Section II: Digital System Design and Synthesis

CEG 360/560; EE 451/651
Digital System Design
Dr. Travis Doom, Assistant Professor
Department of Computer Science and Engineering
Wright State University

Acknowledgements

- These slides were developed with the aid of examples found in:
  - “Logic and Computer Design Fundamentals” - M. Morris Mano

- The original version of many of these slides were kindly provided by:
  - Dr. Roger L. Haggard et al
  - Prentice Hall, Inc.

Outline

- CSSM Design
  - The synthesis process
  - State machine design

- State minimization and assignment

- Ad-hoc Design
  - Finite Memory Machines

- Timing Issues
  - Clock Skew
  - Input synchronization
State Machine Design Procedure

1. Build state/output table (or state diagram) from word description
2. Minimize number of states
3. Choose state variables and assign bit combinations to named states
4. Build transition/output table from state/output table (or state diagram)
5. Choose flip-flop type (D, J-K, etc.)
6. Build excitation table for flip-flop inputs from transition table
7. Derive excitation equations from excitation table
8. Derive output equations from transition/output table
9. Draw logic diagram with excitation logic, output logic, and state memory elements

Synthesis is generally followed by simulation and verification

Step 1: State Table Design from Word Description

- Much like writing a computer program:
  - Start with a vague description
  - Make decisions, sometimes using common sense, and sometimes arbitrarily
  - Handle all special cases (even those not covered in the word description)
  - Design may not work, so debug and iterate

- Try it! Design a state machine with for these example systems. Each uses one input X and one output Z. Let N be the binary number that corresponds the sampled bits of X, MSB to LSB.
  - Example 1: Let Z = 1 if the first bit of N is 0.
  - Example 2: Let Z = 1 if the number of 1's = number of 0's in the current N.
  - Example 3: Let Z = 1 if the current value of N is divisible by 3.

Design Example 1: 110 Detector

- Word description (input sequence detector)
  - Design a state machine with input A and output Y.
  - Y should be 1 whenever the sequence 1 1 0 has been detected on A on the last 3 consecutive clock ticks.
  - Otherwise, Y = 0
  - Note: this is a Moore machine, that is the output, Y, depends only on inputs at previous clocks, not on the current input.

- Interpretation of word description (only rising clock edges, or ticks, are shown):
Design Example 1: Choosing States

- Possible states (What do you need to remember?)
  - Initial: power up, no clocks yet Y = 0
  - No1s: first 1 not found Y = 0
  - First1: first 1 found Y = 0
  - Two1s: at least 2 consecutive 1s found Y = 0
  - ALL: found 1 1 0 Y = 1

- Are all the states needed?
  - Notice: Initial is equivalent to No1s
  - We can drop the state Initial and replace it with state No1s

Design Example 1: State Table and Diagram

- State Table
- State Diagram

Design Example 2: 110/101 Detector

- Word description (input sequence detector)
  - Design a state machine with input A and output Y.
  - Y = 1 when either sequence 1 1 0 or 1 0 1 has been detected on A on the last 3 consecutive clock ticks.
  - Otherwise Y = 0
  - Note: Correct sequences may overlap and still be accepted

- Interpretation of word description (only rising clock ticks are shown)

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</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>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Design Example 2: Choosing States

Possible states (What do you want to remember?)

- **Idle**: Initial, no starting 1 yet \( Y = 0 \)
- **Got1**: \( A = 1 \) on last tick \( Y = 0 \)
- **Got10**: Sequence \( A = 10 \) on last two ticks \( Y = 0 \)
- **Got101**: Sequence \( A = 101 \) on last three ticks \( Y = 1 \)
- **Got11**: Sequence \( A = 11 \) on last two ticks \( Y = 0 \)
- **Got110**: Sequence \( A = 110 \) on last three ticks \( Y = 1 \)

### State Table

<table>
<thead>
<tr>
<th>( A )</th>
<th>( S(t) )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>Idle</td>
<td>0</td>
</tr>
<tr>
<td>( 1 )</td>
<td>Got1</td>
<td>0</td>
</tr>
<tr>
<td>( 1 )</td>
<td>Got10</td>
<td>0</td>
</tr>
<tr>
<td>( 1 )</td>
<td>Got101</td>
<td>1</td>
</tr>
<tr>
<td>( 1 )</td>
<td>Got11</td>
<td>0</td>
</tr>
<tr>
<td>( 1 )</td>
<td>Got110</td>
<td>1</td>
</tr>
</tbody>
</table>

### State Diagram

Format:

- **Arc**: input \( A \)
- **Node**: state/output \( Y \)
Design Example 3: Interpretation

Word description (input sequence detector)
- Design a state machine with inputs A and B, and output Z.
- Z = 1 if either:
  - A had the same value for both previous clock ticks or
  - B has been 1 ever since the first condition occurred.
- Else Z = 0

Interpretation of word description (only rising clock ticks are shown)

A
B
CLK
Z
0
1
0
1
0
1

Design Example 4: State Table

Word description (output sequence generator)
- 3-bit counter with enable (0, 1, 2, 3, 4, 5, 6, 7, 0, 1 ...)

<table>
<thead>
<tr>
<th>S</th>
<th>EN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>0   0</td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
<td>0   0</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>0   1</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>0   1</td>
</tr>
<tr>
<td>S4</td>
<td>S4</td>
<td>1   0</td>
</tr>
<tr>
<td>S5</td>
<td>S5</td>
<td>1   0</td>
</tr>
<tr>
<td>S6</td>
<td>S6</td>
<td>1   0</td>
</tr>
<tr>
<td>S7</td>
<td>S7</td>
<td>1   1</td>
</tr>
</tbody>
</table>

S(t+1)

Design Example 4: State Diagram

Format:
Arc: input EN
Node: state/output C C C C

Reset
Design Example 5: State Table

- Word description (output sequence generator)
  - Design state machine with input GO and BCD output code $B_3B_2B_1B_0$
  - Anytime $GO = 1$ at a clock tick and machine is IDLE, output the BCD sequence 1, 2, 5, 9 during the next 4 clock ticks (regardless of GO)
  - Return to IDLE with BCD = 0 until GO = 1 is next detected

<table>
<thead>
<tr>
<th>State</th>
<th>S(t)</th>
<th>Go</th>
<th>$B_3$</th>
<th>$B_2$</th>
<th>$B_1$</th>
<th>$B_0$</th>
<th>S(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>IDLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IDLE</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>S5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S5</td>
</tr>
<tr>
<td>S5</td>
<td>S9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>S9</td>
</tr>
<tr>
<td>S9</td>
<td>IDLE</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>IDLE</td>
</tr>
</tbody>
</table>

Outline

- CSSM Design
  - The synthesis process
  - State machine design

- State minimization and assignment

- Ad-hoc Design
  - Finite Memory Machines

- Timing Issues
  - Clock Skew
  - Input synchronization

State Minimization

- Minimal state table use fewest possible states
  - desirable because it usually means less hardware
- If state table created from a word problem is NOT minimal, then:
  - Identify equivalent states:
    - Two states are equivalent if both give the same current outputs for all inputs
      and all the next states are the same or equivalent for both states
    - e.g. Two states are equivalent if all their outputs are the same
  - Replace all equivalent states with a single state.
- For small designs, minimization should not be necessary if the designer is careful when creating the state table.
- For larger designs, formal state minimization procedures are often times necessary.
State Minimization - Example 2

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

\(S(t+1)\)

State Assignment

- What binary codes to choose for each state name?
- Example 1: Given 3 states A, B, C (s=3):
  - How many state bits are needed?
    \(n \geq \log_2 s\), \(n \geq 2\)
  - Given # state bits, which bit combination (state code) is used for each named state?

<table>
<thead>
<tr>
<th>State Name</th>
<th>Possible State Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00 01 01 10</td>
</tr>
<tr>
<td>B</td>
<td>01 11 00 00</td>
</tr>
<tr>
<td>C</td>
<td>10 10 11 01</td>
</tr>
</tbody>
</table>

\(\text{?} \text{ choices for A, 3 choices for B, 2 choices for C}\)
\(4 \cdot 3 \cdot 2 = 4! / 3! = 24\)

State Assignment

- Example 2: Given 5 states A, B, C, D, E and minimum # of bits (3), how many possible state assignments are there?

3 bits \(2^3 = 8\) values
\(8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 = 8! / 3! = 6720\)

\(\text{or generally,}\)
\(2^n = \text{(# of state codes)!} / \text{(# of unused states)!}\)

(actually, a bit less after considering column orders and bit inversions)
State Assignment

- Why does choice of state assignment matter?
  - BIG effect on complexity of excitation and output equations AND thus on the amount of logic.

- How to find the best state assignment?
  - The only known way is to try ALL assignments and determine the resulting equations.
    - \( N = 2 \): \((2^2)! = 4! = 24\) assignments
    - \( N = 3 \): \((2^3)! = 8! = \approx 40,000\) assignments
    - \( N = 4 \): \((2^4)! = 16! = \approx 20,000,000,000,000\) assignments for 4 state bits!!!
    - THIS IS NOT PRACTICAL!
  - Use heuristic guidelines for pretty good assignments.
  - This is an active area of research!

- There is no effective way to guarantee a "best" assignment. The heuristic methods sometimes perform poorly!

State Assignment Strategies

- Simplest Assignment
  - Straight binary, NOT best; purely arbitrary assignment

- One Hot Assignment
  - Redundant encoding, each flip-flop is assigned a state.
  - Uses the same number of bits as there are states (not useful in large designs)
  - Simple to assign, simple next-state logic (no state decoding required)
  - Output logic is simple! One OR gate per Moore output!

- Almost One Hot Assignment
  - Almost same as One Hot, but one less state bit
  - Use all 0's to represent a state (usually INIT)
  - Must now decode state 0 if it is needed

- Decomposed Assignment
  - Use the "structure" of the state table to simplify next-state and output logic
  - An "art" which requires much practice

Example: State Assignment Strategies

- Alternative Assignments

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0001</td>
<td>0010</td>
<td>1000</td>
</tr>
<tr>
<td>0010</td>
<td>1000</td>
<td>0001</td>
</tr>
</tbody>
</table>

- Almost One Hot
- One Hot
- Decomposed
- Simplest

- Example decomposition:
  - Initial state = all 0's for easy RESET
  - INIT state is different, so use \( Q_2 = 1 \) for non-INIT states, i.e. \( \text{INIT} \)
  - \( Z \) is only 2 states, so use \( Q_0 = 1 \) for states when \( Z \geq 2 \), i.e. \( \text{OK} \)
  - Use \( Q_2 = 1 \) for state transitions caused by \( A \) having the value of 1 that transitions state cause by \( A = 1 \), i.e. states \( \text{A0} \) and \( \text{OK1} \), state \( \text{OK} \)
  - \( Z \)

- THUS, simpler next state and output logic!
State Assignment - Heuristic Guidelines

Starting from the highest priority to the lowest:

- Choose initial coded state that’s easy to produce at reset: (all 0’s or 1’s)
  - This simplifies the initialization circuitry, but is not always wise.
- Freely use any of the 2^n state codes for best assignment
  (i.e., with s states, don’t just use the first s integers 0,1,…,s-1)
- Define specific bits or fields that have meaning with respect to input or output variables (decomposed codes)
- Consider using more than minimum number of state variables to allow for decomposed codes
- Minimize number of state variables that change at each transition
  - Prioritized adjacency schemes are often used in the field
- Simplify output logic

Minimum bit-change strategy: Minimize the total number of bits that change for all state transitions.

```
<table>
<thead>
<tr>
<th>State</th>
<th>Q1(t+1)</th>
<th>Q0(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Q1(t+1) = Q1’Q0 + Q1Q0’
Q0(t+1) = Q0’

Prioritized Adjacency

- Assign adjacent encodings to states based upon the following priorities
- Priority 1: Assign adjacent encodings to pairs of states that share a common next state for a given input assignment or transition expression.
- Priority 2: Assign adjacent encodings to pairs of states that are “next states” of the same state.
- Priority 3: Assign adjacent encodings to pairs of states for which have the same primary output values [over all input combinations if Mealy outputs].

```
<table>
<thead>
<tr>
<th>Priority</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority 1: (A,B) (B,C)</td>
<td>A = 00</td>
</tr>
<tr>
<td>Priority 2: (B,C) (C,D)</td>
<td>B = 01</td>
</tr>
<tr>
<td>Priority 3: (A,B)</td>
<td>C = 11</td>
</tr>
</tbody>
</table>
```
State Assignment Strategies

These heuristics do not work for all problems. If you can't minimize everything, at least try to minimize something.

Minimal Risk/Cost State Assignments

If extra unused state codes exist (number of available states ≥ s), then two choices are possible:

- **Minimal Risk**: most reliable but most expensive
  - Assume it is possible to enter unused state by noise, weird inputs, etc.
  - Include all unused states as present states, but that corresponding next states go to "initial" or "idle".

- **Minimal Cost**: somewhat risky but least expensive
  - Assume unused states NEVER entered accidentally.
  - Next state and outputs of all unused states = "don't care" to reduce next state and output logic.

Achieving both minimal risk and minimal cost is often possible!

Minimal Risk/Cost State Assignments

Consider a device with three cycles:

- (000) → (001) → (000)
- (101) → (110) → (001)
- (011) → (100) → (111) → (101) → (011)

Which is the primary cycle? Is it self-correcting?

What about the 11 state?
State Machine Design Procedure

1. Build state/output table (or state diagram) from word description
2. Minimize number of states
3. Choose state variables and assign bit combinations to named states
4. Build transition/output table from state/output table (or state diagram)
5. Choose flip-flop type (D, J-K, etc.)
6. Build excitation table for flip-flop inputs from transition table
7. Derive excitation equations from excitation table
8. Derive output equations from transition/output table
9. Draw logic diagram with excitation logic, output logic, and state memory elements

Choosing the Storage Element

- **T-type flip-flops** - excellent for “counter-type” circuits where flip-flops go between 0 and 1 with great frequency
- **D-type flip-flops** - Normally used when information must be stored and used later
  - D is the most commonly used element in VLSI design
  - Most PLDs use only D-type flip-flops
- **SR-type flip-flops** - Normally used when different signals set/reset the flip-flops
- **JK-type flip-flops** - Combined SR-type and T-type applications
  - Input logic may be less complex than a D-type flip-flop due to built-in gating.

Example 1: 1 1 0 detector

- Step 2: Minimize number of states
- Step 3: Choose state variables:
  - initial state all 0s
  - Q2 = last A, so Q2* = A
  - minimize number of transitions

From Step 1:

<table>
<thead>
<tr>
<th>A</th>
<th>Q1</th>
<th>Q2</th>
<th>S</th>
<th>T</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NO1s</td>
<td>NO1s</td>
<td>First1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>First1</td>
<td>NO1s</td>
<td>Two1s</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Two1s</td>
<td>ALL</td>
<td>Two1s</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ALL</td>
<td>NO1s</td>
<td>First1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S(t+1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 1: 1 1 0 detector

Step 4: Transition/output table
- minimum cost/risk does not apply here

<table>
<thead>
<tr>
<th>A</th>
<th>Q1</th>
<th>Q2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1 0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1 0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0</td>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>

Q1(t)=Q2(t+1) = D1, D2
Step 6

Step 5: Choose Flip-Flop type = D, so Q(t+1) = D

Step 6: Excitation table
- Same as Transition/output table with Q1(t+1)=D1, Q2(t+1)=D2

Step 7: Excitation equations: D1, D2 = F(A, Q1, Q2)

D1 = Q1•Q2 + Q2•A
D2 = A
(as planned!)

Step 8: Output equations: Y = F(Q1, Q2)
- Y = Q1•Q2 (directly read from transition table)

Step 9: Logic Diagram

Finished!
Example 2: 1 1 0 / 1 0 1 detector

Step 2: Minimize number of states
- Initial state 0s
- Q1 = Y
- Q3 = last A, so Q3(t+1) = A
- Minimum number of transitions

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>S</th>
<th>0</th>
<th>1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IDLE</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Got1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Got10</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Got101</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Got110</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>IDLE</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

From Step 1:

Step 3: Choose state variables:
- Initial state all 0s
- Q1 = Y
- Q3 = last A, so Q3(t+1) = A
- Minimum number of transitions S

Step 4: Transition/output table

Step 5: Choose Flip-flop type D

Step 6: Excitation table
- Same as transition table
- Choose minimum cost
- If choose minimum risk:
  next state = IDLE (000), Y = 0

Step 7: Excitation equations
- D1, D2, D3 = F(A, Q1, Q2, Q3)
- D1 = Q1'•Q2•Q3•A' + Q2•Q3'•A
- D2 = Q2•A + Q3
- D3 = A (as planned!)

Unused states?
Example 2: 1 1 0 / 1 0 1 detector

- Step 8: Output equation
  - \( Y = Q \) (as planned!)

- Step 9: Logic diagram
  - (3) Flip-flops + (3) 2-input gates + (1) 3-input gate + (1) 4-input gate
  - Actual diagram left to the student!

Synthesis using J-K Flip-flops

- Excitation logic **MAY** be simpler for J-K FF (than D-FF) because of built-in J-K gating
- Procedure starts the same for any type of flip-flop:
  - 1. State/output table
  - 2. State minimization
  - 3. State assignment
  - 4. Transition/output table
- But next steps differ, depending on flip-flop:
  - 5. Choose Flip-flop type
  - 6. Excitation table
- Remaining steps are basically the same:
  - 7. Excitation equations
  - 8. Output equations
  - 9. Logic diagram

Different Excitation Table Required

- Need equations for J and K
- How to get J and K values, given the Q-to-Q(t+1) transitions?
  - Convert the J-K Operation table to a J-K Application table

<table>
<thead>
<tr>
<th>J-K Operation Table</th>
<th>J-K Application table for JK=</th>
<th>J K</th>
<th>Q</th>
<th>Q'(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hold</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Reset</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toggle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 1 (again!)

Transition/output table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>0</th>
<th>1</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

J-K Excitation table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0d</td>
<td>0d,1d</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0d, d1</td>
<td>1d, d0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>d0, d1</td>
<td>d0, d0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>d1, 0d</td>
<td>d1, 1d</td>
</tr>
</tbody>
</table>

J1 K1, J2 K2

Example 1 (again!)

Transition/output table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>0</th>
<th>1</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

J-K Excitation table

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0d</td>
<td>0d,1d</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0d, d1</td>
<td>1d, d0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>d0, d1</td>
<td>d0, d0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>d1, 0d</td>
<td>d1, 1d</td>
</tr>
</tbody>
</table>

J1 K1, J2 K2

Other D-FF Synthesis Examples:

1's Counter

- 1's Counter Requirements:
  - Design state machine; input X, Y and output Z
  - Z = 1 if # of 1 inputs on X and/or Y is a multiple of 4
  - Else Z = 0
  - So count = 0, 4, 8, 12 ... give Z = 1
  - Count increases by 0, 1, or 2 on each clock
- What do you need to remember?
- What states do you need?
### 1's Counter: State/output table

<table>
<thead>
<tr>
<th>Meaning</th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Got zero 1s (modulo 4)</td>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>Got one 1 (modulo 4)</td>
<td>S1</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S2</td>
<td>0</td>
</tr>
<tr>
<td>Got two 1s (modulo 4)</td>
<td>S2</td>
<td>S2</td>
<td>S3</td>
<td>S0</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>Got three 1s (modulo 4)</td>
<td>S3</td>
<td>S3</td>
<td>S0</td>
<td>S1</td>
<td>S0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( S(t+1) \)

### 1's Counter: Transition/excitation table

<table>
<thead>
<tr>
<th>XY</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>

\( Q1(t+1) Q2(t+1) = D1 \cdot D2 \)

State assignment: Karnaugh map order

### 1's Counter: K-maps

\[ D1 = Q2 \cdot X \cdot Y + Q1 \cdot X \cdot \overline{Y} + Q1' \cdot X \cdot Y + Q2 \cdot X \cdot \overline{Y} \]

\[ Z = Q1' \cdot Q2 \quad \text{(by inspection)} \]
Outline

- CSSM Design
  - The synthesis process
  - State machine design
  - State minimization and assignment
- Ad-hoc Design
  - Finite Memory Machines
- Timing Issues
  - Clock Skew
  - Input synchronization

Ad Hoc Design

- Directly translate word description into a circuit
  - Use Flip-Flops to "remember" things
  - Use combinational logic to decode these remembered things
  - Usually simplifies the state assignment process because storage elements remember something meaningful
- Useful only for small state machines
- May be easier or harder to design
- May require more or less complex hardware
- With an Ad Hoc design, the ease of design and complexity are unknown before the design is attempted
- Larger designs are often decomposed into smaller more easily solved problems and then recombined ad-hoc!
  - e.g.: Control and Data units

1's Counter: K-maps

\[ D_2 = Q'_1 X' Y + Q'_1 X Y' + Q_2 X' Y' + Q_2' X Y \]
Example Ad Hoc Problem

Design a state machine with inputs A, B and output Z. Z=1 if A and B inputs were EQUAL for the last 2 clock ticks OR if B has been 1 ever since the first condition was true. Else, Z=0.

Solve the two conditions separately, then OR the results:

a) 1st Condition - Serial Comparator: Remember last 2 comparison results, then AND together. \( \rightarrow \) SAMEBOTH output

b) 2nd Condition - Remember if B has been 1 since 1st condition true \( \rightarrow \) BOK output

Example Ad Hoc Problem

1st Condition - Serial Comparator

![Diagram of 1st Condition]

Example Ad Hoc Problem

1st and 2nd Condition Or'd Together:

![Diagram of 1st and 2nd Condition Or'd Together]
Finite Memory Machines (**FMMs**) are one type of Ad Hoc design.

- Outputs are determined by:
  - Current inputs
  - and \(n\) previous inputs
  - and \(m\) previous outputs

- FMMs are a subclass of Finite State Machines, but FSMs are not necessarily FMMs:
  - an FSM may depend on all past inputs (forever)
  - a FMM can only depend on a finite number of past inputs

---

**Finite Memory Machine**

- flip-flops store previous inputs
- flip-flops store previous outputs

---

**Ad-hoc Implementation with MUX**

- When multiple I/O bits are needed, the implementation uses multiplexers (MUX) to decode the state of the flip-flops and select the appropriate output.
Outline

- CSSM Design
  - The synthesis process
  - State machine design
- State minimization and assignment
- Ad-hoc Design
  - Finite Memory Machines
- Advanced Timing Issues
  - Clock Skew
  - Input synchronization

Synchronous System Example

Combinational logic

State Register

Q3

Q2

Q1

\( t_{cest} = 2 \text{ ns, min and 20 \text{ ns, max}} \)

\( t_{setup} = 5 \text{ ns, min} \)

\( t_{hold} = 2 \text{ ns, min} \)

Max Frequency?

\( t_{clk,min} \geq (15 + 20 + 5) \text{ ns,} \)

\( \Rightarrow f_{max} \leq 25 \text{ MHz} \)

Setup margin @ 10 MHz clk?

\( 100 - (15 + 20 + 5) = 60 \text{ ns} \)

Hold Margin?

\( (3 + 2) - 2 = 3 \text{ ns} \)

Clock Skew Introduction

Clock skew = Difference in arrival times of the clock signal to different devices - this is a BAD thing!

Example of clock skew:

- FF2 loads NEW state of Q1, not OLD
- May violate setup/hold time, causing metastability
Clock Skew

CASES:
1. Q2 misses old data value
2. Violates setup time
3. Violates hold time
4. OK

Clock Skew - Causes and Fixes

- Why does skew occur? How do we fix it?
- A single clock signal is insufficient in a large system to drive many loads, so we need to buffer the clock
  - but improper buffering causes skew

Asynchronous Primary Inputs

- Real Systems are rarely fully synchronous
  - Can you type at 16 MHz?
- Digital systems of all types must deal with asynchronous input signals
  - asynchronous signals are not synchronized with the clock
  - how does this affect setup and hold time guarantees?
- A synchronizer is a circuit that samples an asynchronous input and produces an output that meets setup and hold times required in a synchronous system.
Question: How do we guarantee setup and hold times for the synchronizer? SYNCIN may be a metastable value!
   - When the synchronizer itself becomes metastable, we say that the synchronizer has failed.
     - The actual clock tick upon which the ASYNC signal finally arrives is not important (as it is asynchronous)
     - By reducing the clock rate, we reduce the probability that the hold and setup times of the synchronizer will be met (but we don’t want to slowdown the entire system)
     - By adding another flip-flop to the design, we have time to “recover” from a metastable state (with some probability).

Improving synchronizer reduces the probability of failure, but does not remove it.
   - There is always a chance (however small) that a asynchronous signals comes in at exactly the wrong time and produces metastable behavior.