Counters

- Counters are sequential circuits which "count" through a specific state sequence. They can count up, count down, or count through other fixed sequences. Two distinct types are in common usage:
  - Ripple Counters
    - Clock is connected to the flip-flop clock input on the LSB bit flip-flop
    - For all other bits, a flip-flop output is connected to the clock input, thus circuit is not truly synchronous
    - Output change is delayed more for each bit toward the MSB.
    - Resurgent because of low power consumption
  - Synchronous Counters
    - Clock is directly connected to the flip-flop clock inputs
    - Logic is used to implement the desired state sequencing

Ripple Counter

- How does it work?
  - When there is a positive edge on the clock input of A, A complements
  - The clock input for flip-flop B is the complemented output of flip-flop A
  - When flip A changes from 1 to 0, there is a positive edge on the clock input of B causing B to complement
• The arrows show the cause-effect relationship from the prior slide =>
• The corresponding sequence of states => (B,A) = (0,0), (0,1), (1,0), (1,1), (0,0), (0,1), ...
• Each additional bit, C, D, ... behaves like bit B, changing half as frequently as the bit before it.
• For 3 bits: (C,B,A) = (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1), (0,0,0), ...

Ripple Counter (continued)

• Starting with C = B = A = 1, equivalent to (C,B,A) = 7 base 10, the next clock increments the count to (C,B,A) = 0 base 10. In fine timing detail:
  – The clock to output delay \( t_{\text{PHL}} \) causes an increasing delay from clock edge for each stage transition.
  – Thus, the count “ripples” from least to most significant bit.
  – For \( n \) bits, total worst case delay is \( n \cdot t_{\text{PHL}} \).

Ripple Counter (continued)

• These circuits are called ripple counters because each edge sensitive transition (positive in the example) causes a change in the next flip-flop’s state.
• The changes “ripple” upward through the chain of flip-flops, i.e., each transition occurs after a clock-to-output delay from the stage before.
• To see this effect in detail look at the waveforms on the next slide.

Synchronous Counters

• To eliminate the “ripple” effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
• For an up-counter, use an incrementer =>

\[
\begin{align*}
\text{Incrementer} & : & D_3 & Q_3 & D_2 & Q_2 & D_1 & Q_1 & D_0 & Q_0 \\
\text{Clock} & : & I_{\text{ncr}} & C & A_3 & S_3 & A_2 & S_2 & A_1 & S_1 & A_0 & S_0 \\
\end{align*}
\]

• Internal details =>
  – **Internal Logic**
    – XOR complements each bit
    – AND chain causes complement of a bit if all bits toward LSB from it equal 1
  – **Count Enable**
    – Forces all outputs of AND chain to 0 to “hold” the state
  – **Carry Out**
    – Added as part of incrementer
    – Connect to Count Enable of additional 4-bit counters to form larger counters

Synchronous Counters (continued)

• Carry chain
  – series of AND gates through which the carry “ripples”
  – Yields long path delays
  – Called serial gating
• Replace AND carry chain with ANDs => in parallel
  – Reduces path delays
  – Called parallel gating
  – Like carry lookahead
  – Lookahead can be used on COs and ENs to prevent long paths in large counters

• Symbol for Synchronous Counter

Ripple Counter (continued)

• Internal details =>
  – **Internal Logic**
    – XOR complements each bit
    – AND chain causes complement of a bit if all bits toward LSB from it equal 1
  – **Count Enable**
    – Forces all outputs of AND chain to 0 to “hold” the state
  – **Carry Out**
    – Added as part of incrementer
    – Connect to Count Enable of additional 4-bit counters to form larger counters
Other Counters

• See text for:
  – **Down Counter** - counts downward instead of upward
  – **Up-Down Counter** - counts up or down depending on value a control input such as Up/Down
  – **Parallel Load Counter** - has parallel load of values available depending on control input such as Load

• **Divide-by-n (Modulo n) Counter**
  – Count is remainder of division by n which n may not be a power of 2 or
  – Count is arbitrary sequence of n states specifically designed state-by-state
  – Includes modulo 10 which is the BCD counter

---

Counter with Parallel Load

• Add path for input data
  – enabled for Load = 1

• Add logic to:
  – disable count logic for Load = 1
  – disable feedback from outputs for Load = 1
  – enable count logic for Load = 0 and Count = 1

• The resulting function table:

<table>
<thead>
<tr>
<th>Load</th>
<th>Count</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Hold Stored Value</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Count Up Stored Value</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Load D</td>
</tr>
</tbody>
</table>

---

Design Example: Synchronous BCD

• Use the sequential logic model to design a synchronous BCD counter with D flip-flops
• State Table →
• Input combinations 1010 through 1111 are don’t cares

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>(Q8)</th>
<th>(Q4)</th>
<th>(Q2)</th>
<th>(Q1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q8Q4Q2Q1)</td>
<td>(Q8Q4Q2Q1)</td>
<td>0 0 0 1</td>
<td>0 0 1 0</td>
<td>0 1 1 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>(0 0 0 1)</td>
<td>0 0 1 0</td>
<td>0 1 1 0</td>
<td>0 1 0 1</td>
<td>0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>(0 1 0 0)</td>
<td>0 1 0 1</td>
<td>0 1 0 0</td>
<td>0 1 1 0</td>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>(0 1 1 1)</td>
<td>0 1 1 0</td>
<td>0 1 1 1</td>
<td>0 0 1 1</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

---

Synchronous BCD (continued)

• Use K-Maps to two-level optimize the next state equations and manipulate into forms containing XOR gates:
  - \(D1 = Q1\)
  - \(D2 = Q2 \oplus Q1 \oplus Q8\)
  - \(D4 = Q4 \oplus Q1 \oplus Q2\)
  - \(D8 = Q8 \oplus (Q1 \oplus Q8 + Q1 \oplus Q2 \oplus Q4)\)

• The logic diagram can be draw from these equations
  – An asynchronous or synchronous reset should be added

• What happens if the counter is perturbed by a power disturbance or other interference and it enters a state other than 0000 through 1001?

---

Synchronous BCD (continued)

• For the BCD counter design, if an invalid state is entered, return to a valid state occurs within two clock cycles
• Is this adequate? If not:
  – Is a signal needed that indicates that an invalid state has been entered? What is the equation for such a signal?
  – Does the design need to be modified to return from an invalid state to a valid state in one clock cycle?
  – Does the design need to be modified to return from an invalid state to a specific state (such as 0)?

• The action to be taken depends on:
  – the application of the circuit
  – design group policy

• See pages 278 - 279 of the text.
• The following techniques use an $n$-bit binary counter with asynchronous or synchronous clear and/or parallel load:
  – Detect a terminal count of $N$ in a Modulo-$N$ count sequence to asynchronously Clear the counter to 0 or asynchronously Load in value 0 (These lead to counts which are present for only a very short time and can fail to work for some timing conditions!)
  – Detect a terminal count of $N$ - 1 in a Modulo-$N$ count sequence to Clear the counter synchronously to 0
  – Detect a terminal count of $N$ - 1 in a Modulo-$N$ count sequence to synchronously Load in value 0
  – Detect a terminal count and use Load to preset a count of the terminal count value minus (N - 1)
• Alternatively, custom design a modulo N counter as done for BCD

Counting Modulo 7: Detect 7 and Asynchronously Clear

• A synchronous 4-bit binary counter with an asynchronous Clear is used to make a Modulo 7 counter.
  Use the Clear feature to detect the count 7 and clear the count to 0. This gives a count of 0, 1, 2, 3, 4, 5, 6, 7(short)0, 1, 2, 3, 4, 5, 6, 7(short)0, etc.
• DON'T DO THIS! Referred to as a “suicide” counter! (Count “7” is “killed,” but the designer’s job may be dead as well!)

Counting Modulo 7: Synchronously Load on Terminal Count of 6

• A synchronous 4-bit binary counter with a synchronous Clear is used to make a Modulo 7 counter.
• Use the Load feature to detect the count “6” and load in “zero”. This gives a count of 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, ...
• Using don’t cares for states above 0110, detection of 6 can be done with Load = Q2 Q1

Register Cell Design

• Assume that a register consists of identical cells
• Then register design can be approached as follows:
  – Design representative cell for the register
  – Connect copies of the cell together to form the register
  – Applying appropriate “boundary conditions” to cells that need to be different and contract if appropriate
• Register cell design is the first step of the above process

Register Cell Specifications

• A register
• Data inputs to the register
• Control input combinations to the register
  – Example 1: Not encoded
    • Control inputs: Load, Shift, Add
    • At most, one of Load, Shift, Add is 1 for any clock cycle
      (0,0,0), (1,0,0), (0,1,0), (0,0,1)
  – Example 2: Encoded
    • Control inputs: S1, S0
    • All possible binary combinations on S1, S0
      (0,0), (0,1), (1,0), (1,1)
**Register Cell Specifications**

- A set of register functions (typically specified as register transfers)
  - Example:
    - Load: $A \leftarrow B$
    - Shift: $A \leftarrow \text{sr} B$
    - Add: $A \leftarrow A + B$
- A hold state specification
  - Example:
    - Control inputs: Load, Shift, Add
    - If all control inputs are 0, hold the current register state

**Multiplexer Approach**

- Uses an n-input multiplexer with a variety of transfer sources and functions

**Example 1: Register Cell Design**

- Load enable by OR of control signals $K_0, K_1, \ldots, K_{n-1}$
  - assumes no load for 00…0
- Use:
  - Encoder + Multiplexer (shown) or
  - $n \times 2$ AND-OR
  - to select sources and/or transfer functions

**Example 1: Register Cell Design (continued)**

- Load Control
  - Load = CX + CY
- Since all control combinations appear as if encoded (0,0), (0,1), (1,0) can use multiplexer without encoder:
  - $S_1 = CX$
  - $S_0 = CY$
  - $D_0 = A_1$
  - $D_1 = A_1 \oplus B_1$
  - $D_2 = A_1 \oplus B_1$
- Note that the decoder part of the 3-input multiplexer can be shared between bits if desired

**Sequential Circuit Design Approach**

- Find a state diagram or state table
  - Note that there are only two states with the state assignment equal to the register cell output value
- Use the design procedure in Chapter 6 to complete the cell design
- For optimization:
  - Use K-maps for up to 4 to 6 variables
  - Otherwise, use computer-aided or manual optimization
**Example 1 Again**

- State Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CX</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CY</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Four variables give a total of 16 state table entries
- By using:
  - Combinations of variable names and values
  - Don’t care conditions (for CX = CY = 1)

  only 8 entries are required to represent the 16 entries

- **Example 1 Again (continued)**

- The resulting SOP equation:

  \[ D_i = CX B_i + CY A_i + A_i B_i + CY A_i \]

- Using factoring and DeMorgan’s law:

  \[ D_1 = CX B_i + A_i (CY B_i) \]

  The gate input cost per cell is: 2 + 8 + 2 + 2 = 14

  - The gate input cost per cell for the previous version is:

    Per cell: 19
    Shared decoder logic: 8
    Cost gain by sequential design > 5 per cell
    Also, no Enable on the flip-flop makes it cost less

**Example 1 Again (continued)**

- K-map - Use variable ordering CX, CY, A, B, and assume a D flip-flop

- Multiplexer and Bus-Based Transfers for Multiple Registers

- **Dedicated MUX-Based Transfers**

  - Multiplexer connected to each register input produces a very flexible transfer structure =>
  - Characterize the simultaneous transfers possible with this structure.

- **Multiplexer Bus**

  - A single bus driven by a multiplexer lowers cost, but limits the available transfers =>
  - Characterize the simultaneous transfers possible with this structure.
  - Characterize the cost savings compared to dedicated multiplexers
Three-State Bus
- The 3-input MUX can be replaced by a 3-state node (bus) and 3-state buffers.
- Cost is further reduced, but transfers are limited
- Characterize the simultaneous transfers possible with this structure.
- Characterize the cost savings and compare
- Other advantages?

Serial Transfers and Microoperations
- Serial Transfers
  - Used for “narrow” transfer paths
  - Example 1: Telephone or cable line
    - Parallel-to-Serial conversion at source
    - Serial-to-Parallel conversion at destination
  - Example 2: Initialization and Capture of the contents of many flip-flops for test purposes
    - Add shift function to all flip-flops and form large shift register
    - Use shifting for simultaneous Initialization and Capture operations
- Serial microoperations
  - Example 1: Addition
  - Example 2: Error-Correction for CDs

Serial Microoperations
- By using two shift registers for operands, a full adder, and a flip flop (for the carry), we can add two numbers serially, starting at the least significant bit.
- Serial addition is a low cost way to add large numbers of operands, since a “tree” of full adder cells can be made to any depth, and each new level doubles the number of operands.
- Other operations can be performed serially as well, such as parity generation/checking or more complex error-check codes.
- Shifting a binary number left is equivalent to multiplying by 2.
- Shifting a binary number right is equivalent to dividing by 2.

Serial Adder
- The circuit shown uses two shift registers for operands A(3:0) and B(3:0).
- A full adder, and one more flip flop (for the carry) is used to compute the sum.
- The result is stored in the A register and the final carry in the flip-flop
- With the operands and the result in shift registers, a tree of full adders can be used to add a large number of operands. Used as a common digital signal processing technique.