Midterm 2 - Review
LATCH

Storage

Bi-stability

Latches
Sequential Circuits

• The output of a Combinatorial Circuit depends only on the current inputs

• The output of a Sequential Circuit can remember something about the past
Bi-Stability = Key to Memory

There are 2 stable states - a bi-stable circuit...

This is a stable state - it will sit like this forever

This is also a stable state - it will sit like this forever
SR Latch - A Bi-Stable Circuit

This is a stable state - it will sit like this forever

This is also a stable state - it will sit like this forever
### SR Latch Transition Table

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Diagram:**
- The SR latch has two inputs, S (set) and R (reset), and two outputs, Q and Q'.
- The truth table shows the state transitions for different combinations of S and R.
- The output Q+ indicates the next state, with 'N/A' for invalid states.
Symbology

R → Q
S → Q'

Diagram with inputs R and S and outputs Q and Q'
GLATCH

Gated Latches
The Gated SR Latch

When GATE='0' ⇔
GR=GS='0' ⇔
latch cannot be modified

When GATE='1' ⇔
GR=R, GS=S ⇔
works like an SR latch

The GATE signal allows us to control
when
the latch will be loaded with a new value
Gated SR Latch

- Sometimes known as a *loadable SR latch*
  - Can be *loaded* with new value
The Gated D Latch

\[ Q^+ = GATE \cdot D + GATE' \cdot Q \]

<table>
<thead>
<tr>
<th>GATE</th>
<th>D</th>
<th>Q</th>
<th>Q^+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When GATE='1' \iff Q follows D \hspace{1cm} \text{(storage)}

When GATE='0' \iff Q retains old value \hspace{1cm} \text{(retention)}
Gated D Latches

• Sometimes called a transparent latch
  - When GATE='1':
    • Q follows D
    • D is reflected on Q output

• Allows us to control when to store new data into latch
  - D = data to be stored
  - GATE = control signal
Symbology

D \rightarrow Q

GATE

D \rightarrow Q

Gate

or…

D \rightarrow Q

Load
MSFF

Master/Slave Flip Flops
A Master/Slave Flip Flop (D Type)

Either:
  The master is loading (the master in \textit{on})
or
  The slave is loading (the slave is \textit{on})

But never both at the same time...
Alternative Flip Flops

T

JK
Toggle Flip Flop

![Toggle Flip Flop Diagram]

\[ Q^+ = T' \cdot Q + T \cdot Q' = T \oplus Q \]

<table>
<thead>
<tr>
<th>T</th>
<th>Q</th>
<th>Q^+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Clock edge is assumed in this transition table...
JK Flip Flop

Kind of a cross between a SR FF and a T FF

\[ Q^+ = K' \cdot Q + J \cdot Q' \]
Why Alternative FF’s?

• With discrete parts (TTL family)
  - JK or T FF’s could reduce gate count for the input forming logic
  - Extensively used

• With VLSI IC’s and FPGA’s
  - JK or T FF’s must be built from DFF+gates
  - Larger, slower than a DFF
  - Not used
Flip Flops With Additional Control Inputs
A falling edge triggered, D-type FF with enable

Master only loads when \( \text{CLK}=\text{Enable}='1' \)
What is this?

A falling edge triggered, D-type FF with an asynchronous set

If Set=1 then Q=>1, regardless of CLK or D
What is this?

A falling edge triggered, D-type FF with a **synchronous set**

If Set=1 then Q=>1 on the next falling edge of the clock, regardless of D
Flip Flops With Additional Control Inputs

• A variety of FF’s have been made over the years
• They contain combinations of these inputs:
  - Enable
  - Set
  - Reset
• The Set and Reset can be either:
  - Asynchronous (independent of CLK)
  - Synchronous (work only on CLK edge)
Flip Flop Timing Characteristics
Clock-to-Q Time ($t_{CLK \rightarrow Q}$)

The output does not change instantaneously...

Why $2 \times t_{NOR}$?

$t_{CLK \rightarrow Q} = t_{NOT} + t_{AND} + 2 \times t_{NOR}$
Setup Time ($t_{\text{setup}}$)

\[ t_{\text{setup}} = t_{\text{NOT}} + t_{\text{AND}} + 2 \times t_{\text{NOR}} \]

The input has to get there early enough to set the master latch before the clock turns off...
Rising Edge FF Setup Time ($t_{\text{SETUP}}$)

Same setup time as before

Clock is delayed through the NOT gate
Falling Edge Hold Time ($t_{\text{hold}}$)

$t_{\text{hold}} = 0\text{ns}$  (AND gates turn off immediately)

You have to keep the old D value there until the AND gates are shut off... (but no longer)
Rising Edge Hold Time ($t_{\text{hold}}$)

You have to keep the old D value there until the AND gates are shut off...
Timing of a Synchronous System

D Q  Q D

CLK

Input Forming Logic

D Q

CLK

\[ t_{CYCLE} \geq t_{CLK \rightarrow Q} + t_{IFL} + t_{SETUP} \]
Example of a Synchronous System

One transition per clock edge...
General Sequential Systems

Input Forming Logic → Next State → State Memory → Current State
A Sequential Counter

The current state loads the next state values in response to the clock edge.

IFL reacts after some gate delays to produce a new next state.
Transition Table for 2-Bit Counter

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Q0</td>
<td>N1 N0</td>
<td>Q1 Q0</td>
<td>N1 N0</td>
</tr>
<tr>
<td>00 00</td>
<td>01 01</td>
<td>00 00</td>
<td>01 01</td>
</tr>
<tr>
<td>01 01</td>
<td>10 10</td>
<td>01 01</td>
<td>10 10</td>
</tr>
<tr>
<td>10 10</td>
<td>11 11</td>
<td>10 10</td>
<td>11 11</td>
</tr>
<tr>
<td>11 11</td>
<td>00 00</td>
<td>11 11</td>
<td>00 00</td>
</tr>
</tbody>
</table>

It is the truth table for the input forming logic...

It describes what the *next state* values are as a function of the *current state* (clock is assumed)
General Counter Design Procedure

• Write transition table for counter
  - Use X’s as appropriate
• Reduce each Nx variable to an equation
• Implement input forming logic (IFL) using gates
• Draw schematic using FF’s + IFL
Counters With Outputs

Outputs = f(Current State)
### Combined Transition Table

<table>
<thead>
<tr>
<th>Q2</th>
<th>Q1</th>
<th>Q0</th>
<th>N2</th>
<th>N1</th>
<th>N0</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Current state**
- **Next state**
- **Output**

\[ Z = Q2'Q1'Q0' + Q2'Q1Q0 + Q2Q1Q0' \]

_(implement OFL with gates)_
State Graphs
Binary Counter State Graph

State graphs are graphical representations of TT’s

They contain the same information: no more, no less

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q0</th>
<th>N1</th>
<th>N0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
State Graphs for Counters With Inputs

There is a one-to-one correspondence between the rows of the TT and the arcs in the SG.

INC controls whether transition is taken or not...

<table>
<thead>
<tr>
<th>INC</th>
<th>Q1</th>
<th>Q0</th>
<th>N1</th>
<th>N0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Transition Table Simplification

<table>
<thead>
<tr>
<th>CLR</th>
<th>INC</th>
<th>Q1</th>
<th>Q0</th>
<th>N1</th>
<th>N0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

These are **input** don’t cares. They are a shorthand for the TT on the left. This TT exactly matches SG on previous page.
Simplified Transition Tables With Input Don’t Cares

- Contain *exactly* same information as original
  - Shorthand way of writing
- Should be able to easily convert back/forth
Design Procedure Using State Graphs

1. Draw the state graph

2. Create an equivalent transition table

3. If transition table contains input don’t cares,
   - *unfold* it to a full transition table

4. Complete the design using KMaps, gates, FF’s
Cascaded Counters
A Mod4 Counter With a Rollover Signal

<table>
<thead>
<tr>
<th>CLR</th>
<th>INC</th>
<th>Q1</th>
<th>Q0</th>
<th>N1</th>
<th>N0</th>
<th>Rollover</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Signal Rollover can be used to tell other circuitry that counter is rolling over to all 0's

Mealy output
Cascading 2 Mod4 Counters

Digit1  Digit0

Count sequence

Increment higher digit's counter when lower digit's counter is rolling over

Mod4 Counter

Digit1

clk

Rollover1

CLR

INC

Mod4 Counter

Digit0

clk

Rollover0

CLR
3 Digits' Worth

Increment higher digit's counter when lower digit's counter is rolling over.

Can do this with *any* counter that has a Rollover output.

Could build a digital watch or clock circuit this way with Mod60 and Mod24 counters.
Cascading Counters

• A counter will increment only when
  - The counter below it is at its terminal count and it is being incremented
  • That is the definition of the Rollover signal

• Some people try to tie Rollover to the clk input of the next higher counter
  - Bad idea... Very bad idea...
  - Violates our Globally Synchronous policy
  - Doesn’t work as intended
Ripple Counters

• When you tie a rollover-like signal to a clock on the next higher digit ⇔ ripple counter

• A ripple counter is an ASYNCHRONOUS counter
  - Transitions are not all synchronized to the clock
  - Different flip flops change at different times
  - Similar to gated clocks (seen earlier)

• Asynchronous circuits are an advanced topic
Another Common Ripple Counter

Sequence is:

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
0000

So what is the problem?
Mod4 Counter
A Mod4 Counter

The right way!

A Mod4 Counter

The right way!
A Mod4 Counter

MOD4

Clear

Count Value

Increment

Roll Over

Clock
Registers
A 4-Bit Register

Could be called a parallel-in/parallel-out register.

Why?
A Shift Register

Called a *serial-in, parallel-out* shift register (SIPO)
SIPO Register (Serial-In/Parallel-Out)
SISO Register (Serial-In/Serial-Out)

Useful for delaying a serial bit-stream some number of cycles...
Gated Clocking

- Different flip flops load at different times
  - A form of *clock skew*
  - Makes doing timing analysis more difficult
  - Can lead to circuits which *run more slowly*
  - Can lead to circuits which *fail* at any clock rate
Globally Synchronous Design

- One global clock
- All registers load on that clock’s edge
- Control over loading done via input forming logic (IFL)

- Simplifies timing analysis and requirements
- Makes it possible for even novices to design large, functioning circuits

- Multi-clock circuits ⇔ next semester’s topic
The Correct Way To Make A Loadable Register (1-Bit)

When LOAD='0', FF loads old value

When LOAD='1', FF loads DIN
A Loadable Parallel-In, Parallel-Out Register

DIN(3:0) → LOAD

D Q

CLK

Q(3:0)

PIPO ?
MUX for Register Control

- Loadable register concept can be generalized
  - Provide any combination of inputs to register
Uses of Shift Registers

- Collecting serial input data into a parallel word
- Shifting out bits of a word
- Delaying a serial stream by some # of cycles
A Clearable Counter

From there to here, from here to there, interesting circuits are everywhere...

(when you have a MUX and some flip flops)...
An Up/Down Counter

How about an up/down counter + bi-directional shift register design?
Up/Down Counter + Bi-Directional Shift Register

<table>
<thead>
<tr>
<th>Control</th>
<th>NextQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Q+1</td>
</tr>
<tr>
<td>01</td>
<td>Q-1</td>
</tr>
<tr>
<td>10</td>
<td>Q shifted left</td>
</tr>
<tr>
<td>11</td>
<td>Q shifted right</td>
</tr>
</tbody>
</table>

Diagram:
- Inputs: Q+1, Q-1, Q<<1, Q>>1
- Controls: Control, CLK
- Outputs: Q

Truth Table:
- 00: Q+1
- 01: Q-1
- 10: Q shifted left
- 11: Q shifted right
An Accumulator

Values to be added are placed on A input, one per cycle. Register accumulates their sum.

This one loads 0 when CLR='1'

This one loads A when CLR='1'

Both work, they just have different timings...
Register Files

Small memories holding multiple words of data
Typical Register File

DataIn
\[ n \]
\[ \uparrow \]
clk
regWE
\[ \downarrow \]
Addr
\[ m \]
RegFile

DataOut
\[ n \]
\[ \downarrow \]

ECE238L
Building a Register File

Write Decoder

Register write signals

Reg0
Reg1
Reg2
Reg3
Reg4
Reg5
Reg6
Reg7

8:1 MUX

DataOut

Addr

regWE
DataIn
clk

m=3

n
n
n
n
n
n
n
n

ECE238L © 2006
Multi-Ported Register File

One write port
Two read ports

Can be reading from two locations on same cycle you write to another location

Useful for microprocessor design

Reg0
Reg1
Reg2
Reg3
Reg4
Reg5
Reg6
Reg7

WAddr WE
DataIn
clk
RAddr1
RAddr2

8:1 MUX
DataOut1
DataOut2

Write Decoder
Register write signals
Memories vs. Register Files

- Random Access Memory (RAM) is similar to a register file
  - Stores many multi-bit words for reading/writing
- RAM usually only single-ported
- RAM usually much, much larger
  - Mbytes instead of bytes
- RAM implementation conceptually the same as register file
  - Transistor-level implementation different due to size/usage characteristics
- RAM design beyond the scope of this class
Finite State Machines
State Machine Concepts

- State, current state, next state, state registers
- IFL, OFL, Moore outputs, Mealy outputs
- Transition tables
  - With output don’t cares (X’s)
  - With input don’t cares (-’s)
- State graphs
  - And their correspondence to TT’s
State Machines

- A state machine is a *sequential circuit* which progresses through a series of states in response to inputs
  - The output values are usually significant
  - The state encodings are usually not significant
- Unlike with counters
Implementing the Sequence Detector FSM

1. Create symbolic Transition Table
2. Assign state encoding
3. Create conventional Transition Table
4. Do standard implementation steps

<table>
<thead>
<tr>
<th>Xin CS</th>
<th>NS</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 S0</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>1 S0</td>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>0 S1</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>1 S1</td>
<td>S2</td>
<td>0</td>
</tr>
<tr>
<td>0 S2</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>1 S2</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>- S3</td>
<td>S3</td>
<td>1</td>
</tr>
</tbody>
</table>

Symbolic TT

<table>
<thead>
<tr>
<th>Xin Q1</th>
<th>Q0</th>
<th>N1</th>
<th>N0</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

State Assignment

Conventional TT
Sequence Detector Implementation

\[ N1 = Q1 \cdot Q0 + Xin \cdot Q1 + Xin \cdot Q0 \]
\[ N0 = Xin' + Q1 \]
\[ Z = Q1 \cdot Q0 \]
Resetting State Machines

• Ability to reset the FSM is essential for testing most systems

• Always include a reset capability
  - Add CLR signal to state graph
  - Use flip flops with clear inputs

  - Either method will work
One-Hot Encoded
Finite State Machines
One-Hot - Observations

• Choosing a one-hot encoding results in many, many don’t cares in transition table

• Minimization results in simpler IFL and OFL

• Can do one-hot design by inspection
  – without using transition tables...
Other State Encoding Techniques

• You have learned the 2 extremes
  - Fully encoded (8 states ⇔ 3 state bits)
  - One-hot encoded (8 states ⇔ 8 state bits)

• A range of options exist in between

• A good choice of encoding
  - Can minimize IFL and OFL complexity
  - Algorithms have been developed for this...
  - Beyond the scope of this class