General Sequential Design

So far we have, we have looked at basic latches, FFs and common sequential building blocks.

All of these can be represented by a general block diagram:

Describing Sequential Systems

- So far we have used Truth Tables to describe sequential systems
- Can also use Bubble Diagrams and Algorithmic State Machine Charts (ASM) to describe a sequential system.
- Another name for a sequential system is a Finite State Machine (FSM).
- A sequential system with N flip-Flop has $2^N$ possible states, so the number of possible states is FINITE.
**DFF as a Finite State Machine**
A DFF is a finite state machine with two possible states. Let's call these states S0 and S1. (*state enumeration*).

Furthermore, let's say when the Q output = '0', then we are in State S0, and that when Q output = '1', we are in State S1. This is called the *State Encoding*.

![Bubble Diagram](image)

**Algorithmic State Machine Chart for DFF**

![Algorithmic State Machine Chart](image)

**Algorithmic State Chart (ASM)**

- An ASM chart can be used to describe FSM behavior.

  Only three action signals can appear within an ASM chart:
  - **State box.** Each box represents a state. Outputs within a state box is an UNCONDITIONAL output (always asserted in this state).
  - **Decision box.** A condition in this box will decide next state condition.
  - **Conditional output box.** If present, will always follow a decision box; output within it is conditional.
Algorithmic State Machine Chart for JKFF

Finite State Machine Implementation
Given an Algorithmic State Machine chart that describes a Finite State Machine, how do we implement it????!
Step #1: Decide on the State Encoding (how many Flip Flips do I use and how what should the FF outputs be for EACH state). The problem definition may decide the state encoding for you.
Step #2: Decide what kind of FFs to use! (We will always use DFFs in this class, but you could use JKFFs or TFFs if you wanted to).
Step #3: Write the State Transition Table.
Step #4: Write the FF input equations, and general output equations from the state transition table.

Problem Definition
Design a Modulo three counter. The count sequence is: “00” → “01” → “10” → “00” → “01” → “10”, etc.
There is an "en" input that should control counting (count when en=1, hold value when en=0). Assume ACLR line used to reset counter to "00".
How many states do we need? Well, we have three unique output values, so let's go with three states.
### State Transition Table

State transition table shows next state, output values for present state, input values.

<table>
<thead>
<tr>
<th>Inputs (EN)</th>
<th>Present State</th>
<th>Next State</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>S2</td>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>S0</td>
<td>S1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S0</td>
<td>10</td>
</tr>
</tbody>
</table>

### Decisions

- **State encoding** - will be based on number of FFs we use.
  - Three states means the minimum number of FFs we can use two FFs ($\log_2(3) = 2$).
- If we use two FFs, then could pick a state encodings like:
  - S0: 00, S1: 01, S2: 10 (binary counting order)
  - S0: 01, S1: 01, S2: 11 (gray code - may result in less combinational logic)
- Could also use 1 FF per state (3 FFs) and use one hot encoding
  - S0: 001, S1: 010, S2: 100 (may result in less combinational logic)
Decisions (cont.)

- What type of FF to use?
- DFF - most common type, always available in programmable logic
- JKFF - sometimes available, will usually result in less combinational logic (more complex FF means less combinational logic external to FF)

Let's use two FFs with state encoding S0=00, S1=01, S2=10.

Let's use DFFs.

New State Transition Table

Modify State Transition table to show what FF inputs need to be in order to get to that state. Also, use actual state encodings

<table>
<thead>
<tr>
<th>Inputs (EN)</th>
<th>Present State (Q1Q0)</th>
<th>Next State (Q1Q0)</th>
<th>D1 D0 Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

For DFFs, D inputs are simply equal to next state!!!

D-input Equations, Y equations

Unoptimized equations:

\[ D_0 = EN' Q_1' Q_0 + EN Q_1' Q_0' \]
\[ D_1 = EN' Q_1 Q_0' + EN Q_1' Q_0 \]

\[ Y_0 = Q_0 \]
\[ Y_1 = Q_1 \]

The output Y is simply the DFF outputs! Here is one case where state encoding is affected by problem definition (does not make much sense to use a different state encoding, even though we could do it).
What if we used JKFFs?

Need to change State Transition table to reflect JK input values.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State (Q1Q0)</th>
<th>Next State (Q1Q0)*</th>
<th>J1</th>
<th>K1</th>
<th>J0</th>
<th>K0</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>00</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>10</td>
</tr>
</tbody>
</table>

JK FF Q transitions: 0→0 (J=0, K=X); 0→1 (J=1, K=X); 1→1 (J=X, K=0); 1→0 (J=X, K=1);

JK Input Equations, Output Equations

Unoptimized equations

\[ J_0 = EN \overline{Q}_1 Q_0 \]
\[ K_0 = EN \overline{Q}_1 Q_0 \]
\[ J_1 = EN Q_1 Q_0 \]
\[ K_1 = EN Q_1 \overline{Q}_0 \]

\[ Y_0 = Q_0 \]
\[ Y_1 = Q_1 \]

Using JK FFs will mean simpler external optimized combinational logic because FFs are more complex (provide more functionality).
**JK FF Implementation**

\[ J_0 = EN \bar{Q}_1' \bar{Q}_0' \]
\[ K_0 = EN Q_1' \bar{Q}_0' \]
\[ J_1 = EN Q_1' \bar{Q}_0 \]
\[ K_1 = EN Q_1 Q_0' \]

ACLIR input to JKFFs not shown.

---

**3 DFFs and One Hot Encoding**

State encoding: \( S_0 = 001, S_1 = 010, S_2 = 100 \)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State ( (Q_2Q_1Q_0) )</th>
<th>Next State ( (Q_2Q_1Q_0) )</th>
<th>EN</th>
<th>D2D1D0</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>001</td>
<td>0</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>010</td>
<td>0</td>
<td>010</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>010</td>
<td>1</td>
<td>010</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>001</td>
<td>1</td>
<td>001</td>
<td>10</td>
</tr>
</tbody>
</table>

---

**DFF input equations, Output Equations**

\[ D_0 = EN'Q_0 + ENQ_2 \]
\[ D_1 = EN'Q_1 + ENQ_0 \]
\[ D_2 = EN'Q_2 + ENQ_1 \]

\[ Y_0 = EN'Q_1 + EN Q_1 = Q_1 \]
\[ Y_1 = EN'Q_2 + EN Q_2 = Q_2 \]

In equations, because a FF \( Q \) will only be ‘1’ in a single state, do not have to include all FFs to define state!!

\((Q_2'Q_1'Q_0 = Q_0, Q_2'Q_1Q_0' = Q_1, Q_2Q_1'Q_0' = Q_2)\)

This is one of the advantages of one-hot encoding!
Generic Next State Equations

Generic next state equations can be written directly from the ASM chart as an alternative to the Transition table.

\[ S^* = (\text{conditions to remain in this state}) \times (\text{conditions to enter state}) \]

From ASM chart of modulo three counter:

\[
\begin{align*}
S_0^* &= EN'S_0 + EN'S_2 \\
S_1^* &= EN'S_1 + EN'S_0 \\
S_2^* &= EN'S_2 + EN'S_1
\end{align*}
\]

If One hot encoding and DFFs are used, then Generic Next State equations ARE the specific next State Equations!!

\[
\begin{align*}
D_0 &= EN'Q_0 + EN'Q_2 \\
D_1 &= EN'Q_1 + EN'Q_0 \\
D_2 &= EN'Q_2 + EN'Q_1
\end{align*}
\]