**Sequential logic implementation**

- Sequential circuits
- Primitive sequential elements
- Combinational logic
- Models for representing sequential circuits
- Finite state machines (Moore and Mealy)
- Representation of memory (states)
- Changes in state transitions
- Basic sequential circuits
- Shift registers
- Counters
- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

**Abstraction of state elements**

- Break circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic

**Forms of sequential logic**

- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple values or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a particular waveform – the clock)

**Finite state machine representations**

- States determined by possible values in sequential storage elements
- Transitions change of state
- Clock controls when state can change by controlling storage elements

**Example finite state machine diagram**

- Combination look from Introduction to course

**Can any sequential system be represented with a state diagram?**

- Shift register
  - Input value shown on transition arcs
  - Output values shown within state node
Counters are simple finite state machines

- Counters
  - process through well-defined sequence of states in response to enable
  - Many types of counters: binary, BCD, Gray-code
- 34-bit up-counter: 000, 001, 010, 011, 100, 101, 110, ..., 100, 111, 100, ...
- 34-bit down-counter: 111, 110, 101, 100, 010, 011, 001, 000, 011, ..., 100, 111, 110, ...

How do we turn a state diagram into logic?

- Counter
  - 1 flip-flops to hold state
  - Logic to compute next state
  - Don't use signal when flip-flop memory can change
  - Wait long enough for combinational logic to compute new value
- Don't wait too long as that is slow performance

FSM design procedure

- Start with counters
  - simple because output is just state
  - simple because no choice of next state based on input
- State diagram to state transition table
  - Tabular form of state diagram
  - Like a truth-table
- State encoding
  - encode representation of states
  - for counter it is simpler: just use value
- Implementation
  - flip-flop for each state bit
  - Combinational logic based on encoding

FSM design procedure: state diagram to encoded state transition table

- Tabular form of state diagram
  - Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters - just use value

Implementation

- 3 flip-flop for each state bit
  - Combinational logic based on encoding

Implementation (cont'd)

- Programmable logic building block for sequential logic
  - Macro-cell: FF + logic
  - PAL
  - D-flip
  - ECL or CPL capability like PAL (e.g., 8 product terms)
Another example
- Shift register
- input determines next state

<table>
<thead>
<tr>
<th>Dn</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>D1</th>
<th>D2</th>
<th>N1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

More complex counter example
- Complex counter
- repeats 5 states in sequence
- not a binary number representation
- Step 1: define the state transition diagram
- count sequence: 000, 010, 011, 101, 110
- Step 2: define the state transition table from the state transition diagram

Self-starting counters
- Start-up states
- at power-up, counter may be in an unused or invalid state
- designer must guarantee that it eventually enters a valid state
- Self-starting solution
- design counter so that initial states eventually transition to a valid state
- may limit exploitation of start cases

State machine model
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
  - function of current state and inputs (Moore machine)
  - function of current state only (Mealy machine)
- Outputs
  - function of current state and inputs (Moore machine)
  - function of current state only (Mealy machine)
State machine model (cont'd)
- States: S₁, S₂, ..., Sₖ
- Inputs: I₁, I₂, ..., Iₘ
- Outputs: O₁, O₂, ..., Oₙ
- Transition function: R(S, I)
- Output function: r₁(S) or rₙ(S, I)

Next State

Example: ant brain (Ward, MIT)
- Sensors: L and R antennae, L if touching wall
- Actuators: F - forward step, T/L/R - turn left/right slightly
- Goal: Avoid getting lost
- Strategy: Keep the wall on the right

Ant behavior
- Following a wall, moving towards left
- Following a wall, moving towards right
- Collision with wall, turning right
- Collision with wall, turning left
- LOST: Turn left until touching something

Designing an ant brain
- State diagram
- Using symbolic states and outputs
- Transition truth table
- Implementation of state and output functions using combinational logic: 2-level logic (NAND/AND)
- Using multiple logic levels: next state and output functions can be optimized together
Synthesis

- 5 states: at least 3 state variables required (X, Y, Z)
- State assignment (in this case, arbitrarily chosen)

<table>
<thead>
<tr>
<th>State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>110</td>
</tr>
</tbody>
</table>

A non-redundant representation for the synthesized function:

\[ F = x + y + z \]

Synthesis of next state and output functions

<table>
<thead>
<tr>
<th>State Inputs</th>
<th>Next State Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0</td>
<td>000 1</td>
</tr>
<tr>
<td>001 0</td>
<td>001 1</td>
</tr>
<tr>
<td>010 1</td>
<td>010 0</td>
</tr>
<tr>
<td>011 1</td>
<td>011 0</td>
</tr>
<tr>
<td>100 0</td>
<td>100 1</td>
</tr>
<tr>
<td>101 1</td>
<td>101 0</td>
</tr>
</tbody>
</table>

Don't care in FSM synthesis

- What happens to the "unused" states (011, 110, 111)?
- They were eliminated as don't cares to minimize the logic.

Circuit implementation

- Outputs are a function of the current state only - Moore machine

State minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two states are equivalent if they are impossible to distinguish from the outputs of the FSM, i.e., for any input sequence the outputs are the same
- Two conditions for two states to be equivalent:
  1) Output must be the same in both states
  2) Must transition to equivalent states for all input combinations

Ant brain revisited

- Any equivalent states?
New improved brain

- Merge equivalent A and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3

Mealy vs. Moore machines

- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our new brain is a Moore machine
  - output does not react immediately to input change
  - outputs have immediate reaction to inputs
  - as inputs change, so does next state, does not commit until decision event

Specifying outputs for a Moore machine

- Output is only function of state
  - specify in state table in state diagram
- Example: sequence detector for 01 or 10

Specifying outputs for a Mealy machine

- Output is function of state and inputs
  - specify output on transition and between states
- Example: sequence detector for 01 or 10

Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - different outputs on arcs (n^2) rather than states (n)
- Moore machines are easier to use
  - outputs change at clock edge (always one cycle later)
  - in Moore machines, input change can cause output change as soon as logic is done - a big problem when two machines are interconnected - asynchronous feedback
- Mealy machines need faster to inputs
  - read in same cycle - don’t need to wait for clock
  - in Moore machines, more logic may be necessary to decode state into outputs - more gate delays slower
Mealy and Moore examples
- Recognize A, B = 0,1
- Mealy or Moore?

Mealy and Moore examples (cont’d)
- Recognize A, B = 1,0 then 0,1
- Mealy or Moore?

Registered Mealy machine (really Moore)
- Synchronous (or registered) Mealy machine
- registered state AND outputs
- avoids glibly outputs
- easy to implement in PLDs
- Moore machine with no output encoding
- outputs computed on transition to next state rather than after entering
- view outputs as expanded state vector

Example: vending machine
- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change

Example: vending machine (cont’d)
- Suitable abstract representation
  - tabular typical input sequences:
    - 3 inputs:
      - nickel, dime
      - dime, nickel
      - two dimes
  - state diagram:
    - inputs: N, D, reset
    - output: open, close
  - assumptions:
    - assume input always present
    - each state has a self-loop
    - for N = 0 or D = 0 (no coin)
Example: vending machine (cont'd)

- Unique state encoding

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
<th>Next state output</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>0</td>
<td>q1</td>
</tr>
<tr>
<td>q1</td>
<td>1</td>
<td>q2</td>
</tr>
<tr>
<td>q2</td>
<td>0</td>
<td>q3</td>
</tr>
<tr>
<td>q2</td>
<td>1</td>
<td>q4</td>
</tr>
<tr>
<td>q3</td>
<td>0</td>
<td>q4</td>
</tr>
<tr>
<td>q3</td>
<td>1</td>
<td>q1</td>
</tr>
<tr>
<td>q4</td>
<td>0</td>
<td>q2</td>
</tr>
<tr>
<td>q4</td>
<td>1</td>
<td>q1</td>
</tr>
</tbody>
</table>

Example: vending machine (cont'd)

- Mapping to logic

Example: vending machine (cont'd)

- One-hot encoding

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
<th>Next state output</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>0</td>
<td>q1</td>
</tr>
<tr>
<td>q1</td>
<td>1</td>
<td>q2</td>
</tr>
<tr>
<td>q2</td>
<td>0</td>
<td>q3</td>
</tr>
<tr>
<td>q2</td>
<td>1</td>
<td>q4</td>
</tr>
<tr>
<td>q3</td>
<td>0</td>
<td>q2</td>
</tr>
<tr>
<td>q3</td>
<td>1</td>
<td>q1</td>
</tr>
<tr>
<td>q4</td>
<td>0</td>
<td>q2</td>
</tr>
<tr>
<td>q4</td>
<td>1</td>
<td>q1</td>
</tr>
</tbody>
</table>

Example: traffic light controller

- A basic Highway is controlled by a simple traffic controller
- Controls the presence of cars waiting on the freeway
- When a car is present, lights remain green in the highway direction
- When vehicles are present, highway lights go from green to yellow to red, allowing the freeway lights to become green
- Highway lights turn green only when a car is detected but never longer than a one-second interval
- When there are no cars, green lights transition from green to yellow to red, allowing highway traffic to return to green
- Cars in the freeway traffic are waiting, and highway gets at least a one-second interval green
- Assume traffic is a finite type of traffic that generates:
  - a short time pulse (T1): 1 sec
  - a long-time pulse (T2): 2 sec
  - a response to a red (T3) signal
  - T5 is to be used for turning yellow, red, and green lights

Example: traffic light controller (cont')

- Highway/Intersection

<table>
<thead>
<tr>
<th>Freeway</th>
<th>Car sensor</th>
<th>Highway</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: traffic light controller (cont')

1. Transition of inputs and outputs

<table>
<thead>
<tr>
<th>Input Description</th>
<th>Output Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red phase if valid</td>
<td>Red phase if valid</td>
</tr>
<tr>
<td>Green phase if valid</td>
<td>Green phase if valid</td>
</tr>
<tr>
<td>Yellow phase if valid</td>
<td>Yellow phase if valid</td>
</tr>
<tr>
<td>Red phase if invalid</td>
<td>Red phase if invalid</td>
</tr>
<tr>
<td>Green phase if invalid</td>
<td>Green phase if invalid</td>
</tr>
<tr>
<td>Yellow phase if invalid</td>
<td>Yellow phase if invalid</td>
</tr>
</tbody>
</table>

2. Validity of unique states - some light configurations imply others

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Red phase if valid</td>
</tr>
<tr>
<td>S2</td>
<td>Green phase if valid</td>
</tr>
<tr>
<td>S3</td>
<td>Yellow phase if valid</td>
</tr>
<tr>
<td>S4</td>
<td>Red phase if invalid</td>
</tr>
<tr>
<td>S5</td>
<td>Green phase if invalid</td>
</tr>
<tr>
<td>S6</td>
<td>Yellow phase if invalid</td>
</tr>
</tbody>
</table>

Example: traffic light controller (cont')

State diagram

- S1: Red
- S2: Green
- S3: Yellow
- S4: Red
- S5: Green
- S6: Yellow

Logic for different state assignments

<table>
<thead>
<tr>
<th>State Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Red</td>
</tr>
<tr>
<td>S2: Green</td>
</tr>
<tr>
<td>S3: Yellow</td>
</tr>
<tr>
<td>S4: Red</td>
</tr>
<tr>
<td>S5: Green</td>
</tr>
<tr>
<td>S6: Yellow</td>
</tr>
</tbody>
</table>

Vending machine example (PLD mapping)

Vending machine (cont'd)

- CFER = Q1: creates a combinational delay after Q1 and Q2 change
- This can be corrected by redesigning, i.e., move flip-flops and logic through each other to improve delay
- CFER = reset(Q1 + D + Q1)(Q1'H + Q1 + Q1D + Q1'H) = reset(Q1)(Q1'H + Q1 + Q1D + Q1'H)
- Implementation flow looks like a synchronous Mealy machine
- It is common for programmable devices to have F at end of logic
Algorithmic approach to state minimization

- **Goal**: Identify and combine states that have equivalent behavior
- **Equivalent states**:
  - same output
  - for all input combinations, states transition to same or equivalent states
- **Algorithm sketch**:
  1. Place all states in one set
  2. Initially partition set based on output behavior
  3. Successively partition resulting subsets based on next state transitions
  4. Repeat (3) until no further partitioning is required
  - States left in the same set are equivalent
  - Polynomial time procedure

State minimization example

- Sequence detector for 010 or 10

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>X=1</th>
<th>Output X=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S3</td>
<td>S4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S1 or S2</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S3 or S4</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S0 or S1</td>
<td>S6</td>
<td>S7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2 or S3</td>
<td>S7</td>
<td>S8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S4 or S5</td>
<td>S8</td>
<td>S9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S6 or S7</td>
<td>S9</td>
<td>S10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S8 or S9</td>
<td>S10</td>
<td>S11</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Minimized FSM

- State minimal sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Time Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>X=0</th>
<th>Output X=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>S3</td>
<td>S4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S1 or S2</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S3 or S4</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2 or S3</td>
<td>S6</td>
<td>S7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S4 or S5</td>
<td>S7</td>
<td>S8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S6 or S7</td>
<td>S8</td>
<td>S9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S8 or S9</td>
<td>S9</td>
<td>S10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
More complex state minimization

- Multiple input example

<table>
<thead>
<tr>
<th>present</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
</tr>
<tr>
<td>output</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Minimized FSM

- Implication chart method

- Cross out incompatible states based on outputs
- Then cross out more if implied states are already crossed out

Minimizing incompletely specified FSMs

- Equivalence of states is true if machine is fully specified
- But it is not true if don't cares are present

<table>
<thead>
<tr>
<th>state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>-1</td>
</tr>
</tbody>
</table>

No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states.

Minimizing states may not yield best circuit

- Example: edge detector - outputs 1 when both input changes from 0 to 1

<table>
<thead>
<tr>
<th>X</th>
<th>Q0</th>
<th>Q1</th>
<th>Q0'</th>
<th>Q1'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Q1' = X Q0 = X Q0

Q1 = X Q0 Q1

Another implementation of edge detector

- "Ad-hoc" solution - not minimal but cheap and fast

State assignment

- Choose bit vectors to assign to each "symbolic" state
- With n state bits for m states there are 2^m (2^n - m)
  \log n < m < 2^n

- 2^n codes possible for l state, 2^n-l for 2m, 2^n-2 for 3m, ...
- Large number even for small values of n and m
- Intractable for state machines of any size
- Heuristics are necessary for practical solutions
- Optimize some metric for the combinational logic
  - Size (amount of logic and number of FEs)
  - Speed (depth of logic and fanout)
  - Dependencies (complementation)
**State assignment strategies**

- Possible strategies
  - Sequential - just number states as they appear in the state table
  - Random - pick random states
  - 1-hot - use as many state bits as there are states (bit[i] = state)
  - output - use outputs to help encode states
- Heuristic rules of thumb that seem to work in most cases
- No guarantee of optimality - another interwoven problem

**One-hot state assignment**

- Simple
  - easy to encode
  - easy to debug
- Small logic functions
  - each state function requires only predecessor state bits as input
  - good for programmable devices
- lots of flip-flops readily available
- simple functions with small support (signals dependent upon)
- impractical for large machines
  - too many states require too many flip-flops
  - decompose 16 bits into smaller pieces that can be one-hot encoded
- Many digit variations to one-hot
  - one-hot + all 0

**Heuristics for state assignment**

- Adjacent codes to states that share a common next state
  - group 1's in next state map
    - $i$ = $j$ = $k$
    - $i$, $j$, $k$ = 1
  - Adjacent codes to states that share a common ancestor state
    - group 1's in next state map
    - $i$ = $j$ = $k$
    - $i$, $j$, $k$ = 0
  - Adjacent codes to states that have a common output behavior
    - group 1's in output map
    - $i$ = $j$ = $k$
    - $i$, $j$, $k$ = 0

**General approach to heuristic state assignment**

- All current methods are variants of this
  - 1) determine which states "attract" each other (weighted pairs)
  - 2) generate constraints on codes (which should be in the same code)
  - 3) place codes on Boolean cube so as to minimize constraints satisfied (weighted sum)
- Different methods make sense depending on whether we are optimizing for logic implementation
- Can't consider all possible embeddings of state clusters in Boolean cube
- heuristics for coding embeddings
  - to prune search for best embedding
  - to expand code (more state bits) to satisfy more constraints

**Output-based encoding**

- Reuse outputs as state bits - use outputs to help distinguish states
- useful in hardware implementations
- simple in hardware implementations
- good in software implementations
- good in both hardware and software implementations

<table>
<thead>
<tr>
<th>State</th>
<th>Next State</th>
<th>Present State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NG</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NG</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Output patterns are similar to the above ones, but not used in this example. One-hot encoding generates more patterns than the above, but this is not used in this example. One-hot encoding is more efficient than this, but it is not used in this example.

**Current state assignment approaches**

- For tight encodings, using close to the minimum number of state bits
  - logic optimizers seem to be adequate (even more so for hardware)
- heuristic approaches are not even close to optimality
- used in custom chip design
- easy for small data machines
- generates simple equations with easy to estimate complexity
- common in TCGs and other programmable logic

- One-hot encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic

- Output-based encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic

- Current state assignment approaches
  - One-hot encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic

- Output-based encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic

- Current state assignment approaches
  - One-hot encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic

- Output-based encoding
  - easy for small data machines
  - generates simple equations with easy to estimate complexity
  - common in TCGs and other programmable logic