Kam and Villa]
- At the implementation level, only one of the possible next states and outputs in chosen (complete specification).
Incompletely Specified Machines

- Next state and output functions have don’t cares.
- However, for an implementation, $\delta$ and $\lambda$ are functions,
  - thus they are uniquely defined for each input and state combination.
- Don’t cares arise when some combinations are of no interest:
  - they will not occur or
  - their outputs will not be observed.
- For these, the next state or output may not be specified.
  - (In this case, $\delta$ and $\lambda$ are relations, but of special type.
    We should make sure we want these as don’t cares.)
- Such machines are called incompletely specified.
Example

By adding a dummy state this can be converted to a machine with only the output incompletely specified.

Could also specify “error” as the output when transitioning to the dummy state.

Alternatively (better for optimization), can interpret undefined next state as allowing any next state.
Initializing Sequences


Q: How many states does a circuit (implementation) with n memory elements have?
A: $2^n$, one for each possible vector of values of these memory elements.
   - Must assume on power on, that any of the $2^n$ states is possible.

M1: No initialization sequence possible
M2: Initialization sequence is possible
Initializing Sequences (cont)


• The set of states of normal operation forms a strongly connected component.

• Initializing Sequence: A sequence of input vectors that forces the machine into a known set of “reset states”.
  - May be implemented with extra hardware, using a single reset signal.

Pixley: If an aligning sequence exists for each state pair, then an initializing sequence exists.
**FSM Extraction**

*Problem:* Given a netlist, extract an FSM from it.

Extraction of the Transition and Output Relation

*Method 1:* Represent it by its characteristic function, \( \chi_\eta(i,p,o,n) \).

\[
\chi_\eta(i,p,o,n) = (\delta(i,p) \oplus n) \land (\lambda(i,p) \oplus o)
\]

\( \chi_\eta \) is the characteristic function of the transition and output relation.

- may be represented in several ways, the ROBDD representation seems to be most useful.
FSM Extraction

Explicit/Semi-Implicit Extraction of all Transitions
Method 2: (Ref: [Devadas, Ma, Newton 88])
Visit states starting from the reset states (in breadth-first-order).

```java
extract(C) {    /* C is the given circuit */
    st_table := { };    
    list := { };    
    foreach(s in reset_states)    
        add_list(list, s);    
    while((ps := next_unvisited(list)) != NIL) {    
        /* iterate until all states have been visited */    
        while(((in, ns, out) := generate_ns(ps)) != NIL) {    
            /* generate transitions from ps one by one */    
            st_table := st_table + {(in, ps, ns, out)};    
            if(!(in_list(list, ns)) add_list(list, ns);    
        }    
        mark_visited(list, ps);    
    }
    return(st_table); }
```

Of course, could do this in DFS order too, but see next slide.
FSM Extraction

Semi-Implicit Extraction of all Transitions

generate_ns(ps):

1. Set present state lines to ps.
2. Select a value for the next state and output lines (sequentially go through all possibilities).
3. Find input values (possibly none) that will result in that next state and output value. This need not be a minterm but may be a cube. Use ATPG justification techniques to find input cube (however, must also find a cover of input cubes - i.e. must enumeration all possible edges).

Semi-Implicit Extraction:
- method is exponential in the number of states
- may not be possible to represent $\chi_\eta$ in reasonable space as a STT

Implicit Method
- Use BDD’s to represent $\chi_\eta$. 
Interconnected FSMs - FSM Networks

Natural way of describing complex systems (hierarchy, decomposition).
- Naturally extracted from HDL's with modules as sub-processes.
Interconnected FSMs - FSM Networks (cont)

- Interconnected FSMs $\equiv$ Single product machine (similar to flattening in Boolean circuits)
- Directed Graph - Each node an FSM.
  - Arcs are variables used for communication.

- Similar to Boolean network, but
  - each node is an FSM
  - possibly cyclic.
FSM Networks

- Consider $k$ component machines
  \[ M_i = (I_i, S_i, O_i, \delta_i, \lambda_i), \ i = 1, \ldots, k \]
  interconnected to form a network of FSMs
  \[ M_N = M_1 \times M_2 \times \ldots \times M_k = (I, S_N, O, \delta_N, \lambda_N). \]
- $M_N$ is a single FSM consisting of
  - the state set of $M_N$ is $S_N = S_1 \times S_2 \times \ldots \times S_k$, and
  - $\delta_N$ and $\lambda_N$ are the next state and output mappings induced by
    the properties of the component machines.
- $M_N$ realizes the product (flattened) machine
- Using BDD's, the transition relation for the product machine is:
  \[ T(I,S,O,S') = \prod T_i(I_i,S_i,O_i,S'_i) \]
FSM Networks

If we had perfect optimization tools, single (product) machine is best, however:
- product machine is huge (state explosion problem)
- tools are heuristic (imperfect)

Tools needed for:
- synthesis of a network (optimize components separately, using global information)
- verification on a network (verify network of reduced components)
- restructuring FSM networks (decomposition/flattening)

Tools analogous to ones we have for Boolean networks.
Sequential Input Don’t Cares

Input don’t care vector sequences

- Machine $A$ does not output a certain combination / at $x$, when $B$ is in set of states $S_i$. The transitions of $B$ from states in $S_i$ under input / are therefore unspecified.
- Can exploit this in state minimization and state encoding algorithms.
Sequential Input Don’t Cares

Input don’t care sequences of vectors

Suppose that machine A (left) drives machine B (right).
- The two outputs of A are the inputs to B.
- We note that A does not produce all possible output sequences.
  - for instance, (11,11) is a don’t care input sequence for B.
- This implies that a certain sequence of transitions will not occur in B.

However, note that we can’t simply remove states in B.
1. In general, automaton may be nondeterministic. Then have to determinize it with "subset construction."

2. Note: product machine is incompletely specified.
   - Can use this to state-minimize M2.
   - Input don't care sequences are due to the constrained controllability of the driven machine B in a cascade A → B.
   - Papers by Unger, Kim-Newborn, Devadas, Somenzi, and Wang to exploit input don't care sequences for logical optimization.
End of lecture 22
Sequential Output Don't Cares

Due to the constrained observability of the driving machine A.

A feeds B. The third transition of A can output either I1 or I2, without changing the terminal behavior of the cascade $A \rightarrow B$. Called output expansion. Note that now machine A has a don't care on an output.
Sequential Output Don’t Cares

Output expansion produces a multiple-output FSM in which a transition outputs any element of a subset of values (rather than any element of all possible values as in the case of an unspecified transition. Should be called output non-determinism).

Modify state minimization procedures to exploit output don’t cares. In previous example sa2 becomes compatible with sa3. One less state after state minimization (at the beginning both A and B are individually state minimized).
Overview of FSM Optimization

Initial: FSM description
- provided by the designer as a state table
- extracted from netlist
- derived from HDL description
  - obtained as a by-product of high-level synthesis
  - translate to netlist, extract from netlist

State minimization: Combine equivalent states to reduce the number of states. For most cases, minimizing the states results in smaller logic, though this is not always true.

State assignment: Assign a unique binary code to each state. The logic structure depends on the assignment, thus this should be done optimally (NOVA, JEDI).

Minimization of a node: in an FSM network
Decomposition/factoring: of FSMs, collapsing/elimination
Sequential redundancy removal: using ATPG techniques