Lecture 22

◆ Logistics
  ■ HW8 due Monday (6/2), HW9 due Friday (6/6)
  ■ Ant extra credit due 6/6
  ■ Take home extra credit final handed out 6/6
  ■ Final exam 6/9 8:30am
  ■ Review?

◆ Last lecture
  ■ Simplification

◆ Today
  ■ State encoding
    ◦ One-hot encoding
    ◦ Output encoding

Example: A vending machine

◆ 15 cents for a cup of coffee
◆ Doesn’t take pennies or quarters
◆ Doesn’t provide any change

■ FSM-design procedure
  1. State diagram
  2. state-transition table
  3. State minimization
  4. State encoding
  5. Next-state logic minimization
  6. Implement the design
A vending machine: State minimization

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>D N</td>
<td></td>
<td>D N</td>
</tr>
<tr>
<td>0¢ 0¢</td>
<td>0 0</td>
<td>0¢ 0</td>
</tr>
<tr>
<td>0¢ 1 10¢</td>
<td>0 1</td>
<td>5¢ 0</td>
</tr>
<tr>
<td>0¢ 1 15¢</td>
<td>0 1</td>
<td>5¢ 0</td>
</tr>
<tr>
<td>1 1 1 10¢</td>
<td>1 0</td>
<td>10¢ 0</td>
</tr>
<tr>
<td>1 1 1 15¢</td>
<td>1 0</td>
<td>15¢ 0</td>
</tr>
<tr>
<td>1 1 1 open</td>
<td>1 1</td>
<td>15¢ 1</td>
</tr>
</tbody>
</table>

symbolic state table

A vending machine: State encoding

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state</th>
<th>output open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Q0 D N D1</td>
<td></td>
<td>D0</td>
<td>open</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0</td>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1 0</td>
<td>1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1</td>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 -</td>
<td>1 1</td>
<td>1 - -</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1 0</td>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 -</td>
<td>1 1</td>
<td>1 - -</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1</td>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

CSE370, Lecture 22
A vending machine: Logic minimization

\[
D1 = Q1 + D + Q0 N \\
D0 = Q0' N + Q0 N' + Q1 N + Q1 D \\
OPEN = Q1 Q0
\]

A vending machine: Implementation
State encoding

- Assume \( n \) state bits and \( m \) states
  - \( 2^n! / (2^n - m)! \) possible encodings
    - Example: 3 state bits, 4 states, 1680 possible state assignments

- Want to pick state encoding strategy that results in optimizing your criteria
  - FSM size (amount of logic and number of FFs)
  - FSM speed (depth of logic and fan-in/fan-out)
  - FSM ease of design or debugging

State-encoding strategies

- No guarantee of optimality
  - An intractable problem

- Most common strategies
  - Binary (sequential) – number states as in the state table
  - Random – computer tries random encodings
  - Heuristic – rules of thumb that seem to work well
    - e.g. Gray-code – try to give adjacent states (states with an arc between them) codes that differ in only one bit position
  - One-hot – use as many state bits as there are states
  - Output – use outputs to help encode states
  - Hybrid – mix of a few different ones (e.g. One-hot + heuristic)
One-hot encoding

- **One-hot:** Encode n states using n flip-flops
  - Assign a single “1” for each state
    - Example: 0001, 0010, 0100, 1000
  - Propagate a single “1” from one flip-flop to the next
    - All other flip-flop outputs are “0”

- **The inverse: One-cold encoding**
  - Assign a single “0” for each state
    - Example: 1110, 1101, 1011, 0111
  - Propagate a single “0” from one flip-flop to the next
    - All other flip-flop outputs are “1”

- **“almost one-hot” encoding (modified one-hot encoding)**
  - Use no-hot (000…0) for the initial (reset state)
  - Assumes you never revisit the reset state till reset again.

One-hot encoding (con’t)

- Often the best/convenient approach for FPGAs
  - FPGAs have many flip-flops

- Draw FSM directly from the state diagram
  - + One product term per incoming arc
  - - Complex state diagram ⇒ complex design
  - - Many states ⇒ many flip flops
Example: A vending machine ... again

- 15 cents for a cup of coffee
- Doesn’t take pennies or quarters
- Doesn’t provide any change

- FSM-design procedure
  1. State diagram
  2. State-transition table
  3. State minimization
  4. State encoding
  5. Next-state logic minimization
  6. Implement the design

One-hot encoded transition table

<table>
<thead>
<tr>
<th>present state inputs $Q_3Q_2Q_1Q_0$, D, N</th>
<th>next state $D_2D_1D_0$</th>
<th>output open</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 D N</td>
<td>0 0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 1 D N</td>
<td>0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 1 D N</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0 1 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 0 D N</td>
<td>0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0 D N</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 0 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 1 1 0 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>0 1 0 0 D N</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0 D N</td>
<td>1 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 0 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 1 1 0 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 0 0 0 D N</td>
<td>- - 1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0 1 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 0 1 1 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 1 0 1 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>1 1 1 1 D N</td>
<td>- - - - -</td>
<td>-</td>
</tr>
</tbody>
</table>

CSE370, Lecture 22
Designing from the state diagram

\[ D_0 = Q_0D'N' \]
\[ D_1 = Q_0N + Q_1D'N' \]
\[ D_2 = Q_0D + Q_1N + Q_2D'N' \]
\[ D_3 = Q_1D + Q_2D + Q_2N + Q_3 \]
\[ OPEN = Q_3 \]

Output encoding

- Reuse outputs as state bits
  - Why create new functions when you can use outputs?
  - Bits from state assignments are the outputs for that state
  - Take outputs directly from the flip-flops

- ad hoc - no tools
  - Yields small circuits for most FSMs
Vending machine
--- already in output encoding form

D' N'

D = Q_0D'N'
D_1 = Q_0N + Q_1D'N'
D_2 = Q_0D + Q_1N + Q_2D'N'
D_3 = Q_1D + Q_2D + Q_2N + Q_3
OPEN = Q_3

Example: Digital combination lock

An output-encoded FSM
- Punch in 3 values in sequence and the door opens
- If there is an error the lock must be reset
- After the door opens the lock must be reset
- Inputs: sequence of number values, reset
- Outputs: door open/close
Separate data path and control

- **Design datapath first**
  - After the state diagram
  - Before the state encoding

- **Control has 2 outputs**
  - Mux control to datapath
  - Lock open/closed

Draw the state diagram
Design the datapath

- Choose simple control
  - 3-wire mux for datapath
    - Control is 001, 010, 100
  - Open/closed bit for lock state
    - Control is 0/1

Output encode the FSM

- FSM outputs
  - Mux control is 100, 010, 001
  - Lock control is 0/1

- State are: S0, S1, S2, S3, or ERR
  - Can use 3, 4, or 5 bits to encode
  - Have 4 outputs, so choose 4 bits
    - Encode mux control and lock control in state bits
    - Lock control is first bit, mux control is last 3 bits
      - S0 = 0001 (lock closed, mux first code)
      - S1 = 0010 (lock closed, mux second code)
      - S2 = 0100 (lock closed, mux third code)
      - S3 = 1000 (lock open)
      - ERR = 0000 (error, lock closed)
FSM has 4 state bits and 2 inputs...

- Output encoded!
  - Outputs and state bits are the same

- How do we minimize the logic?
  - FSM has 4 state bits and 2 inputs (equal, new)
  - 6-variable kmap?

- Notice the state assignment is close to one-hot
  - ERR state (0000) is only deviation
  - Is there a clever design we can use?

Encode 4 state bits

\[ D_0 = Q_0N' \]
\[ D_1 = Q_0EN + Q_1N' \]
\[ D_2 = Q_1EN + Q_2N' \]
\[ D_3 = Q_2EN + Q_3 \]

Can we encode the ERR state with reset/preset of flipflops?
Answer: Yes!

D₀ = Q₀N'
D₁ = Q₀EN + Q₁N'
D₂ = Q₁EN + Q₂N'
D₃ = Q₂EN + Q₃

Preset₀ = start
Preset₁,₂,₃ = 0
Reset₀ = start'(E'N + (Q₀+Q₁+Q₂+Q₃)')
Reset₁,₂,₃ = start + (E'N + (Q₀+Q₁+Q₂+Q₃)')
FSM design

- **FSM-design procedure**
  1. State diagram
  2. State-transition table
  3. State minimization
  4. State encoding
  5. Next-state logic minimization
  6. Implement the design