Chapter #7: Sequential Logic Case Studies

Contemporary Logic Design

Storage Register

Group of storage elements read/written as a unit

4-bit register constructed from 4 D FFs
Shared clock and clear lines
**Input/Output Variations**

- **Selective Load Capability**
- **Tri-state or Open Collector Outputs**
- **True and Complementary Outputs**

### 74377 Octal D-type FFs with input enable

- EN enabled low and low-to-high clock transition to load new data into register

### 74374 Octal D-type FFs with output enable

- OE asserted low presents FF state to output pins; otherwise high impedance

---

**Register Files**

- Two dimensional array of flipflops
- Address used as index to a particular word
- Word contents read or written

### 74670 4x4 Register File with Tri-state Outputs

- Contains 16 D-ffs, organized as four rows (words) of four elements (bits)

---

**74377 Octal D-type FFs**

```
11  11
  11  11
CLK  CLK
  18  16
EN   H  Q7  19
  17  18
D7   D  Q6  T6
  14  15
D6   G  Q5  T5
  13  14
D5   F  Q4  T4
  12  13
D4   E  Q3  T3
  11  12
D3   D  Q2  T2
  10  11
D2   C  Q1  T1
  9   10
D1   B  Q0  T0
  8   9
D0   A  Q  
```

**74374 Octal D-type FFs**

```
11  11
  11  11
CLK  CLK
  18  16
QH   Q  Q16
  17  18
D7   Q  Q15
  14  15
D6   E  Q14
  13  14
D5   D  Q13
  12  13
D4   C  Q12
  11  12
D3   B  Q11
  10  11
D2   A  Q10
  9   10
D1   G  Q9
  8   9
D0   F  Q8
```

---

**670 4x4 Register File**

```
11  11
  11  11
RE   WE
  12  12
RB   WB
  10  11
RA   WA
  9   10
WB
  8   9
WE
  7   8
WA
  6   7
WB
  5   6
WE
  4   5
WA
  3   4
WB
  2   3
WE
  1   2
WA
```

---

**Separate Read and Write Enables**

- Separate Read and Write Address
- Data Input, Q Outputs
### Shift Registers

**Storage + ability to circulate data among storage elements**

- **Shift from left storage element to right neighbor on every low-to-high transition on shift signal.**
- **Wrap around from rightmost element to leftmost element.**

**Master Slave FFs:** sample inputs while clock is high; change outputs on falling edge.

---

### Shift Register I/O

**Serial vs. Parallel Inputs**
- **Serial Inputs:** LSI, RSI
- **Parallel Inputs:** D, C, B, A

**Serial vs. Parallel Outputs**
- **Parallel Outputs:** QD, QC, QB, QA

**Clear Signal**
- **Positive Edge Triggered Devices**

- **S1, S0 determine the shift function**
  - S1 = 1, S0 = 1: Load on rising clk edge (synchronous load)
  - S1 = 1, S0 = 0: Shift left on rising clk edge (LSI replaces element D)
  - S1 = 0, S0 = 1: Shift right on rising clk edge (RSI replaces element A)
  - S1 = 0, S0 = 0: Hold state

**74194 4-bit Universal Shift Register**

- Multiplexing logic on input to each FF!

---

Shifters well suited for serial-to-parallel conversions, such as terminal to computer communications.
Shift Register Application: Parallel to Serial Conversion

Counters
- Proceed through a well-defined sequence of states in response to count signal
- 3 Bit Up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, …
- 3 Bit Down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, …
- Binary vs. BCD vs. Gray Code Counters

A counter is a "degenerate" finite state machine/sequential circuit where the state is the only output
**Johnson (Mobius) Counter**

8 possible states, single bit change per state, useful for avoiding glitches

**Catalog Counter**

- Synchronous Load and Clear Inputs
- Positive Edge Triggered FFs
- Parallel Load Data from D, C, B, A
- P, T Enable Inputs: both must be asserted to enable counting
- RCO: asserted when counter enters its highest state 1111, used for cascading counters

"Ripple Carry Output"

74161: similar in function, asynchronous load and reset
Counter Design Procedure

Introduction

This procedure can be generalized to implement ANY finite state machine

Counters are a very simple way to start: no decisions on what state to advance to next current state is the output
**Example: 3-bit Binary Upcounter**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Flipflop Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(000)</td>
<td>(000)</td>
<td>(0)</td>
</tr>
<tr>
<td>(001)</td>
<td>(001)</td>
<td>(0)</td>
</tr>
<tr>
<td>(010)</td>
<td>(011)</td>
<td>(0)</td>
</tr>
<tr>
<td>(011)</td>
<td>(100)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

State Transition Diagram

State Table

Flipflop Input Table

Decide to implement with Toggle Flipflops

What inputs must be presented to the T FFs to get them to change to the desired state bit?

This is called "Remapping the Next State Function"

---

**Counter Design Procedure**

**Introduction**

This procedure can be generalized to implement ANY finite state machine.

Counters are a very simple way to start: no decisions on what state to advance to next current state is the output.

**Example: 3-bit Binary Upcounter**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Flipflop Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(000)</td>
<td>(000)</td>
<td>(0)</td>
</tr>
<tr>
<td>(001)</td>
<td>(001)</td>
<td>(0)</td>
</tr>
<tr>
<td>(010)</td>
<td>(011)</td>
<td>(0)</td>
</tr>
<tr>
<td>(011)</td>
<td>(100)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

State Transition Diagram

State Table

Flipflop Input Table

Decide to implement with Toggle Flipflops

What inputs must be presented to the T FFs to get them to change to the desired state bit?

This is called "Remapping the Next State Function"
K-maps for Toggle Inputs:

<table>
<thead>
<tr>
<th>CB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

TA = 1

<table>
<thead>
<tr>
<th>CB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TB = A

<table>
<thead>
<tr>
<th>CB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

TC = A • B

More Complex Count Sequence

Step 1: Derive the State Transition Diagram

Count sequence: 000, 010, 011, 101, 110

Step 2: State Transition Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C 3</td>
<td>C+</td>
</tr>
</tbody>
</table>
**Step 1:** Derive the State Transition Diagram

Sequence: 000, 010, 011, 101, 110

![State Transition Diagram](image)

**Step 2:** State Transition Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C B</td>
<td>C+ A</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>X</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>X</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>X</td>
</tr>
</tbody>
</table>

Note the Don't Care conditions

**Step 3:** K-Maps for Next State Functions

- C+ = ![K-Map for C+](image)
- B+ = ![K-Map for B+](image)
- A+ = ![K-Map for A+](image)
Step 3: K-Maps for Next State Functions

Step 4: Choose Flipflop Type for Implementation
Use Excitation Table to Remap Next State Functions
Counter Design Procedure

More Complex Counter Sequencing

Step 4: Choose Flipflop Type for Implementation
Use Excitation Table to Remap Next State Functions

**Toggle Excitation Table**

<table>
<thead>
<tr>
<th>Q</th>
<th>Q⁺</th>
<th>T</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>

**Remapped Next State Functions**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Toggle Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>IC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Remapped K-Maps**

TC

TB

TA

TC =
TB =
TA =
Remapped K-Maps

\[ TC = \overline{A} \overline{C} + A \overline{C} = A \text{xor} C \]
\[ TB = A + \overline{B} + C \]
\[ TA = \overline{A} B \overline{C} + B \overline{C} \]

Resulting Logic:
5 Gates
13 Input Literals + Flipflop connections

Timing Waveform:
**Self-Starting Counters**

**Start-Up States**

At power-up, counter may be in possible state

Designer must guarantee that it (eventually) enters a valid state

Especially a problem for counters that validly use a subset of states

**Self-Starting Solution:**

Design counter so that even the invalid states eventually transition to valid state

Two Self-Starting State Transition Diagrams for the Example Counter

**Deriving State Transition Table from Don't Care Assignment**

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>
### Implementation with Different Kinds of FFs

#### R-S Flipflops

Continuing with the 000, 010, 011, 101, 110, 000, ... counter example

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Remapped Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>0 0 X 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 X</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**RS Excitation Table**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Remapped Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Remapped Next State Functions

\[Q+ = S + R \bar{S}\]
Implementation with Different Kinds of FFs

RS FFs Continued

RC

SC

RB

SB

RA

SA

RC = \bar{A}

SC = A

RB = A B + B C

SB = B

RA = C

SA = B C
Implementation With Different Kinds of FFs

**RS FFs Continued**

Resulting Logic Level Implementation:
3 Gates, 11 Input Literals + Flipflop connections

---

Implementation with Different FF Types

**J-K FFs**

<table>
<thead>
<tr>
<th>C</th>
<th>Q^+</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ Q^+ = J \overline{Q} + K Q \]

J-K Excitation Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Remapped Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Q^+</td>
<td>J</td>
</tr>
<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>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

Remapped Next State Functions
Implementation with Different FF Types

**J-K FFs**

<table>
<thead>
<tr>
<th>G</th>
<th>Q+</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ Q^+ = J \bar{Q} + K \bar{Q} \]

**J-K Excitation Table**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Remapped Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>J+</td>
<td>K+</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Remapped Next State Functions

**Implementation with Different FF Types**

**J-K FFs Continued**

\[ JC = \]
\[ KC = \]
\[ JB = \]
\[ KB = \]
\[ JA = \]
\[ KA = \]
Implementation with Different FF Types

J-K FFs Continued

\[
\begin{align*}
JC &= A \\
KC &= A \\
JB &= 1 \\
KB &= A + C \\
JA &= B \cdot C \\
KA &= C
\end{align*}
\]

Resulting Logic Level Implementation:
2 Gates, 10 Input Literals + Flipflop Connections
**Implementation with Different FF Types**

* **D FFs**
  - Simplest Design Procedure: No remapping needed!
  - \( DC = A \)
  - \( DB = \overline{A} \overline{C} + \overline{B} \)
  - \( DA = B \overline{C} \)

  ![Diagram of D Flipflops]

  Resulting Logic Level Implementation:
  3 Gates, 8 Input Literals + Flipflop connections

**Implementation with Different FF Types**

* **Comparison**
  - T FFs well suited for straightforward binary counters
    - But yielded worst gate and literal count for this example!
  - No reason to choose R-S over J-K FFs: it is a proper subset of J-K
    - R-S FFs don't really exist anyway
    - J-K FFs yielded lowest gate count
    - Tend to yield best choice for packaged logic where gate count is key
  - D FFs yield simplest design procedure
    - Best literal count
    - D storage devices very transistor efficient in VLSI
    - Best choice where area/literal count is the key
Asynchronous vs. Synchronous Counters

Ripple Counters
Deceptively attractive alternative to synchronous design style

Count signal ripples from left to right

State transitions are not sharp!

Can lead to "spiked outputs" from combinational logic decoding the counter's state

Asynchronous vs. Synchronous Counters

Cascaded Synchronous Counters with Ripple Carry Outputs

First stage RCO enables second stage for counting
RCO asserted soon after stage enters state 1111
also a function of the T Enable
Downstream stages lag in their 1111 to 0000 transitions
Affects Count period and decoding logic
Asynchronous vs. Synchronous Counters

The Power of Synchronous Clear and Load

Starting Offset Counters:
- e.g., 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 0110, ...

Use RCO signal to trigger Load of a new state

Since 74163 Load is synchronous, state changes only on the next rising clock edge

0110 is the state to be loaded

Asynchronous vs. Synchronous Counters

Offset Counters Continued

Ending Offset Counter:
- e.g., 0000, 0001, 0010, ..., 1100, 1101, 0000

Clear signal takes effect on the rising count edge

Decode state to determine when to reset to 0000

Replace '163 with '161, Counter with Async Clear
Clear takes effect immediately!