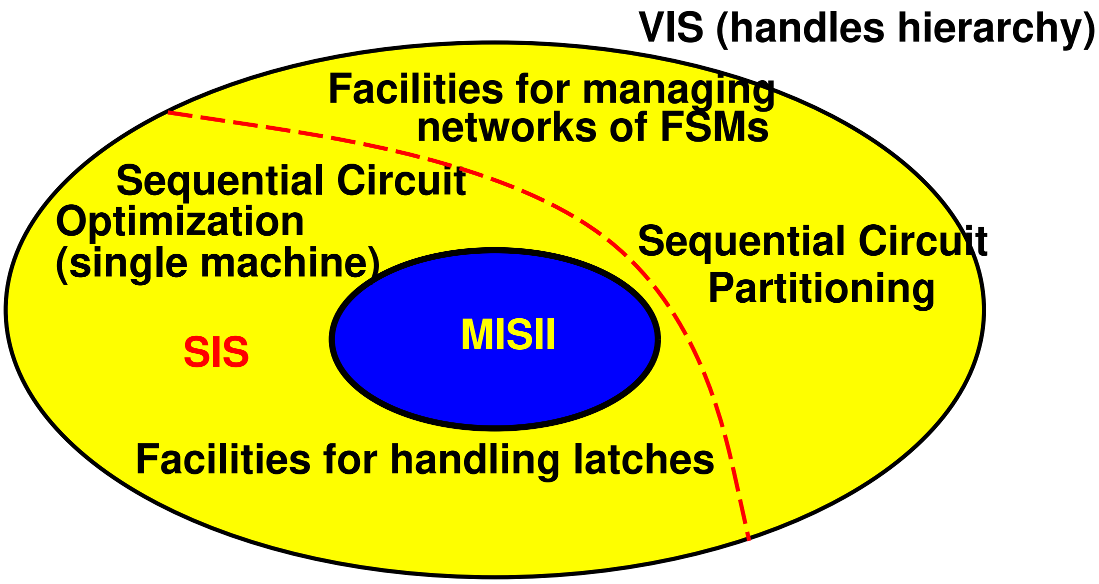
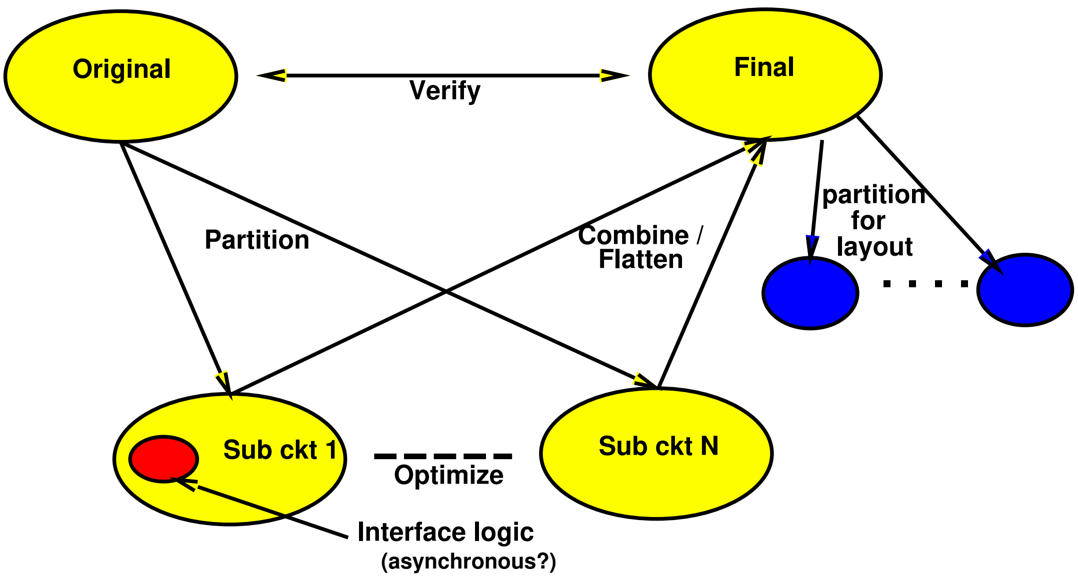
Sequential Synthesis

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





What are Combinational Circuits?

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

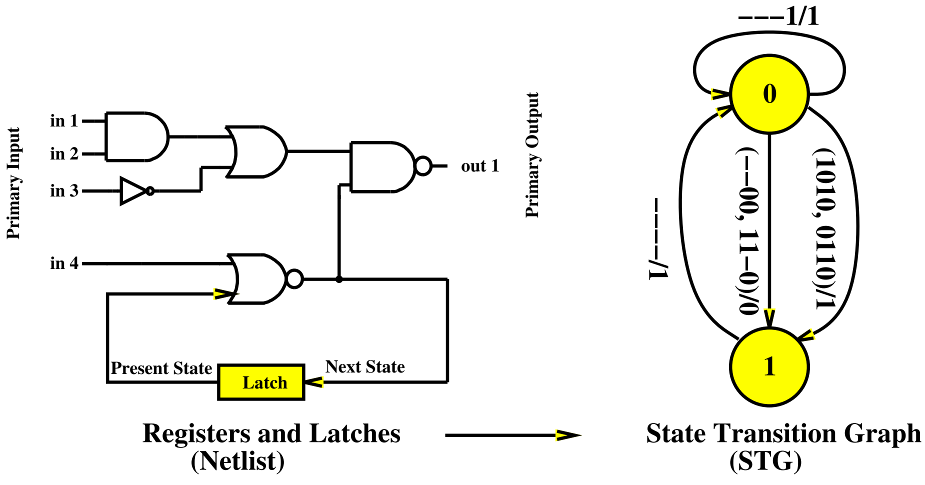
- Circuits with an acyclic underlying topology are combinational.
- Cyclic circuits can be combinational, in fact, there are combinational circuits whose minimal form must have cycles [Kautz 1970].
- Recent work on checking if circuit combinational [Malik'94, Shiple'95]. These are based on X-valued simulation.

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.

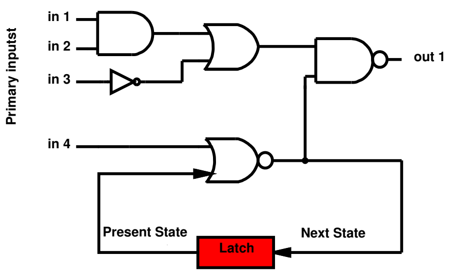
- Feedback (cyclic) is a necessary, but not sufficient condition for a circuit to be sequential.

- Synthesis of sequential circuits is not as well developed as combinational. Sequential synthesis techniques not really used in commercial software.

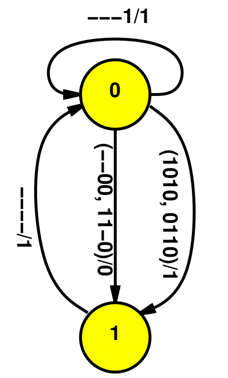


The above circuit is sequential since output depends on the state and input.

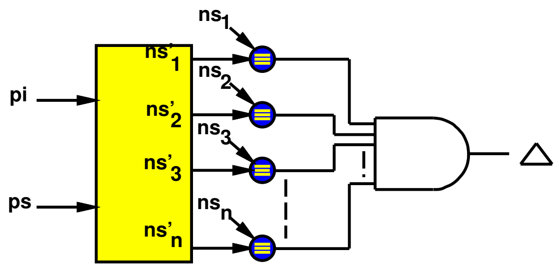
Representations of Sequential Circuits



Gates and Latches
(Netlist)



State Transition Graph
(STG)



Transition Relation

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

$$T(pi, ps, ns) = \prod (ns_i \overline{\oplus} f_i(pi_i, ps_i))$$

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

$$T = \prod T_i(pi_i, ps_i, ns_i)$$

Example - Highway Light (Verilog)

```
module hwy_control(clk, car_present, enable_hwy,
        short_timer, long_timer, hwy_light,
        hwy_start_timer, enable_farm);
input clk,car_present,enable_hwy,short_timer,
       long_timer;
output hwy_light,hwy_start_timer,enable_farm;
boolean wire car_present;
wire short_timer, long_timer, hwy_start_timer,
       enable_farm, enable_hwy;
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_timer);
always @(posedge clk) begin
        case (hwy_light)
        GREEN: if ((car_present == YES) &&
                long_timer) hwy_light = YELLOW;
        YELLOW: if (short_timer) hwy_light = RED;
        RED: if (enable_hwy) hwy_light = GREEN;
        endcase
end
endmodule
```

Finite State Machines

Finite State Machines in STG or transition relation form are a behavioral view of sequential circuits. They describe the transitional behavior of these circuits. They can distinguish among a **finite** number of classes of **input 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
$\delta : S \times I \mapsto S$ transition (or next state) function
$\lambda : S \mapsto O$ output function

Mealy Machine: $M = (S, I, O, \delta, \lambda)$ but

$\lambda : S \times I \mapsto O$

For digital circuits, typically $I = \{0, 1\}^m$ and $O = \{0, 1\}^n$.

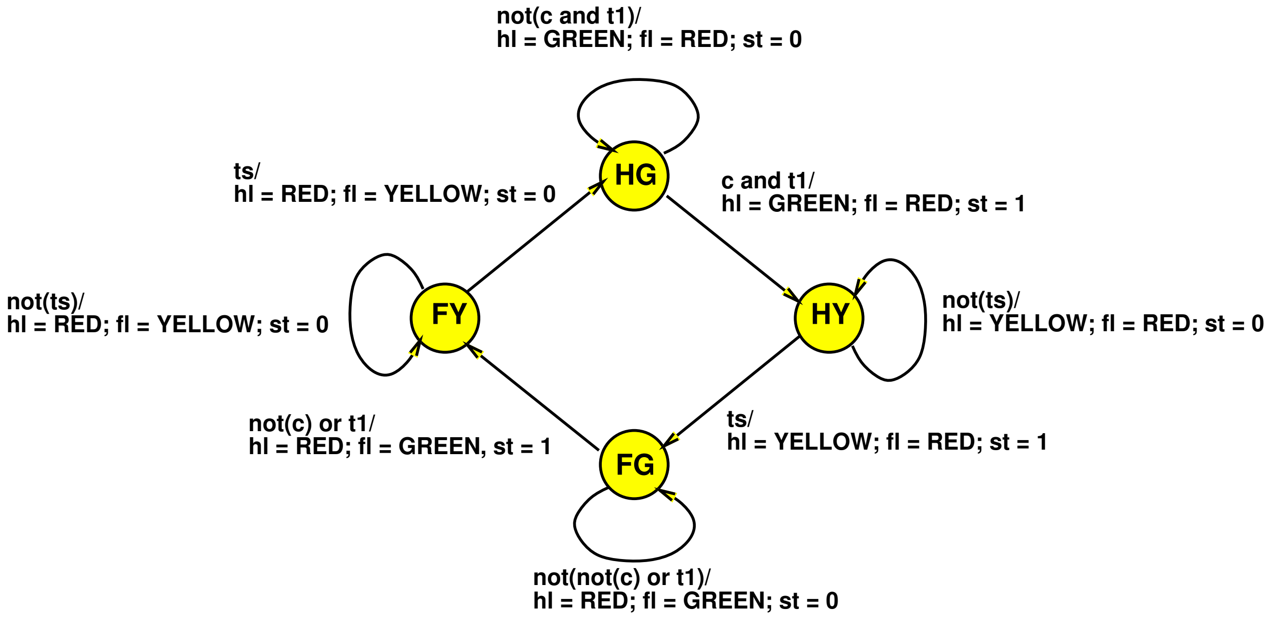
In addition certain states may be classified as *reset or initial states*.

Automata are similar to FSM's, however they do not produce any outputs, they just *accept* input sequences (*accepting set of states is given*).

Representing State Machines

State Transition Graphs and Tables

Example: Traffic Light Controller - Mealy machine



State Transition Graph: Example

PS	IN	NS	OUT
HG	not(c and t1)	HG	hl = GREEN; fl = RED; st = 0
HG	c and t1	HY	hl = GREEN; fl = RED; st = 1
HY	not(ts)	HY	hl = YELLOW; fl = RED; st = 0
HY	ts	FG	hl = YELLOW; fl = RED; st = 1
FG	not(not(c) or t1)	FG	hl = RED; fl = GREEN; st = 0
FG	not(c) or t1	FY	hl = RED; fl = GREEN; st = 1
FY	not(ts)	FY	hl = RED; fl = YELLOW; st = 0
FY	ts	HG	hl = RED; fl = YELLOW; st = 1

State Transition Table: Example

Representing State Machines

- State Transition Graphs and State Transition Tables are similar; the first is graphical, the second is tabular.
- In this example the edges (transitions) are labeled with general logic functions (predicates) of the inputs.
- Traditionally minterms or cubes have been used for the transitions (e.g. KISS format), especially for tables, since used as input to two-level minimizers. Minterms need the most edges and arbitrary logic functions (predicates) the least.

Transition and Output Relations

$R \subset I \times S \times S \times O$ is the transition and output relation.

$r = (in, s_{ps}, s_{ns}, out) \in R$ if and only if input in causes a transition from s_{ps} to s_{ns} and produces output out.

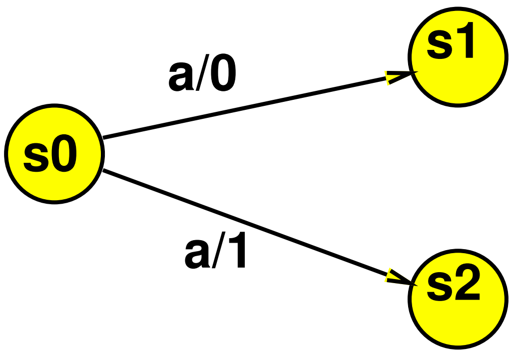
Since R is a set, it can be represented by its characteristic function (and hence as a BDD).

Depending on the application it may be preferable to keep the transition and output relation separate:

Transition Relation: $R_\delta \subset I \times S \times S$

Output Relation: $R_\lambda \subset I \times S \times O$

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. Nondeterminism 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** (i.e. the 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. This also describes a **set** of valid behaviors.

Non-Determinism and Incomplete Specification

Given an input and present state:

- **Nondeterminism:** some next states and outputs are ruled out. Result is subset of next states and outputs admissible for a transition.

- **Don't cares:** all next states and outputs are allowed. These may be because the given state can't be reached, so will never occur, or the state is a binary code not used during state assignment.

- **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 need to "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.

All describe a set of behaviors. These are used to describe flexibility for the implementation during synthesis, and to describe a subset of acceptable behaviors.

Non-Determinism and Incomplete Specification

- Optimization tools for logic synthesis and verification exploit in various fashions incomplete specification to achieve optimization objectives.

  More recently, methods to exploit flexibility given by non-determinism have been devised [Kim and Newborn, Somenzi, Wang, Watanabe, Kam&Villa]

- At the implementation level, only one of the possible next states and outputs is chosen (complete specification).

Incompletely Specified Machines

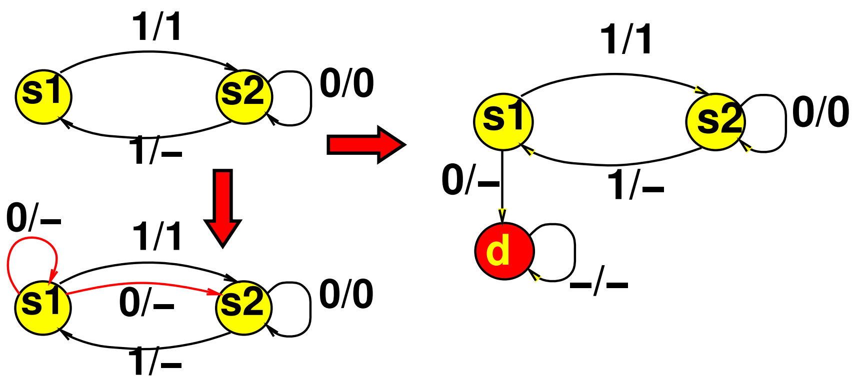
Next state and output functions have **don't cares**. However, for an **implementation**, δ and λ are functions, thus they are uniquely defined for each input and state combination.

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, δ and λ are relations, but of special type). (*We should make sure we want these as don't cares*). Such machines called **incompletely specified.**

Example:

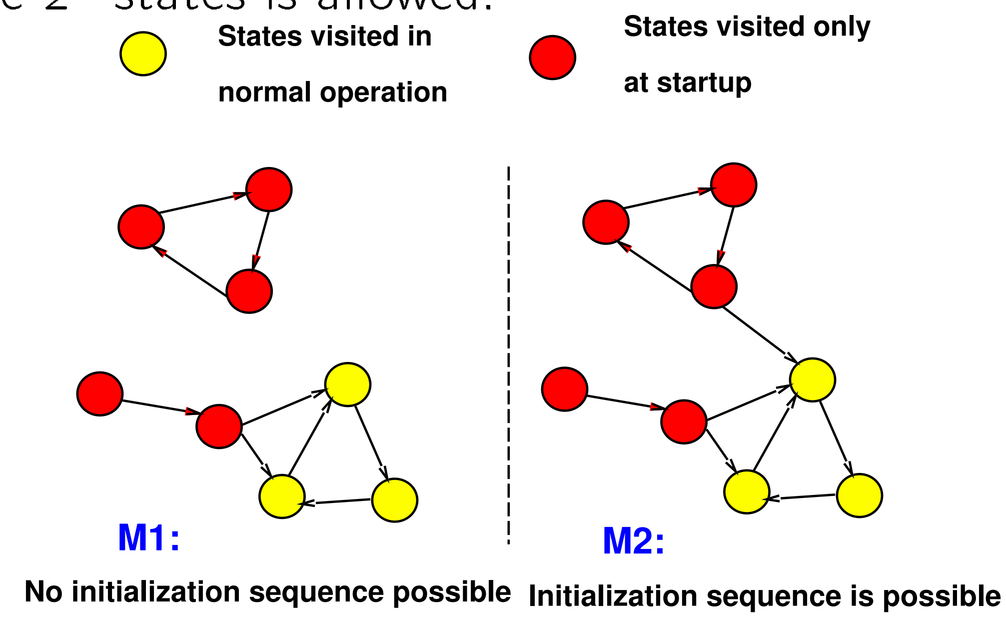


By adding a dummy state this can be converted to a machine with only the output incompletely specified. Could also make "error" as the output when transiting to the dummy state. Alternately (better), can interpret undefined next state as allowing **any next state**.

Initializing Sequences

Reference: [C. Pixley, TCAD Dec. 1992]

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 up, that any of the 2^n states is allowed.



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

Initializing Sequence: A sequence of input vectors that gets the machine to an equivalence class of known reset states.

May be implemented using a single reset signal.

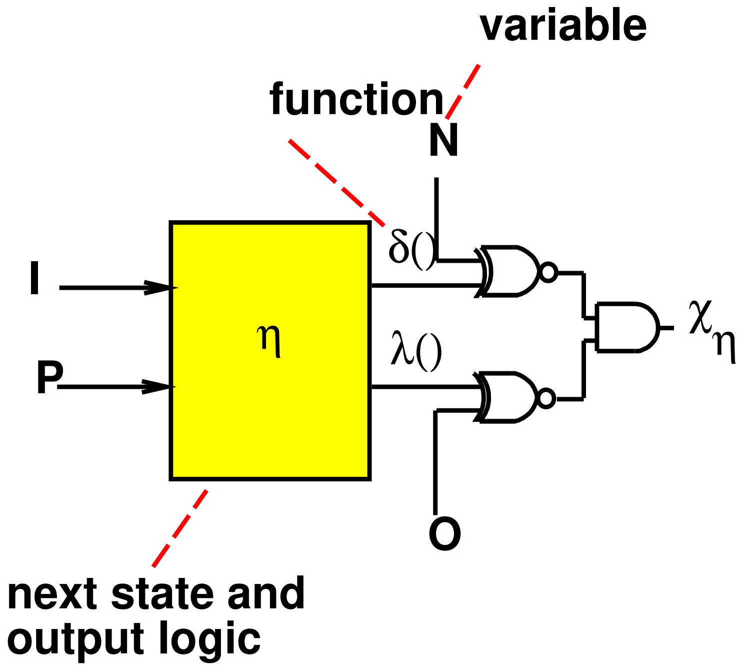
Pixley: *If an aligning sequence exists for each state pair, then an initializing sequence exists.*

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, $\chi_\eta(i, p, o, n)$.



$$\chi_\eta(i, p, o, n) = (\delta(i, p)\overline{\oplus}n) \wedge (\lambda(i.p)\overline{\oplus}o)$$

χ_η is the characteristic function of the transition and output relation.

χ_η may be represented in several ways, the ROBDD representation seems to be most useful.

FSM Extraction

Explicit/Semi-Implicit Extraction of all Transitions

Method 1:
Reference: [Devadas-Ma-Newton88]

Visit states starting from the reset states (in breadth-first-order).

```
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 till 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 enumerate all possible edges.)

Semi-Implicit Extraction:

- method is exponential in the number of states.

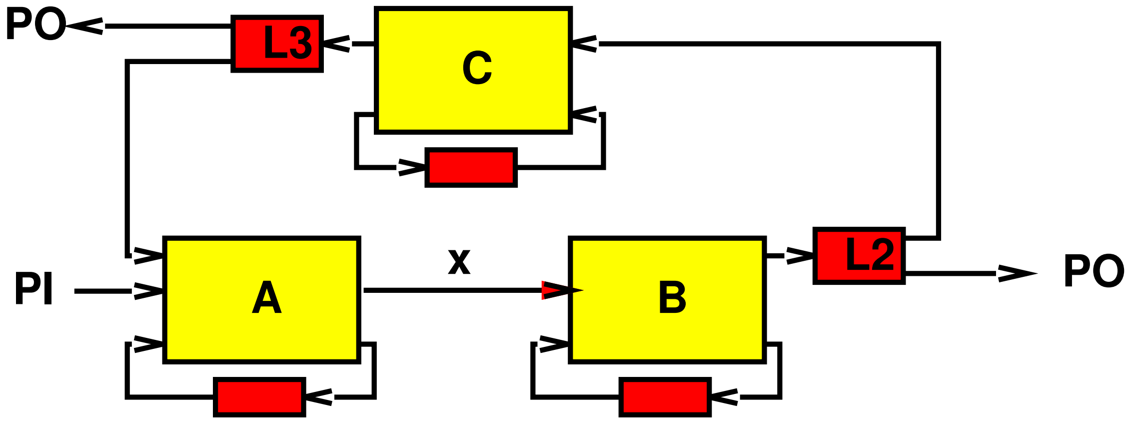
- May not be possible to represent χ_η in reasonable space as a STT.

Implicit Method

- Use BDD's to represent χ_η.

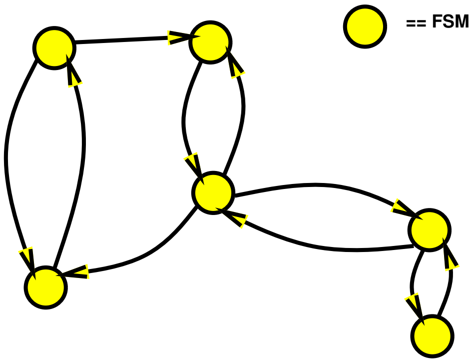
Interconnected FSMs - FSM Networks

- Natural way of describing complex systems (hierarchy, decomposition). Naturally extracted from HDL's with modules or sub-processes.



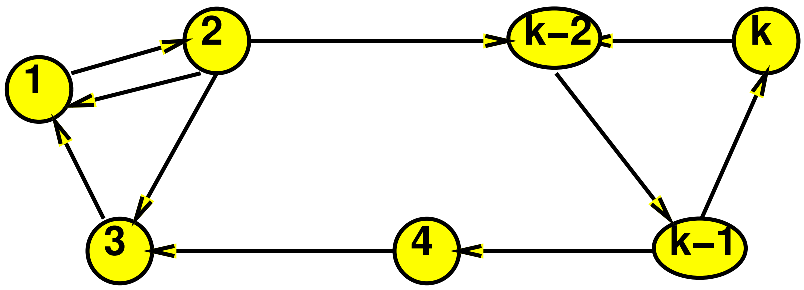
- Interconnected FSMs $\equiv$ Single product machine (similar to flattening in boolean circuits)

- Directed Graph - Each node an FSM. Arcs are variables used for communication.



Similar to Boolean network, possibly cyclic.

FSM Networks



Consider k **component** machines

$$M_i = (I_i, S_i, O_i, \delta_i, \lambda_i), \quad i \;=\; 1, ..., k$$

interconnected to form a **network** of FSMs (or **composite** machine or **decomposed** machine)

$$\mathcal{M}_{\mathcal{N}} = M_1 \times M_2 \times ... \times M_k \;=\; (I, S_N, O, \delta_N, \lambda_N).$$

The state set of $\mathcal{M}_{\mathcal{N}}$ is $S_N \;=\; S_1 \times S_2 \times ... \times S_K$, and δ_N and λ_N are the next state and output mappings induced by the properties of the component machines.

$\mathcal{M}_{\mathcal{N}}$ realizes (or defines) the **product** (**flattened** or **lumped** or **composed**) machine $M = (I, S, O, \delta, \lambda)$.

Using BDD's, the transition relation for the product machine is:

$$T(I, S, O, S') = \prod_{i=1}^{k} T_i(I_i, S_i, O_i, S'_i)$$

FSM Networks

If we had perfect optimization tools, product machine is best, however

- product machine is huge (state explosion problem).
- tools are heuristic (imperfect)

Tools needed for

- synthesis on 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.

Research:

- near-minimal collapsing of networks of FSMs (Wolf, 1991)
- Component minimization.
- VIS development.

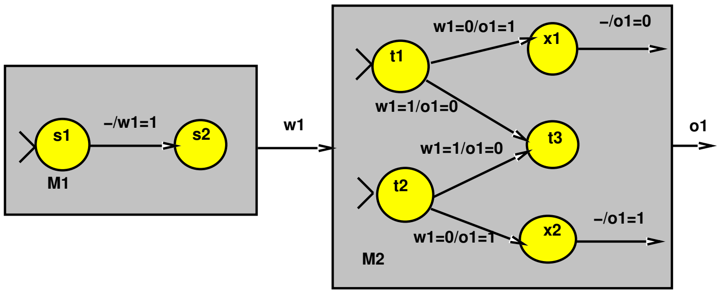
Near-Minimal Collapsing of FSM Networks

- Collapsing a network of completely specified minimized FSMs may create a huge product machine
- If the product machine contains many equivalent states, a small product FSM may be obtained after minimization. However intermediate machines may be too big.

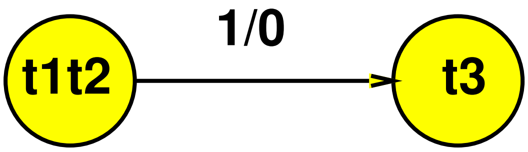
Is it sometimes possible to detect, while collapsing, most equivalence classes to produce directly a small product FSM [Wolf, 1991].

- Key observation: network of FSMs have many equivalences recognizable in a single cycle. If two states have the same outputs and go to the same next states, then equivalent.
- A network composed of state-minimized component FSM's can still have **equivalent product states** because the transitions which make states in a component distinguishable may not be allowed in the network (called *communication equivalence*)

Communication Equivalence



Example:
When $M1$ in state $s1$, $w1 = 1$ and $M2$ goes to state $t3$ (from $t1$ or $t2$) with output $o1 = 0$. Since $M1$ never outputs $w1 = 0$, product machine never goes into states $s2 \times x1$ or $s2 \times x2$. Since $x1$ and $x2$ make $t1$ and $t2$ non-equivalent, product states $s1 \times t1$ and $s1 \times t2$ are equivalent.

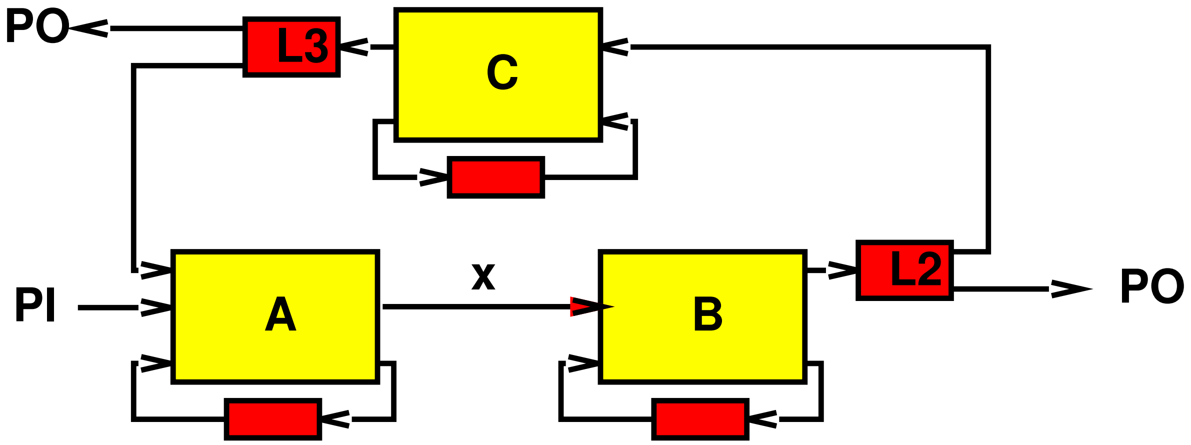


Procedure: Use communication equivalences to deduce other state equivalences by working backward one-cycle implications from known equivalences. If two states have the same outputs and go to the same next (equivalent) states then equivalent.

Result: Only undetected equivalences are those dependent on nets not visible as primary inputs or primary outputs.

Sequential Input Don't Cares

- Input don't care **vectors**

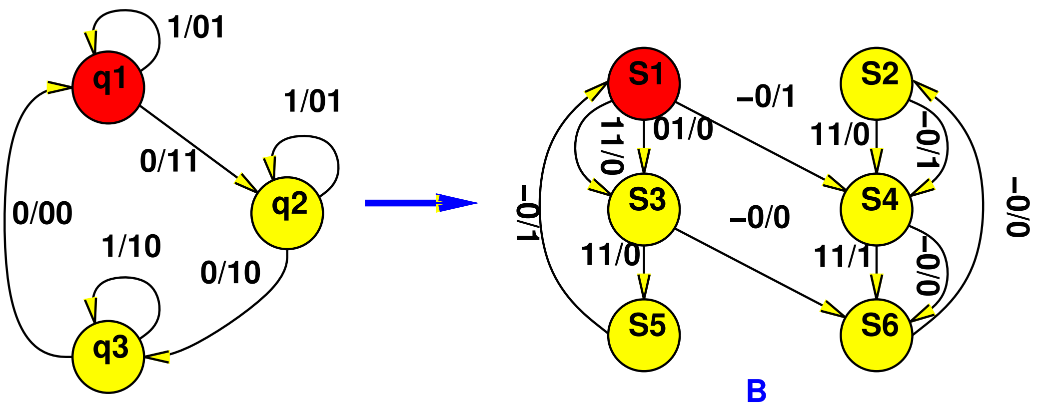


Machine A does not output a certain combination l at x, when B is in set of states S_l. The transitions of B from states in S_l under input l are therefore unspecified.

Exploited by state minimization and state encoding algorithms.

Sequential Input Don't Cares

- Input don't care **sequences of vectors**

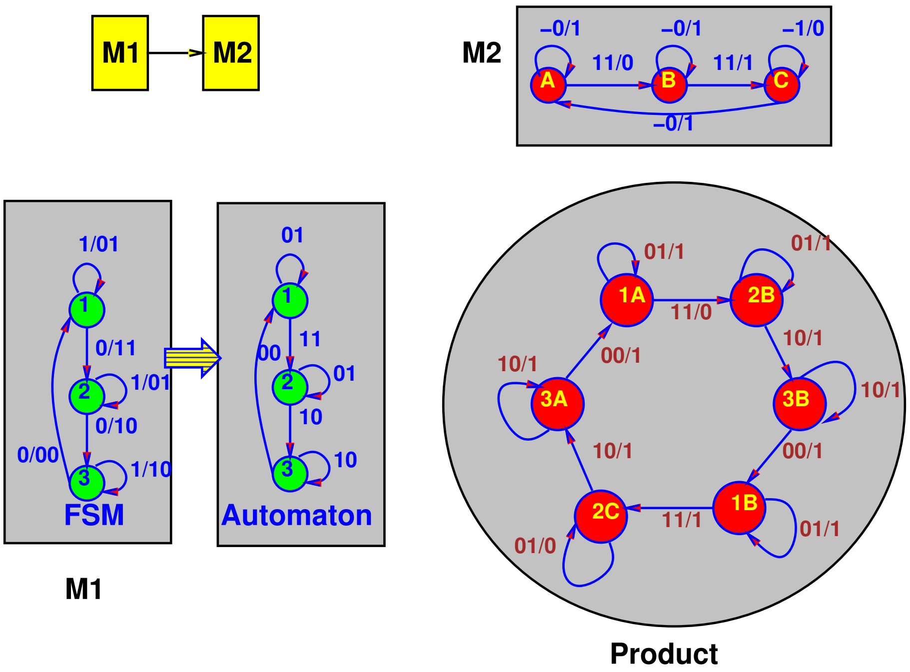


The sequence (11, 11) is a don't care.

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

However, note that can't simply remove states in B.

Kim and Newborn Procedure



1. In general, automaton may be nondeterministic. Then have to determinize it with "subset construction".

2. Note that 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 $A \to B$.

- Papers by Unger, Kim-Newborn, Devadas, Somenzi, Wang to exploit input don't care sequences for logical optimization.

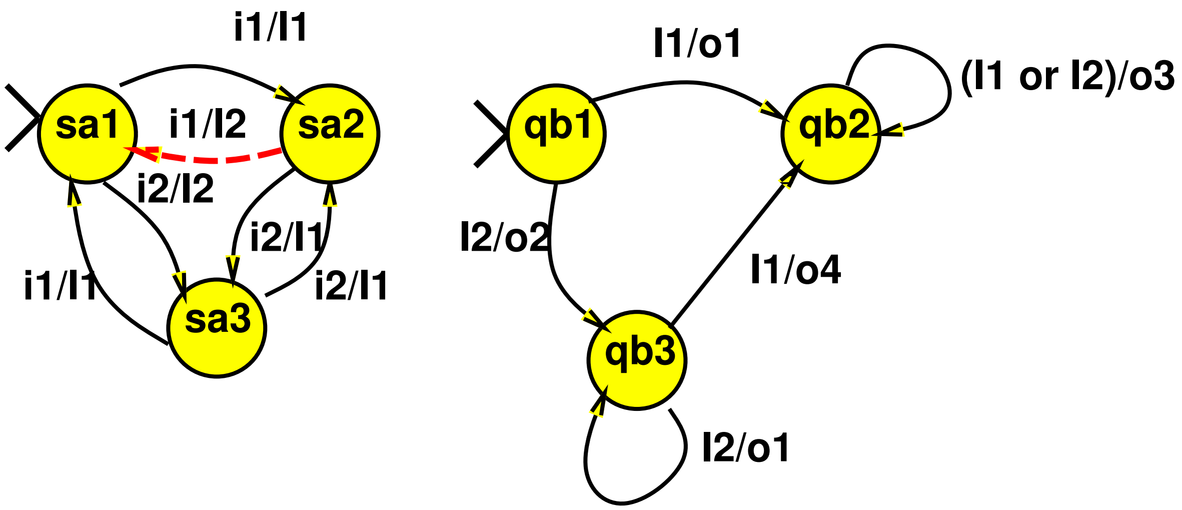
Sequential Output Don't Cares

- Output don't cares. Due to the constrained observability of the driving machine A.

i1	sa1	sa2	I1
i2	sa1	sa3	I2
i1	sa2	sa1	I2
i2	sa2	sa3	I1
i1	sa3	sa1	I1
i2	sa3	sa2	I1

I1	qb1	qb2	o1
I2	qb1	qb3	o2
I1	qb2	qb2	o3
I2	qb2	qb2	o3
I1	qb3	qb2	o4
I2	qb3	qb3	o1

A ⟶ **B**

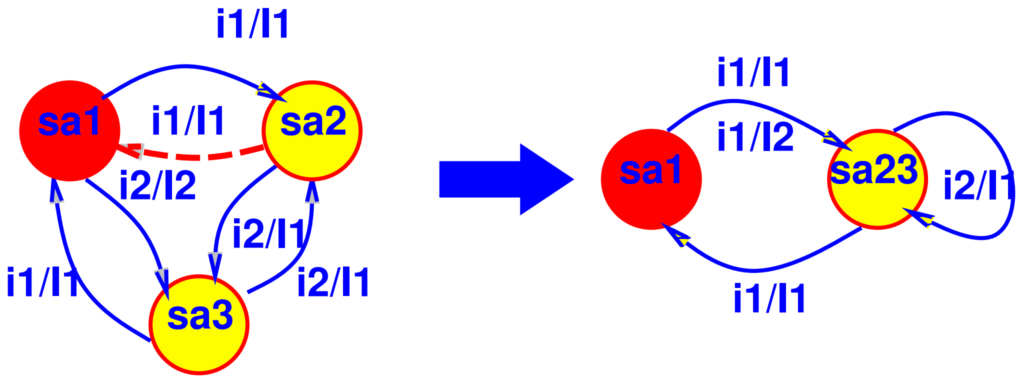


A feeds B. The third transition of A can output either $I1$ or $I2$, without changing the terminal behavior of the cascade $A \rightarrow B$. Called output expansion. (Starting state of A is $sa1$ and of B is $qb1$).

Sequential Output Don't Cares

- Output expansion produces a multiple-output FSM in which a transition outputs any element of a subset of symbolic or binary values (rather than any element of all possible values as in the case of an unspecified transition).

- 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

- *obtain the FSM description*
  - provided by the designer as a state table
  - extracted from netlist
  - derived from HDL description
    * obtain 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 needs to be done optimally. (NOVA, JEDI)
- minimization of a node in an FSM network.
- decomposition of FSM's and/or Collapsing
- sequential redundancy removal using ATPG techniques.