# Chapter 5

**SEQUENTIAL CIRCUITS:**
***Small Designs***



## In this Chapter

- Sequential circuit design models
- Design examples:
  - registers, counters, sequence recognizer
- Sequential circuit timing
- Interfacing sequential circuits



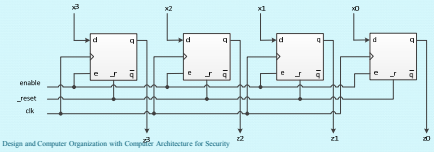
## Sequential Circuit as a Finite State Machine (FSM)

- Requires flip-flop(s) to save circuit state
- Requires combinational circuit(s) to generate next circuit state and output(s)
- Two types of combinational circuits:
  - Functions of the circuits cannot be determined in advance
    - FSM design is modeled with a finite state diagram (FSD)
  - Functions of the circuits can be determined in advance
    - E.g., MUX, adder, ALU
    - No need for an FSD



## A simple design Example (Parallel-load register with enable)

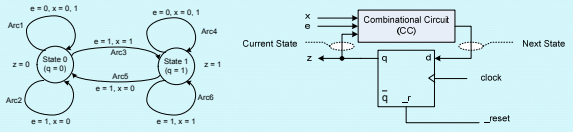
- Assume unknown combinational circuit
  1. Design 1-bit register 1st
     - We have seen D flip-flop with enable in Ch4
     - Here, the flip-flop formally modeled as FSD (next slide)
  2. Combine register slices to create 4-bit register below
     - *enable*, *clk*, *_reset* connect to all






## FSM design steps (1-bit parallel-load register example)

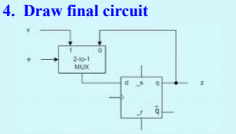
1. Draw FSD and detailed block diagram of 1-bit register slice




2. Convert FSD to truth table (also called transition table)

| Current State | External Inputs | | Next State |
|---|---|---|---|
| *q* | *e* | *x* | *d* |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

4. Draw final circuit



3. Find minimal SOP or POS expression(s) for the output(s)

$d = \bar{e}q + ex$ Ah, a MUX!

Circuit shown 1st in Chap. 4.

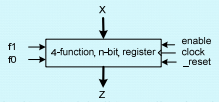


## Some design rules

- If cannot determine function(s) of combinational circuit(s) in advance:
  1. Model FSM as FSD
     - May need to design bit-slice 1st
  2. Determine number of flip flops
  3. Convert the FSD to truth table
  4. Find minimal expressions for next state variable(s) and output(s)
  5. Draw the complete circuit with flip-flops
- Otherwise
  - Use bit-serial design with known modules
  - Or, bit-parallel design with known modules

# Multi-Function Register

- Performs one of many functions
- Can be designed bit-serial or bit-parallel
- If cannot determine function(s) of combinational circuit(s) in advance
  - Model a bit-slice (e.g., 1-bit) as FSD
- Otherwise: Oh, I can use a MUX
  - Bit-serial: Use 1-bit 4-to-1 MUXs, for example
  - Bit-parallel: Use n-bit 4-to-1 MUX



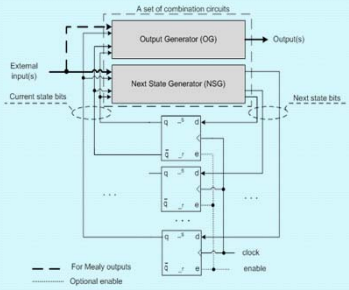

| f1 | f0 | Action |
|---|---|---|
| 0 | 0 | Clear (synchronous reset) |
| 0 | 1 | Load X |
| 1 | 0 | Arithmetic shift (shifts right repeating the sign bit) |
| 1 | 1 | Right shift (shifts right entering 0 from left) |



# FSM (A formal view)

Three types of FSMs:

- Moore
  - External inputs do not synchronously affect outputs
  - Outputs called Moore
- Mealy
  - External inputs synchronously affect outputs
  - Outputs called Mealy
- Hybrid
  - Generates both Moore and Mealy outputs

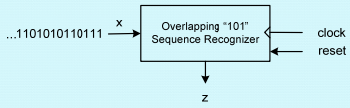





# Sequence Recognizer

Suppose FSM recognizes overlapping input sequence "101":

- Inputs processed one bit at a time (one per clock cycle)
- Not possible to determine the functions of the combinational circuits in advance
  - Model FSM as FSD
- Reset initializes the machine to known state
- Can be designed either as Moore or Mealy machine






# Moore Sequence Recognizer ('101")

- FSD has 4 states (4 bobbles)
  - E.g., labeled A to D
    - E.g., A being the initial state
    - D being the acceptance state where output *z* becomes 1
- *z* is called Moore output
  - *I.e.*, synchronized to *clk*, can only change on *clk* edge



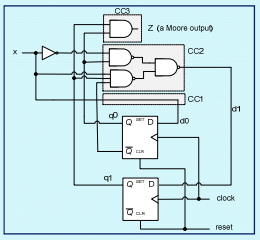
# Ways to encode FSM states

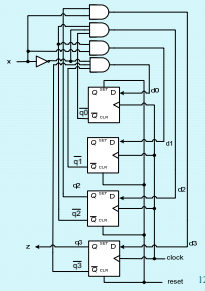
- Binary encoded states
  - Assign unique binary numbers to states (each bobble in FSD)
  - Advantage: Minimum number of flip-flops
  - Disadvantage: More complex combinational circuits
- One hot design
  - Use one flip-flop per state
    - Only one flip-flop is set during each clock cycle
  - Disadvantage: Requires maximum number of flip-flops
  - Advantage: Less complex combinational circuits
  - Better with PLDs (e.g., FPGAs)



# Example

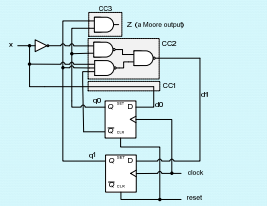
- Binary Encode State Values
  - Fewer flip-flops
  - Relatively more complex CCs
- One-hot State Values
  - More flip-flops
  - Simpler CCs



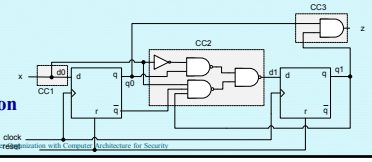



# Circuit Layouts

Layout matches the formal view




Typical implementation layout:






# How to deal with unknown states (if any)

1. Unknown FSM states are ignored in the design
   - Write don't cares in the NSG and OG truth tables
     - Help further minimize combinational circuits
     - Machine must be reset if enters unknown states
2. Unknown states are transitioned to a known state
   - Machine recovers after one clock cycle
   - Some inputs may be lost
3. Fault tolerant FSM
   - Machine can detect error and correct itself
     - I.e., continues to operate normally



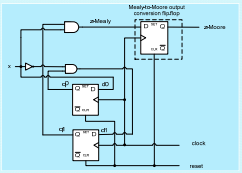
# Mealy Sequence Recognizer ("101")

- Requires 3 states
- E.g., labeled A to C
  - Binary encoded states results in one unknown state D
- Output *z* assigned to arcs
  - *z* called Mealy output synchronized to both *clk* and input *x*
    - If *x* changes, *z* can change before *clk* edge



# Converting Mealy outputs to Moore outputs

- Some designs are easier as Mealy
- Sometimes Moore outputs are preferred
- Use flip-flops to convert Mealy outputs to Moore outputs





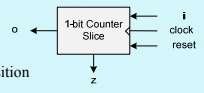

# Counters

- Generate sequence of numbers, one per clock cycle
  - Binary counter, also called mod-k counter
    - E.g., Mod-4 counter counts 0, 1, 2, 3, and repeats
  - BCD counter
    - Counts 0, 1, 2, …, 9, and repeats
  - Gray code counter
    - 1-bit difference between consecutive numbers
    - Application example: Two FSM accessing a shared FIFO buffer
      - See Exercise section
  - Other sequences with fixed patterns
    - E.g., 0, 3, 6, 9, etc.
  - An arbitrary sequence
- Can count up or down
- Designed as bit-serial, bit-parallel, or hybrid



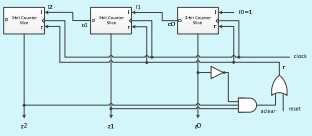
# Mod-8 Up-Counter

- Counts 0, 1, 2, 3, 4, 5 ,6, 7, and repeats
- Design options
  - Bit-parallel
    - Requires a 3-bit adder
  - Bit-serial
    - E.g., for 1-bit slice, each slice is mod-2 counter
      - 1st slice counts every clock cycle
      - 2ns slice counts every other clock cycle,
      - 3rd slice counts every 4th clock cycle
    - Designed as hybrid FSM
      - *i* decides slice counts up or retains
      - *z*-Moore is counter bit
      - *o*-Mealy is input *i* for the next slice
        - *o* asserted each time z makes 1-0 transition

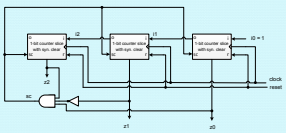


# Mod-6 Up-Counter

- Counts 0, 1, 2, 3, 4, 5, and repeats
- Partially mod-8 counter
- Wrong way to design:
  - Reset counter asynchronous when counter reaches 6
- Right way to design:
  - Reset counter synchronously when count becomes 5
    - Modelled in FSD



The wrong way!
(Using asynchronous reset)

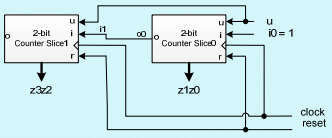


The right way!
(using synchronous reset)



# 8-bit up/down counter (bit-serial)

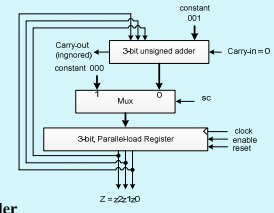
- Assume 2-bit slices
- *i* decides next higher slice shall count up or down depending on *u*
- Design a 2-bit counter FSD






# Mod-8 Up Counter (bit-parallel)

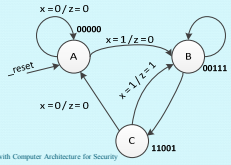
- Assume implements synchronous reset (clear)
- Required modules can be determined in advanced:
  - 3-bit adder
    - One input set to 1
    - Initial carry = 0
  - 3-bit 2-to-1 MUX
    - One input set to 0
  - 3-bit parallel-load register
- Less hardware in final circuit
  - Adder and MUX each with one fixed input will simplify to simpler expressions






# Fault Tolerant FSM

- Flip-flop outputs may change due to environmental hazards (e.g., static electricity)
- Use Hamming codes as binary encoded states
  - Codes are separated with Hamming distance of 3 or more
- A 1-bit change in a state number can be detected and corrected
  - 1-bit change has 1 Hamming distance from valid state
  - 1-bit change has 2 or more Hamming distances away from other states
- Design options:
  1. Design as you would an FSM but enter single bit faults in Truth Table
  2. Design as you would non-fault tolerant FSM but then add Hamming code generation, error detection, and correction circuits






# Hamming Coding Scheme and Error Detection

- Example: Consider 7-bit Hamming codes with 4-bit data 1001
  - Use set of XOR gates to generate three parity bits

$p_1 = d_3 \oplus d_5 \oplus d_7$
$p_2 = d_3 \oplus d_6 \oplus d_7$
$p_4 = d_5 \oplus d_6 \oplus d_7$

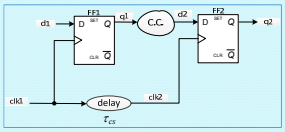
7-bit Code: $d_7 d_6 d_5 p_4 d_3 p_2 p_1 = 1001100$, $p4p2p1 = 100$

- Faults can happen when 7-bit code is stored in memory or FFs or transmitted
- Fault detection and correction
  - E.g., 7-bit code loaded in FFs as $d'_7 d'_6 d'_5 p'_4 d'_3 p'_2 p'_1 = (1101100)_2$
    - Note, bit 6 has changed due to 1-bit fault
  - Calculate parity bits ($p''$) from $d'$ bits
  - Compare $p''$ bits (100) with $p'$ bits (010)
    - Use bit-wise XOR
    - E = e2e1e0 = 100 ⊕ 010 = 110 , indicates bit 6 is in error
  - Can use a 3-to-8 decoder and XOR gates to correct the error bit



# Sequential Circuit Timing (1)

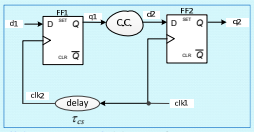
- Clock skew due to delay ($\tau_{CS}$) in clock signal
- Clock skew can create circuit timing problems
  1. If Δcc < $\tau_{CS}$, FF2 will load $d2^{New}$ and not $d2^{Current}$
     - FSM results in functional error
  2. If Δcc ~= $\tau_{CS}$, *d2* may be still changing when *clk2* arrives
     - FSM malfunction due to setup or hold time violation at FF2
- Δcc > $\tau_{CS}$, FF2 will load $d2^{Current}$ as it should
  - Circuit will function correctly




*clk1* reaches FF1 1st and after some delay as *clk2* to FF2

# Sequential Circuit Timing (2)

- Possible problem: Next *clk1* reaches FF2 before previous *clk2* reaches FF1
  - If Δcc ~= $\tau - \tau_{cs}$, *d2* may be still changing when *clk1* arrives
    - FSM malfunction due to setup or hold time violation at FF2
- Normal operation
  - When Δcc < $\tau - \tau_{cs}$




*clk1* reaches FF2 1st and after some delay as *clk2* to FF1



# Clock Frequency Estimation – With Clock Skew

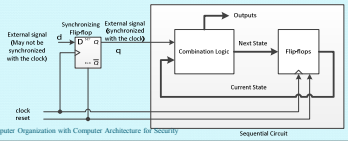
Add $\tau_{cs}$ to the clock period determined in Ch. 4

$$\tau \geq \tau_{cq-max} + \tau_{pd-max} + \tau_{st} + \tau_{cs}$$



# Interfacing Sequential Circuits

- How to handle asynchronous inputs
  - Asynchronous inputs may change at any time with respect to *clock* signal
    - They can cause setup or hold time violation if directly enter FSM
- Solution: Use synchronization flip-flop(s)
  - Asynchronous input *d* may violate setup or hold time of synchronizing FF, not FFs of FSM
    - *q* of synchronizing FF will eventually stabilize if setup or hold time violated
  - May use two synchronizing FFs if *q* slow to stabilize






# FSM HDL Model (All Behavioral)

- E.g., consider modeling sequence "101" recognizer
- Single module with three "always" blocks:

1. Combinational circuit NSG
   - Use "case" statement to list states
     - E.g., Moore: four states A, B, C, and D (the *q* bits)
   - Use "if-else" statements to model FSD arcs
     - Uses current state A, B, C, or D and input *x* to model next state (the *d* bits)
2. Combinational circuit OG
   - Use "case" and/or "if-else" to model outputs
     - Moore: if current state is D then *z* = 1 else *z* = 0
     - Mealy: if current state is D and *x* = 1 then *z* = 1 else *z* = 0
3. Flip-flops
   - Model FFs with either asynchronous or synchronous reset and enable (if necessary)